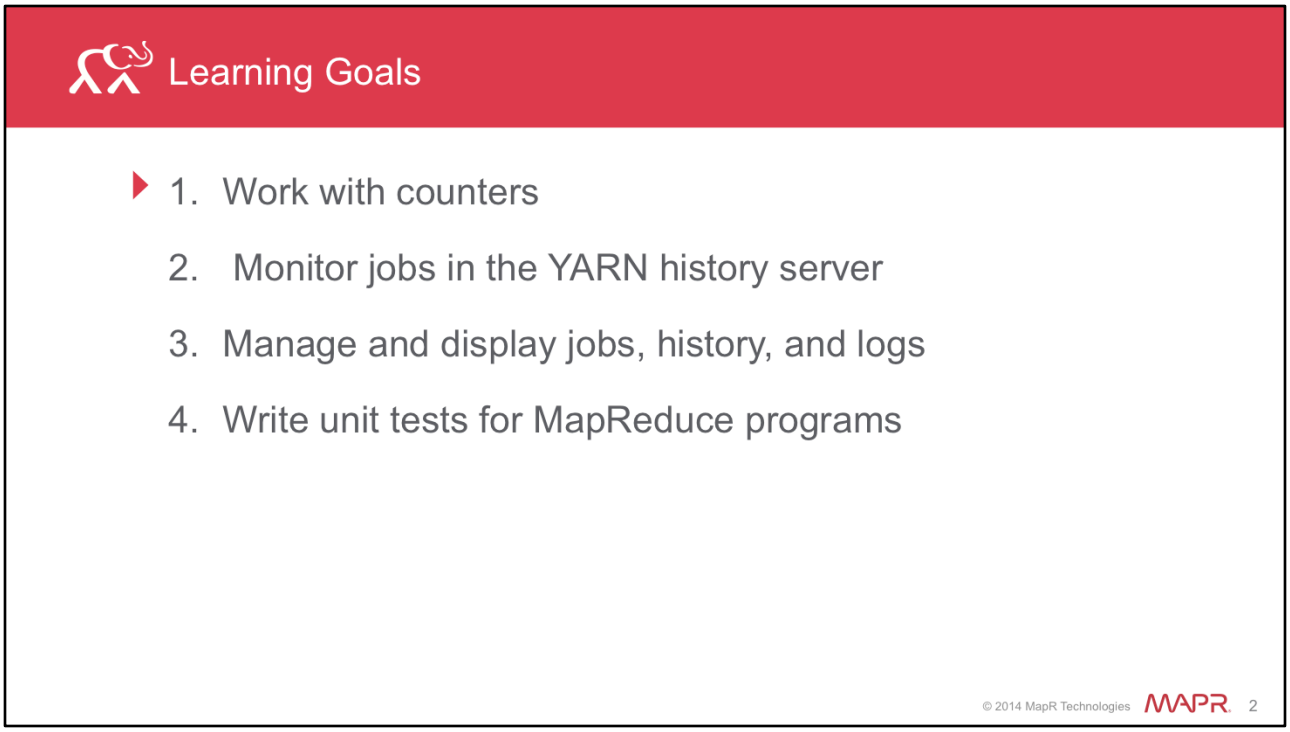
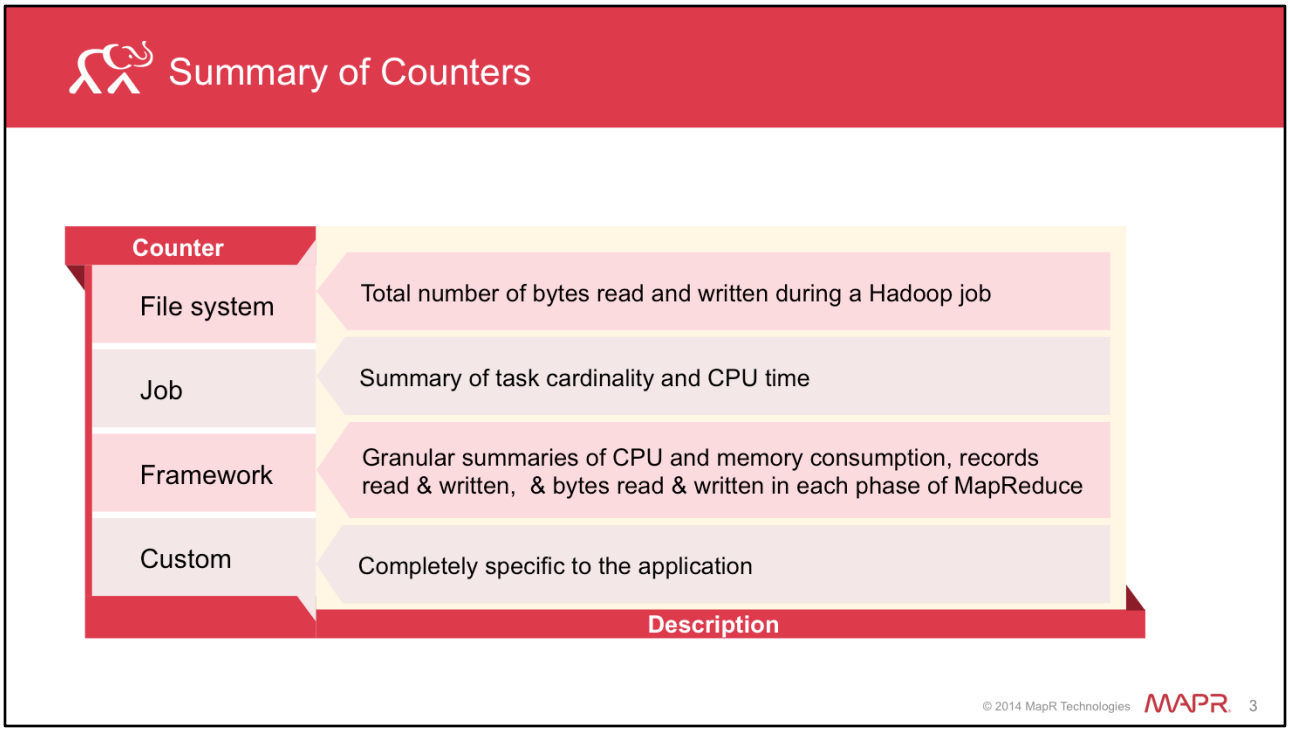

MAPR Academy
Managing, Monitoring, and
Testing MapReduce Jobs
© 2014 MapR Technologies MAPR 1



In this lesson you will learn how to work with counters, use the MCS to monitor jobs. You will learn how to manage an display jobs, history, and logs using the command line interface. You will learn how to write unit tests for MapReduce programs.

Let's begin by working with Counters.

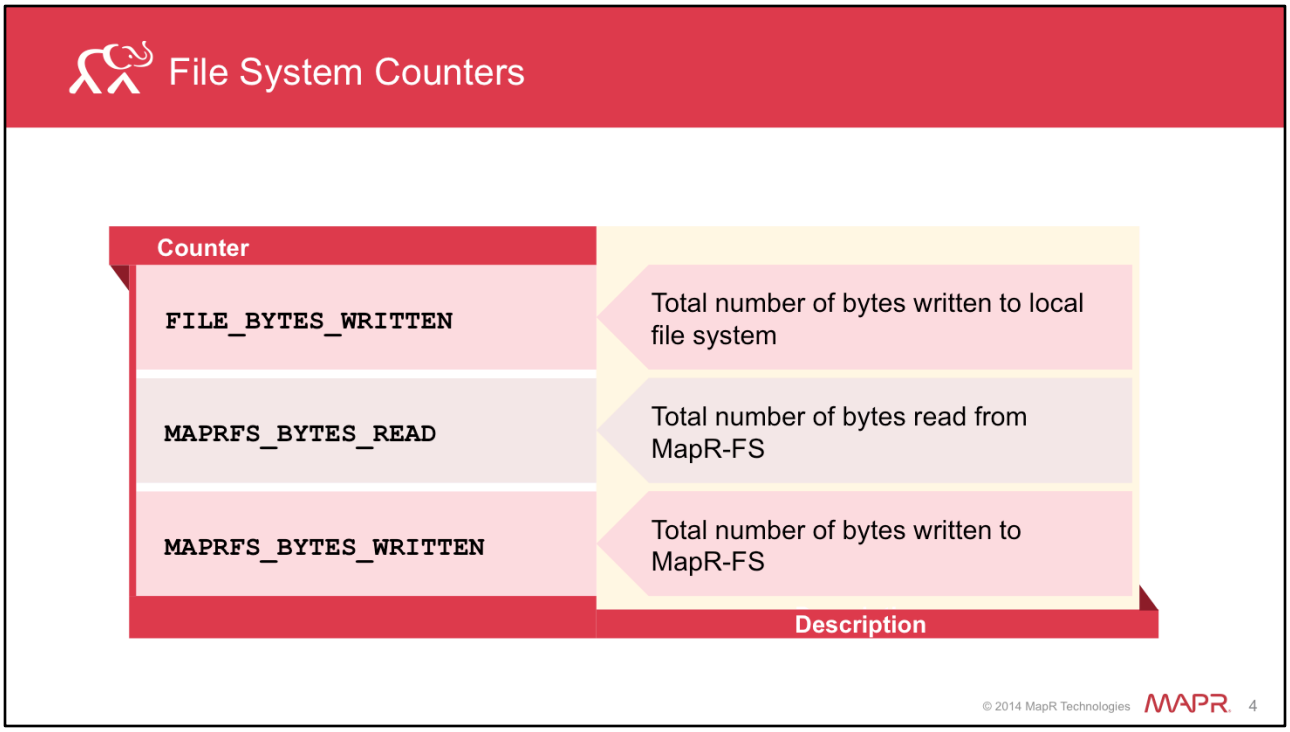


There are 4 categories of counters in Hadoop: file system, job, framework, and custom. Click on each to learn more.

You can use the built-in counters to validate that:

- The correct number of bytes were read and written
- The correct number of tasks were launched and successfully ran
- The amount of CPU and memory consumed is appropriate for your job and cluster nodes
- The correct number of records were read and written

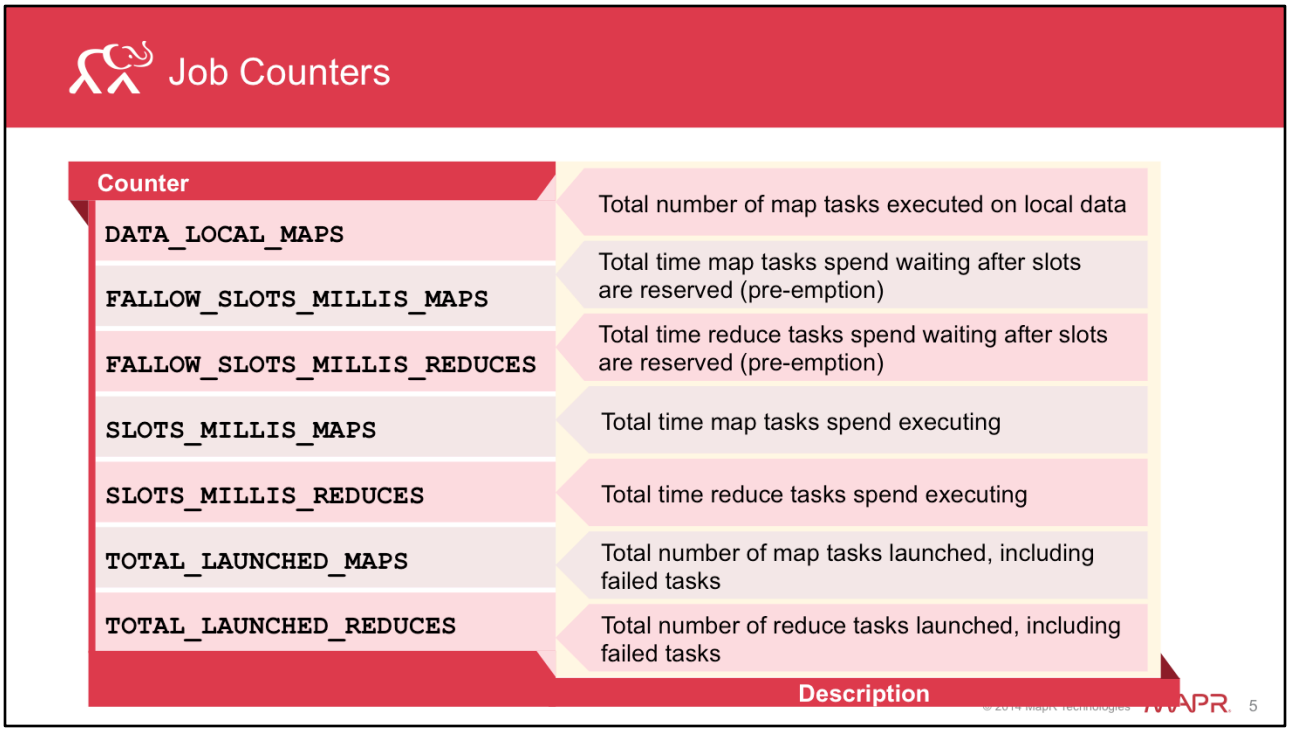
You can also configure custom counters that are specific to your application. We will discuss this later in this section.



The FILE_BYTES_WRITTEN counter is incremented for each byte written to the local file system. These writes occur during the map phase when the mappers write their intermediate results to the local file system. They also occur during the shuffle phase when the reducers spill intermediate results to their local disks while sorting.

The off-the-shelf Hadoop counters that correspond to MAPRFS_BYTES_READ and MAPRFS_BYTES_WRITTEN are HDFS_BYTES_READ and HDFS_BYTES_WRITTEN.

Note that the amount of data read and written will depend on the compression algorithm you use, if any.

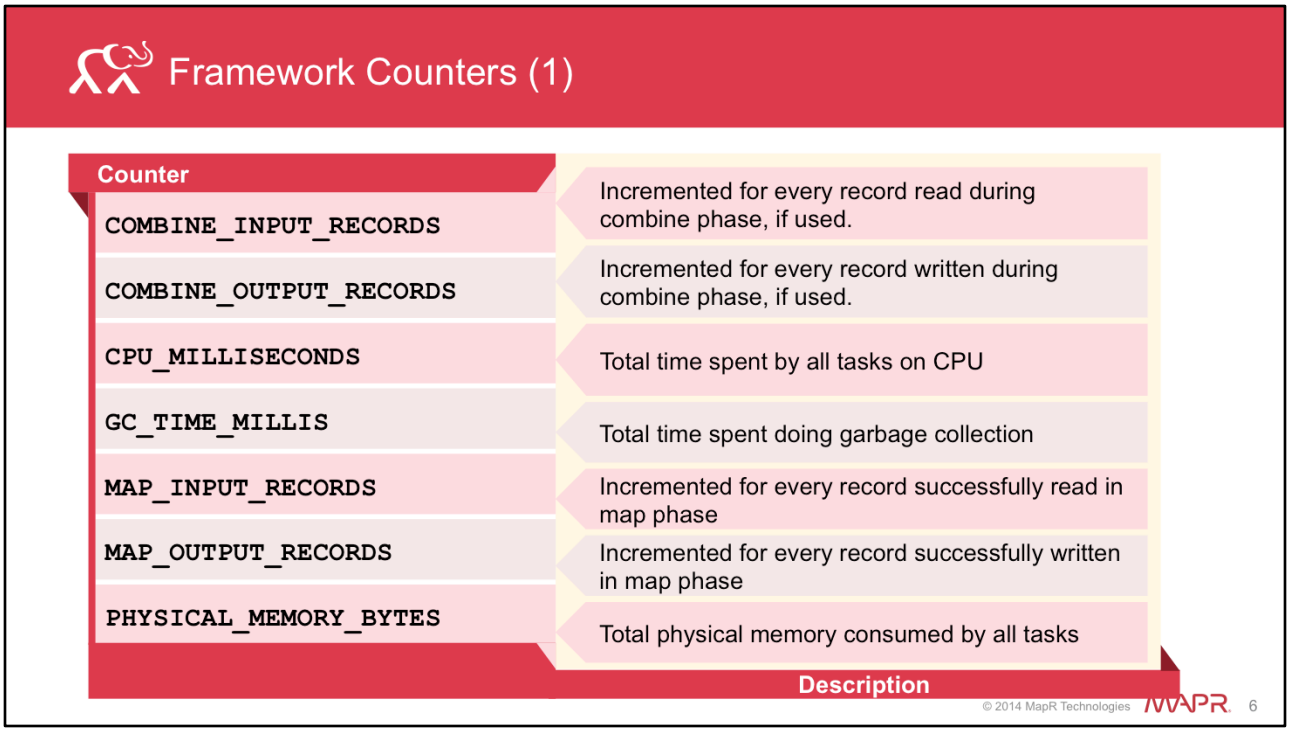


The table above describes the counters that apply to Hadoop jobs.

The DATA_LOCAL_MAPS indicates how many map tasks executed on local file systems. Optimally, all the map tasks will execute on local data to exploit locality of reference, but this isn't always possible.

The FALLOW_SLOTS_MILLIS_MAPS indicates how much time map tasks wait in the queue after the slots are reserved but before the map tasks execute. A high number indicates a possible mismatch between the number of slots configured for a task tracker and how many resources are actually available.

The SLOTS_MILLIS_* counters show how much time in milliseconds expired for the tasks. This value indicates wall



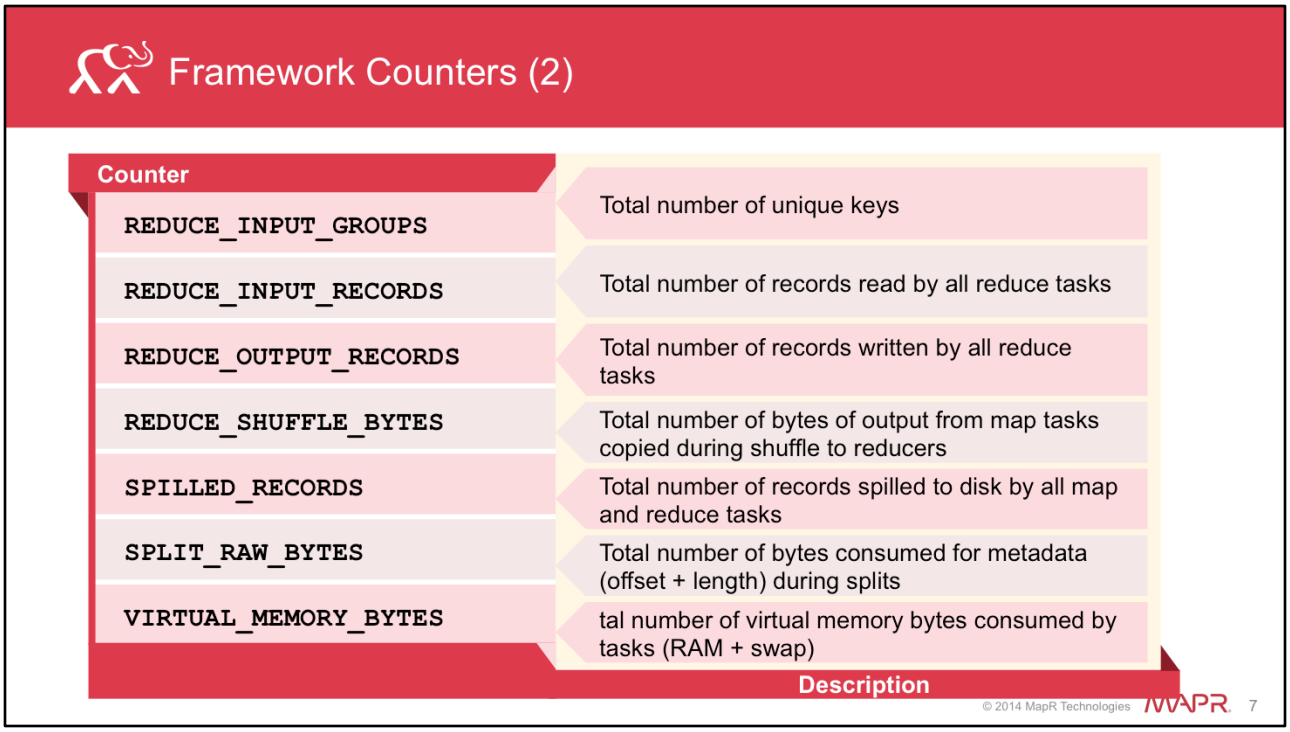
The COMBINE_* counters show how many records were read and written by the optional combiner. If you don't specify a combiner, these counters will be 0.

The CPU statistics are gathered from /proc/cpuinfo and indicate how much total time was spent executing map and reduce tasks for a job.

The garbage collection counter is reported from GarbageCollectorMXBean.getCollectionTime().

The MAP*RECORDS are incremented for every successful record read and written by the mappers. Records that the map tasks failed to read or write are not included in these counters.

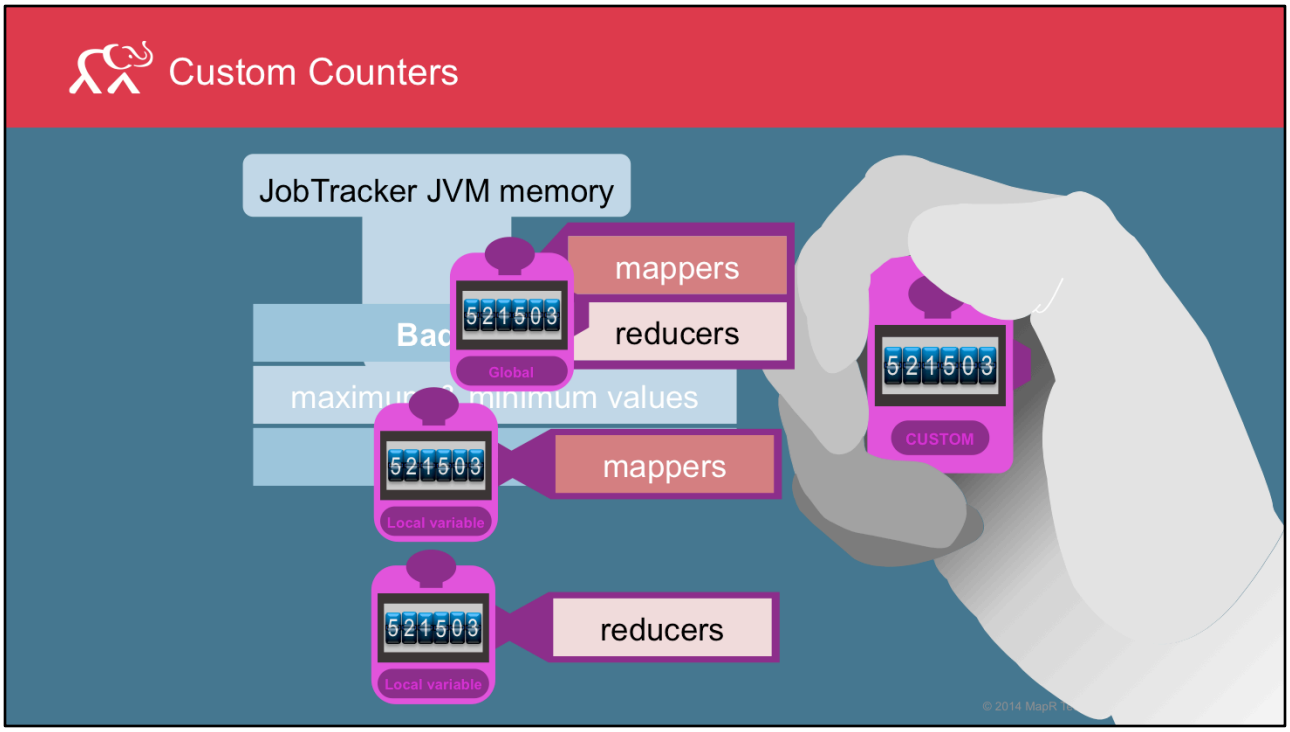
The PHYSICAL_MEMORY_BYTES statistics are gathered from /



The REDUCE_INPUT_GROUPS counter is incremented for every unique key that the reducers process. This value should be equal to the total number of different keys in the intermediate results from the mappers.

The REDUCE*RECORDS counters indicate how many records were successfully read and written by the reducers. The input record counter should be equal to the MAP_OUTPUT_RECORDS counter.

The REDUCE_SHUFFLE_BYTES counter indicates how many bytes the intermediate results from the mappers were transferred to the reducers. Higher numbers here will make the job go slower as the shuffle process is the primary network consumer in the MapReduce process.



You can optionally define custom counters in your MapReduce applications. They are useful for counting specific records such as Bad Records, as the framework counts only total records Custom counters can be used to count outliers such as example maximum and minimum values, and for summations.

Counters may be incremented (or decremented) in all the mappers and reducers of a given job. They are referenced using a group name and a counter name, and may be dereferenced in the driver and reported in the job history.

All the counters whether custom or framework are stored in the JobTracker JVM memory, so there's a practical limit to the number of counters you should use. The rule of thumb is to use less than 100, but this will vary based on physical memory

Creating Custom Counters

- **Use custom counters for**
  - Counting specific records (e.g. "bad records")
  - Keeping track of outliers
  - Summations
- **How to define custom counters**
  - As an enum object ➔ compile-time binding
  - As a context variable (counterGroup, counterName) ➔ runtime binding
- **All counters stored in memory**
  - JobTracker/Resource Manager JVM stores counters
  - Limit the number of custom counters to under 100

 9

You can optionally define custom counters in your MapReduce applications. They are useful for counting specific records (the framework counts only total records), counting outliers (e.g maximum and minimum values), and for summations.

There are two ways to define a custom counter: as an enum object and as a context variable.

All the counters (custom and framework) are stored in the JobTracker JVM memory (in MRv1) or Resource Manager (in MRv2), so there's a practical limit to the number of counters you should use. The rule of thumb is to use less than 100, but this will vary based on physical memory capacity.

Example Custom Counter

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

. . .
   if (somecondition)
      context.getCounter("MYGROUP", "MYCOUNTER").increment(1);
. . .
}
```

NOTE: you don't declare or initialize counters – that is done by the framework the first time you increment one.

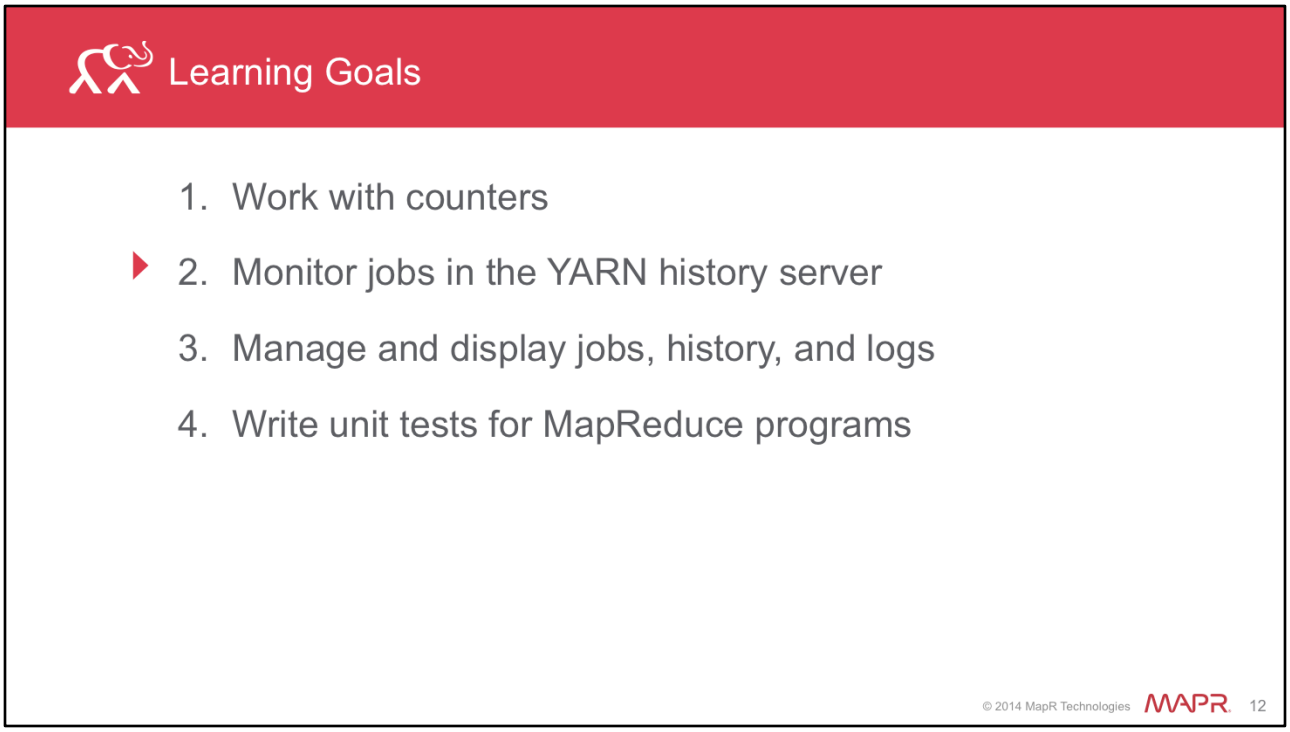
NOTE: if you never increment a counter during a job, it will not appear in the job output.

The example above shows using a custom local counter ("MYGROUP", "MYCOUNTER") in the map tasks.

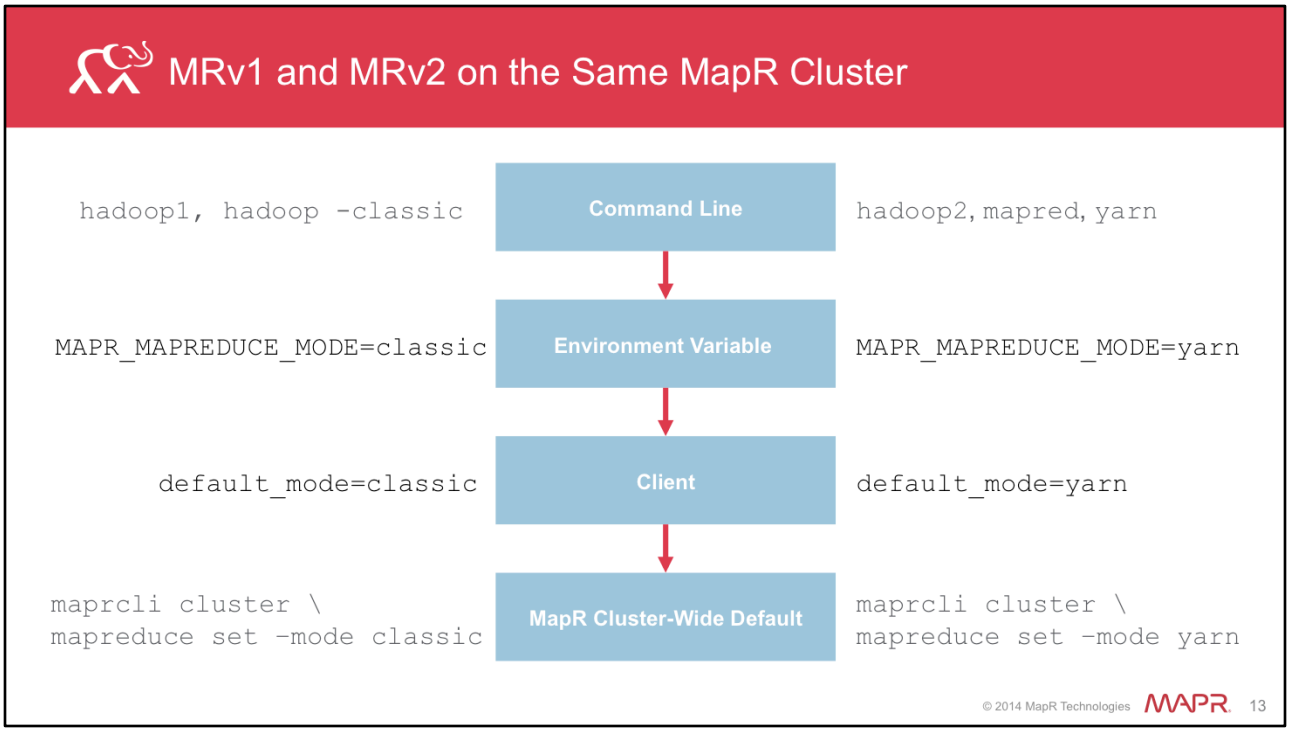
Knowledge Check

You can use the built-in counters to validate that:

1. The correct number of bytes were read and written
2. The correct number of tasks were launched and successfully ran
3. The amount of CPU and memory consumed is appropriate for your job and cluster nodes
4. The correct number of records were read and written
5. All of the above



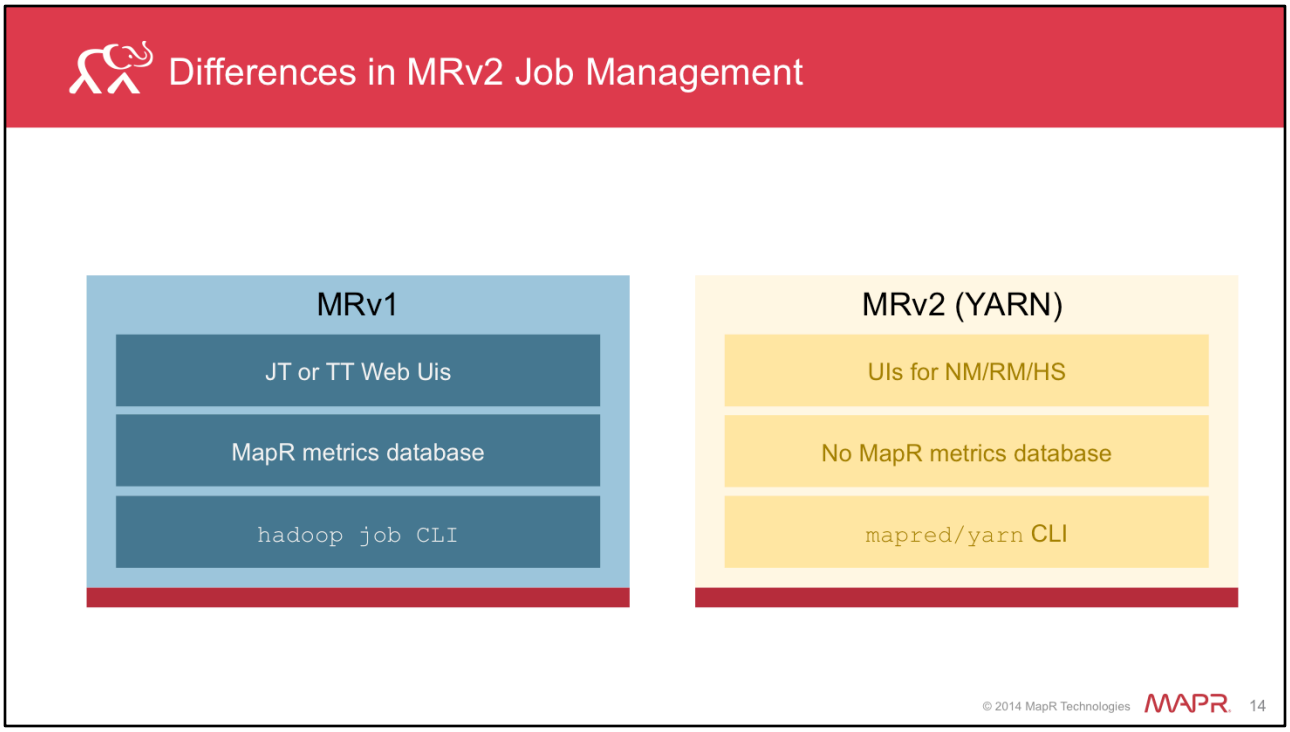
In this section, we will discuss using the MCS to monitor mapreduce jobs.



Users submitting and managing jobs can control which version of MapReduce to use by invoking a particular command. For MRv1, users run hadoop1 or use hadoop with the –classic option set. For MRv2, users run hadoop2 or mapred or yarn. The hadoop command itself is linked to MRv2 when both versions are installed.

For ecosystem components that use mapreduce under the hood, you can set their running mode by modifying the MAPR_MAPREDUCE_MODE environment variable in the appropriate ecosystem configuration file.

MapR clients submitting jobs from outside the cluster can set a parameter called default_mode in their /opt/mapr/conf/hadoop_version configuration file. This setting will override the default setting on the cluster itself.

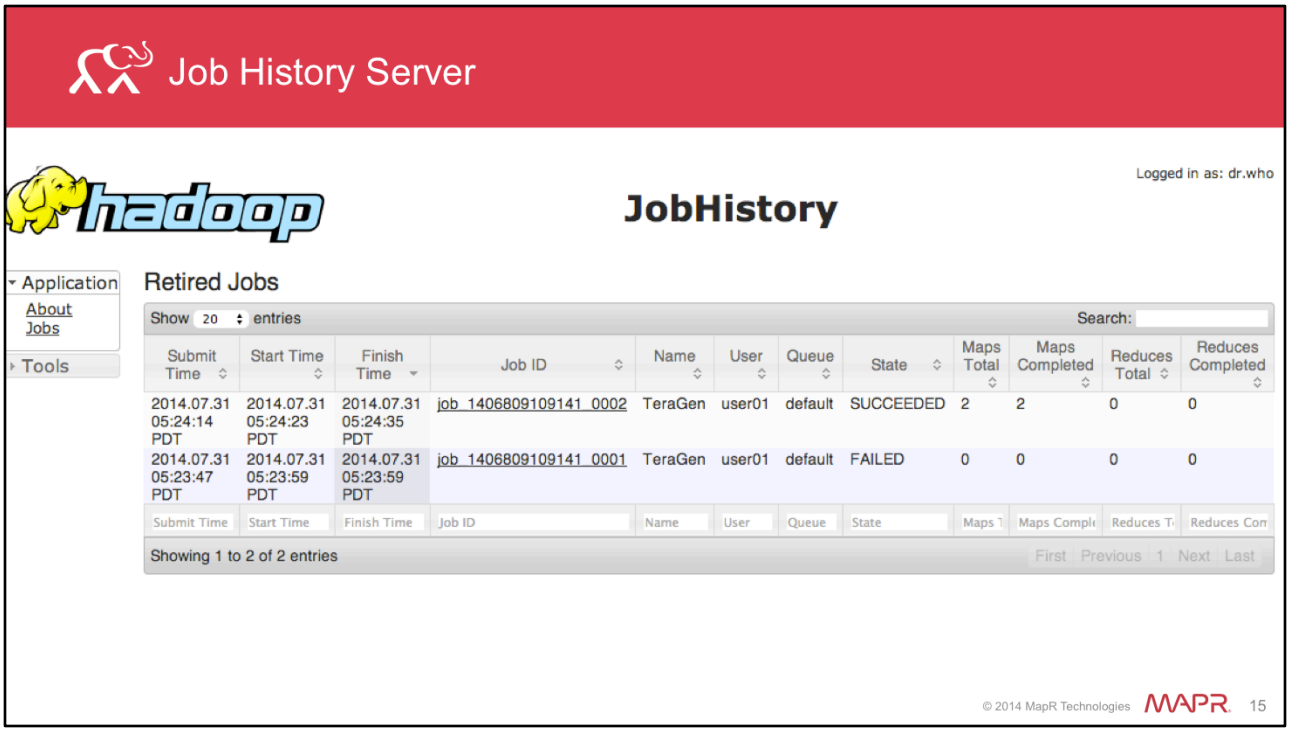


Let's look at the high-level differences of YARN job management.

In MRv1, you can use the Job Tracker or Task Tracker Web Uis to monitor your jobs. in YARN, there is a history server which runs a Web UI.

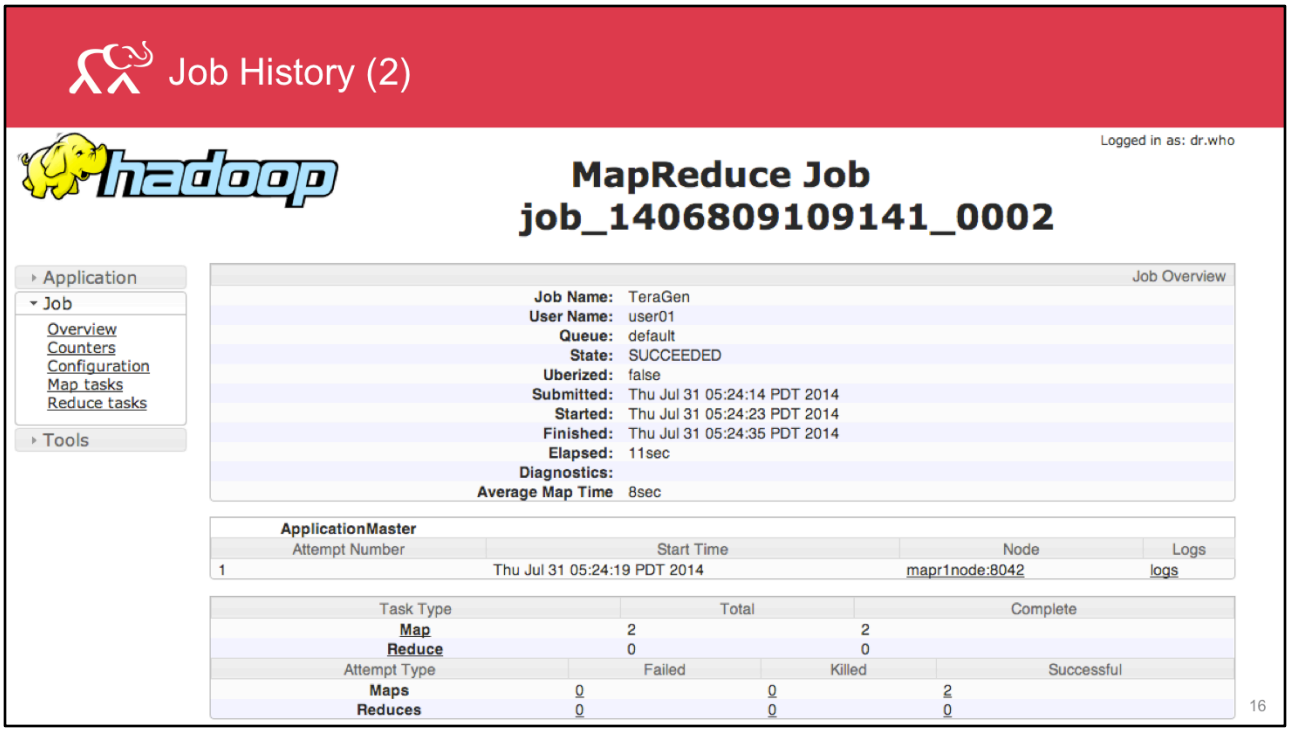
In MRv1, you can use the metrics database (available through the MCS) to monitor jobs. The metrics database is not available for YARN.

You can use the hadoop job command to manage and monitor jobs in MRv1. In YARN, use the mapred or yarn command to monitor jobs.

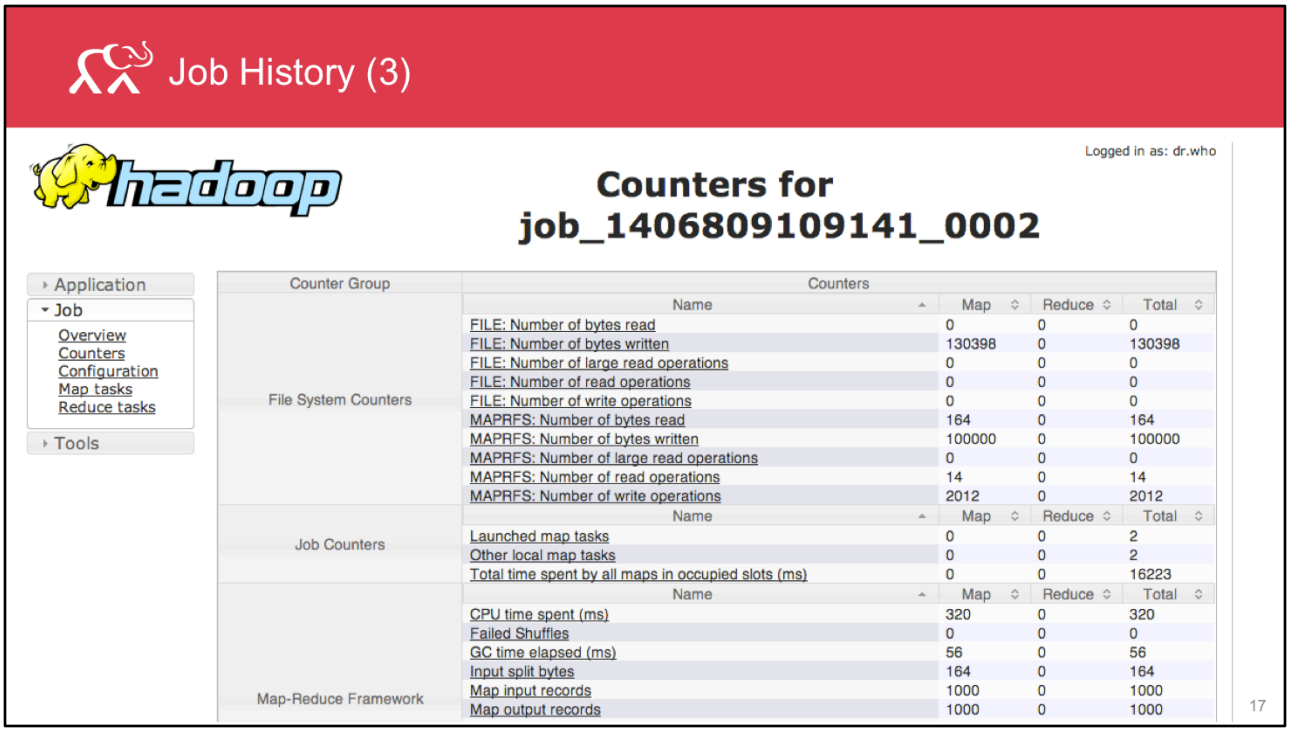


Let's look at the Job History Server in the web UI. This screen shot shows the summary of a launched job in MaprReduce v2 . You can drill down to get more job details.

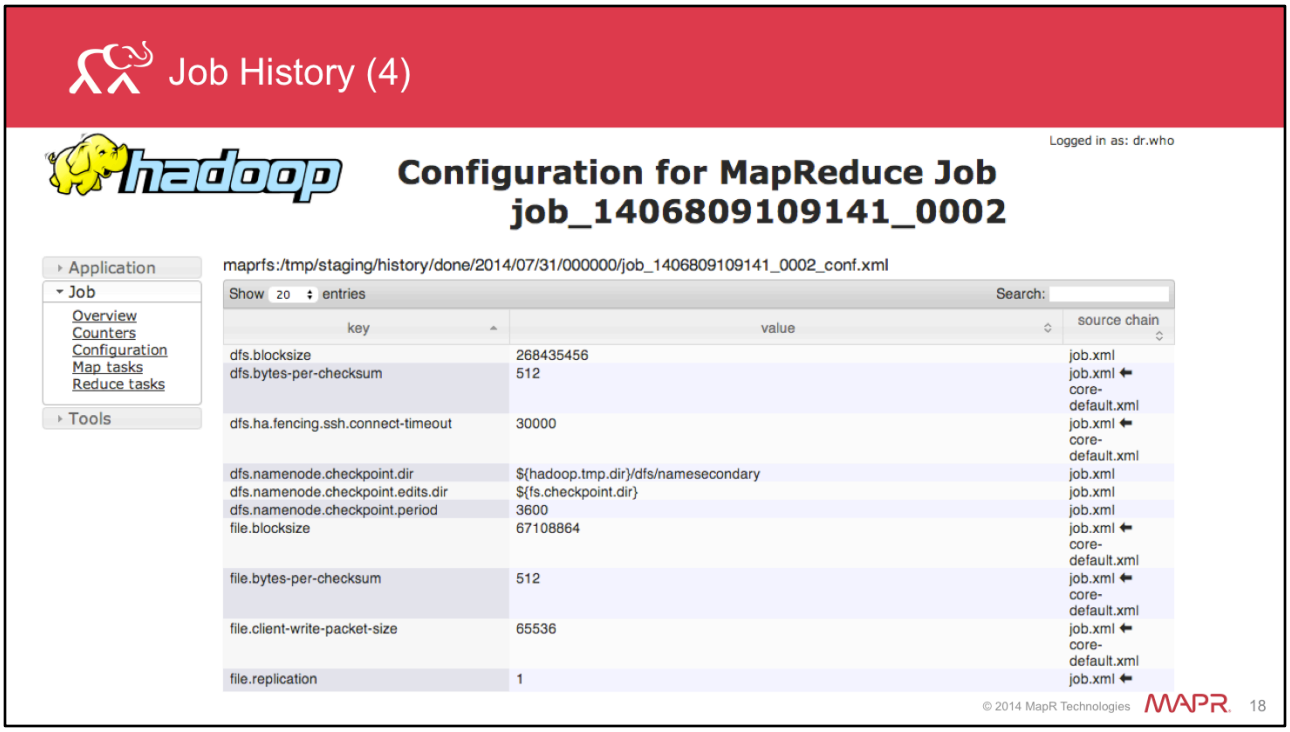
Job History (2)



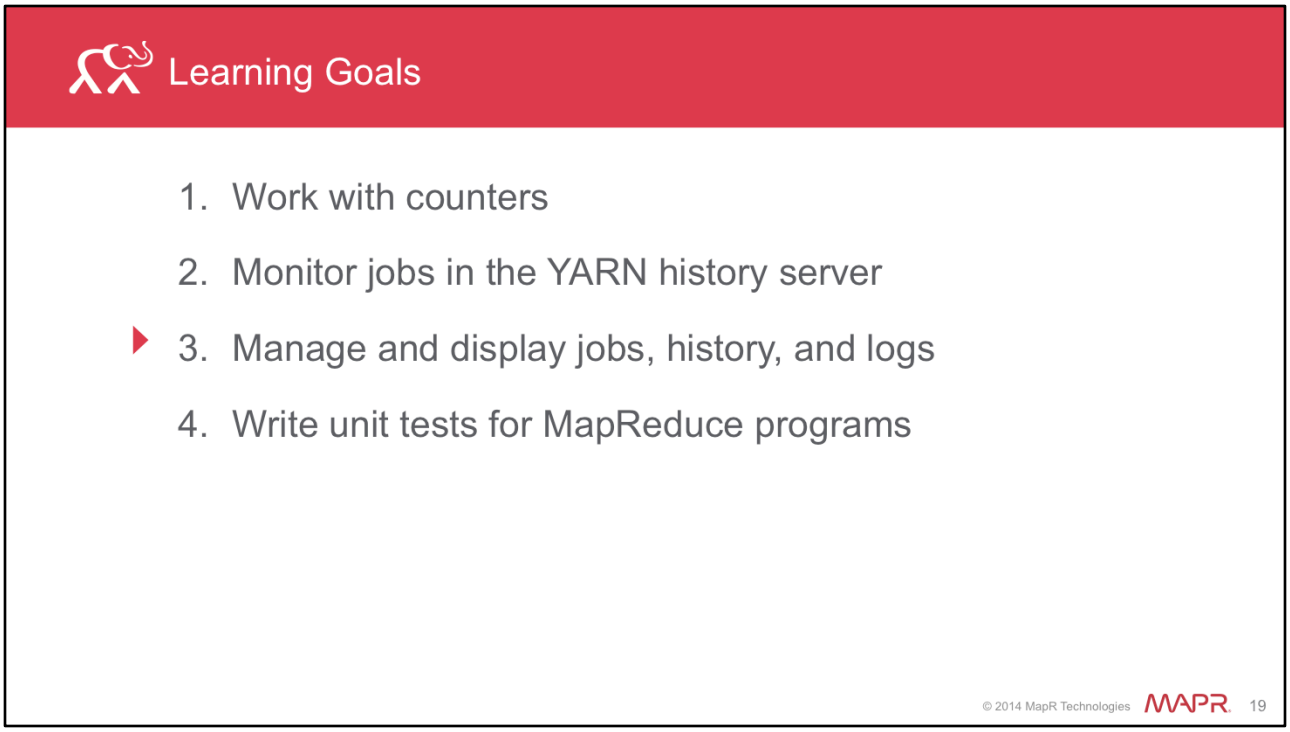
The screen shot above shows the summary of a launched job in MRv2. You can dig into the job to get more details.



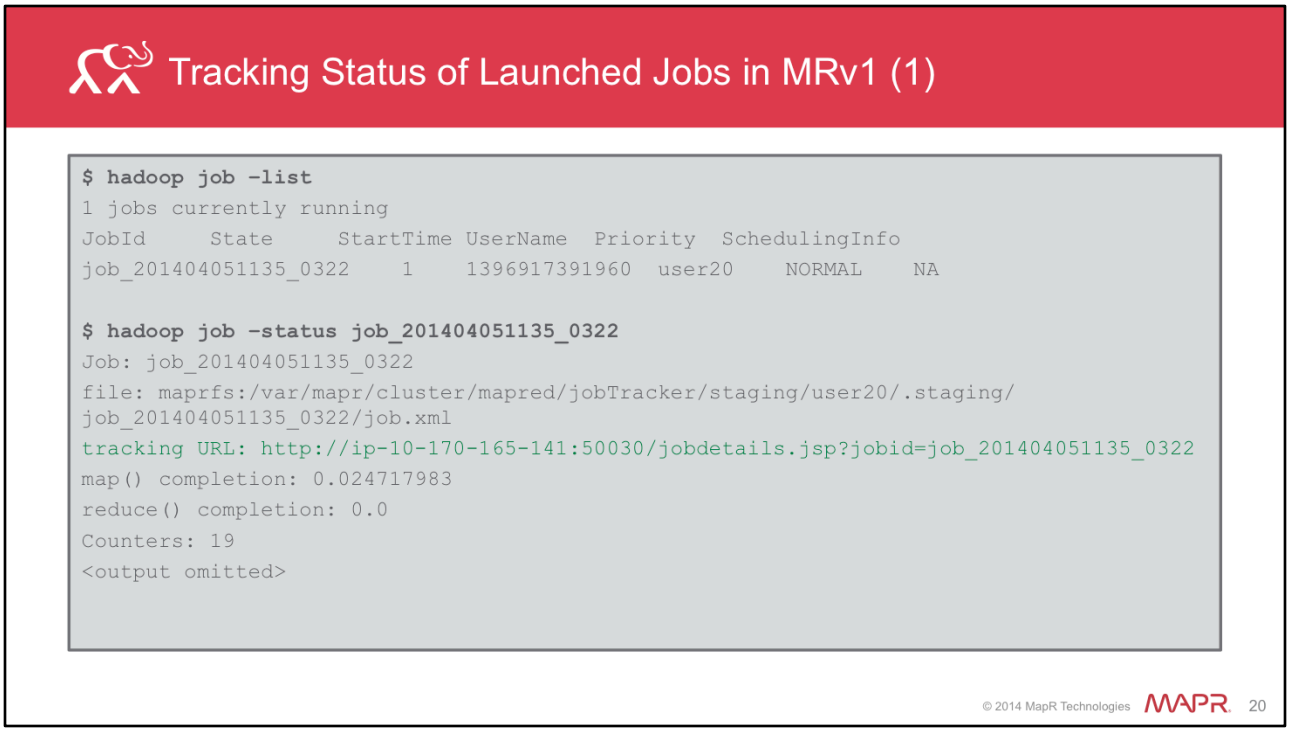
The partial screen shot above depicts the counters for a launched MapReduce job.



The partial screen shot above depicts the configuration parameters that were effective for a particular MapReduce job.

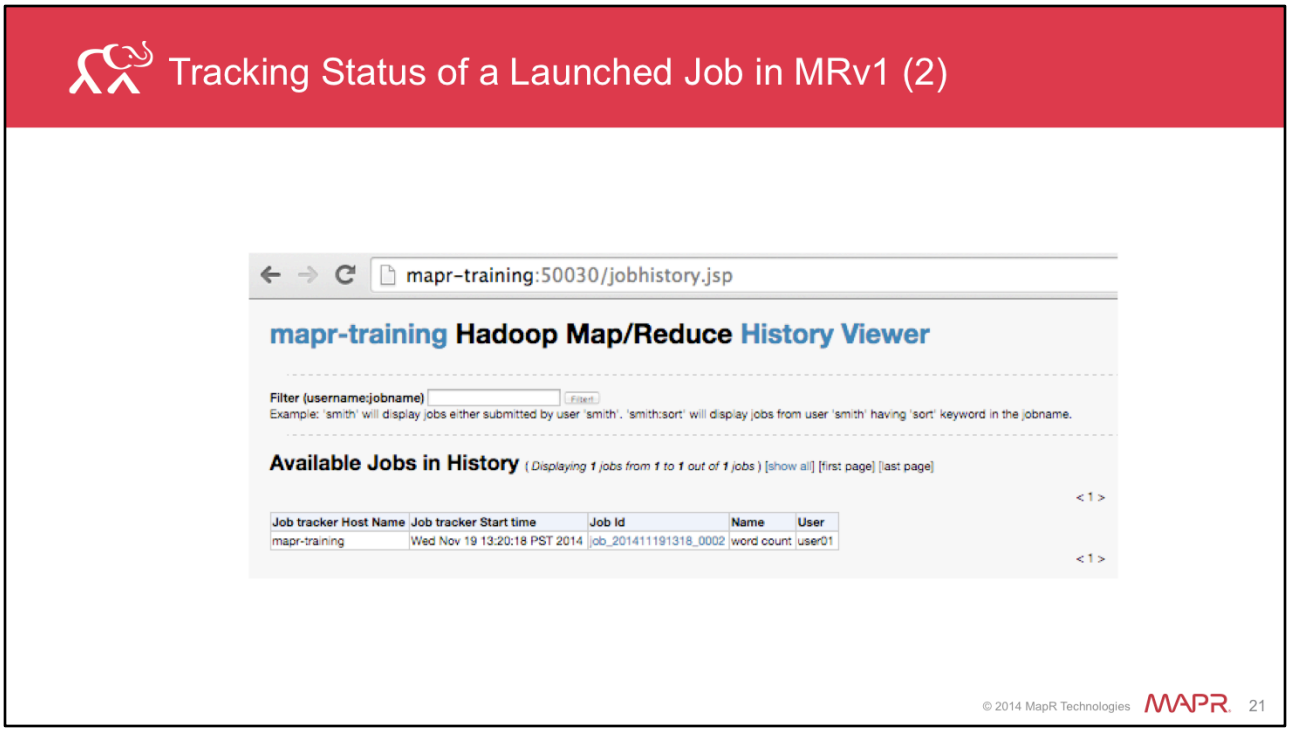


In this section, we discuss how to manage and display jobs, history, and log files.



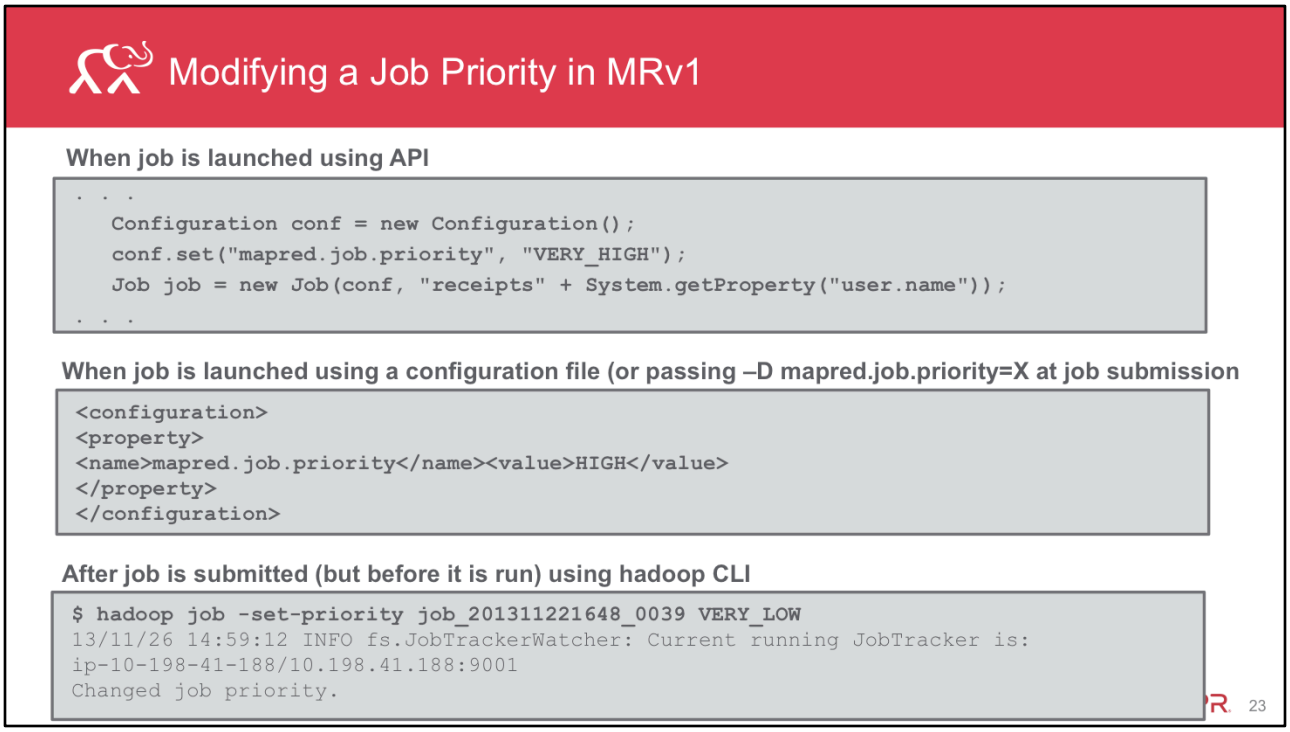
Now let's discuss how to manage and display jobs, history, and logs.

Use the hadoop job command to list and get status of running MapReduce jobs. In this particular case, it appears we have a single job running on the cluster, and it is just over 2% into the map phase. If we want to examine the task tracker Web UI for this task, we could open the URL specified.

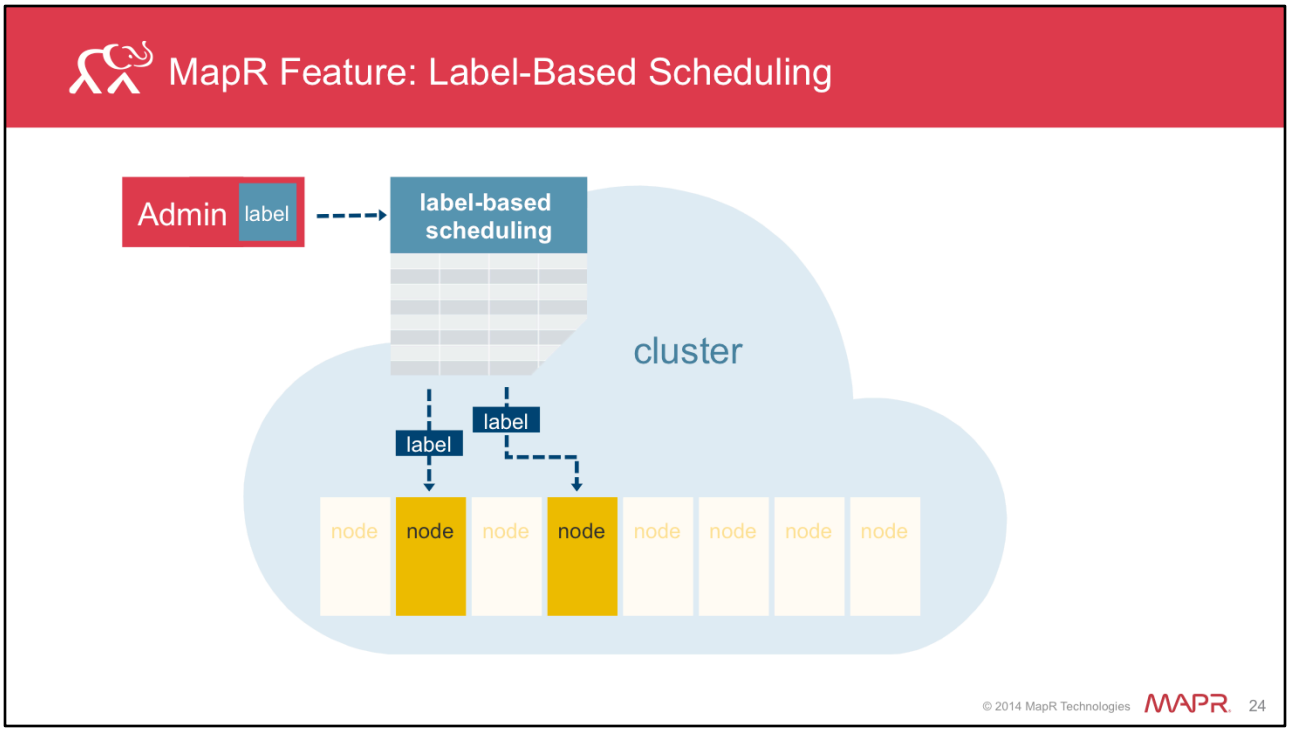


We can also use the Job tracker and Task tracker web Uis to track the status of a launched job, or to check the history of previously run jobs. The job tracker runs a Web service on port 50030 by default, and the task trackers run a Web service on port 50060 by default.

The sequence of commands above shows how to stop a job that is already launched. Do not use the operating system "Kill" to manage job and task processes. Job processes are monitored by the tasktracker, so a killed process will be relaunched on another task tracker.



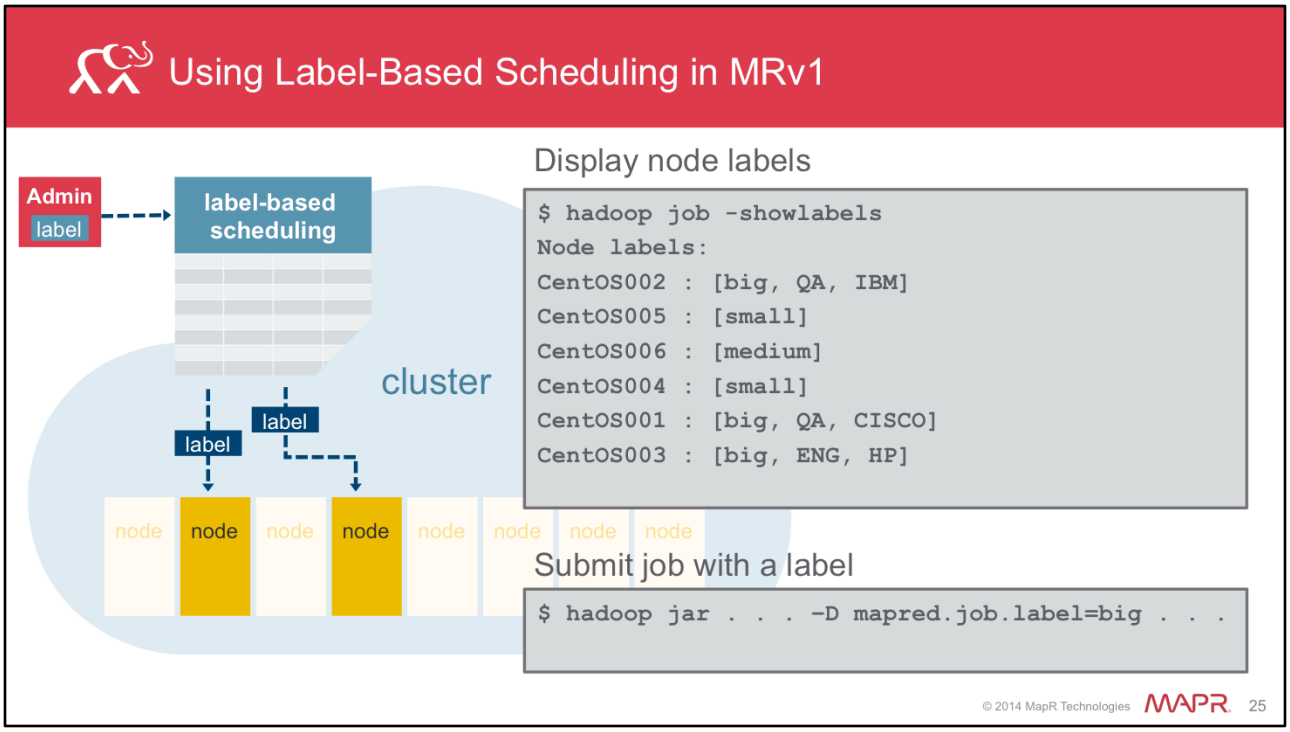
Like most Hadoop parameters, you can modify a job priority by using the API, using a configuration file, and when submitting the job with the hadoop command. You can use the API and the CLI to modify a job priority. Note that the priority is with respect to the other jobs in your pool or queue, depending on the scheduler your cluster is running. You cannot use the job priority to prioritize your job over other jobs in other pools or queues.



MapR has a feature called "label-based scheduling". This feature enables you to override the default scheduling algorithm and launch job tasks on specific nodes. This allows Admins and End Users more control over where jobs run on the cluster.

First, the administrator creates a file that associates labels (arbitrary strings) with one or more cluster nodes. The location of this file is defined by the jobtracker.node.labels.file configuration parameter in the mapred-site.xml file.

Then users submit jobs specifying a label. The scheduler will match the submitted label against the available node labels and run the tasks for the job on the nodes that match the node labels.

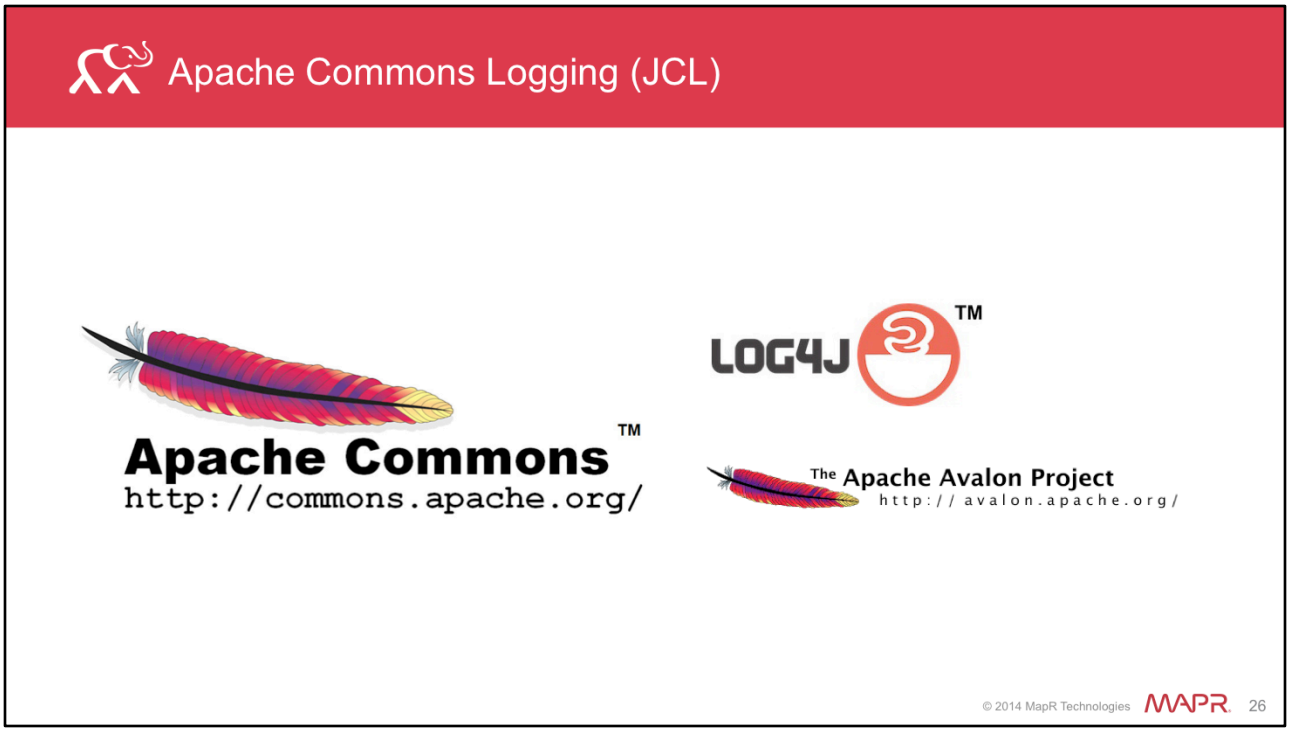


Here is a sample label file that is configured by the MapR cluster administrator.

You can display the labels associated with your MapR cluster by running the 'hadoop job –showlabels' command.

You submit your job to run on a specific set of nodes by using the “-label” option to the hadoop command. In this example, we are requesting that a job be run on specific task trackers that match the "big” label.

Note that if you submit a job with a non-existing label, the job will hang and never get scheduled. This is because a request for a node label is a "hard" requirement, not a soft scheduler hint. Also note that if you don't provide a label, then the default scheduling algorithm for selecting task trackers is used.

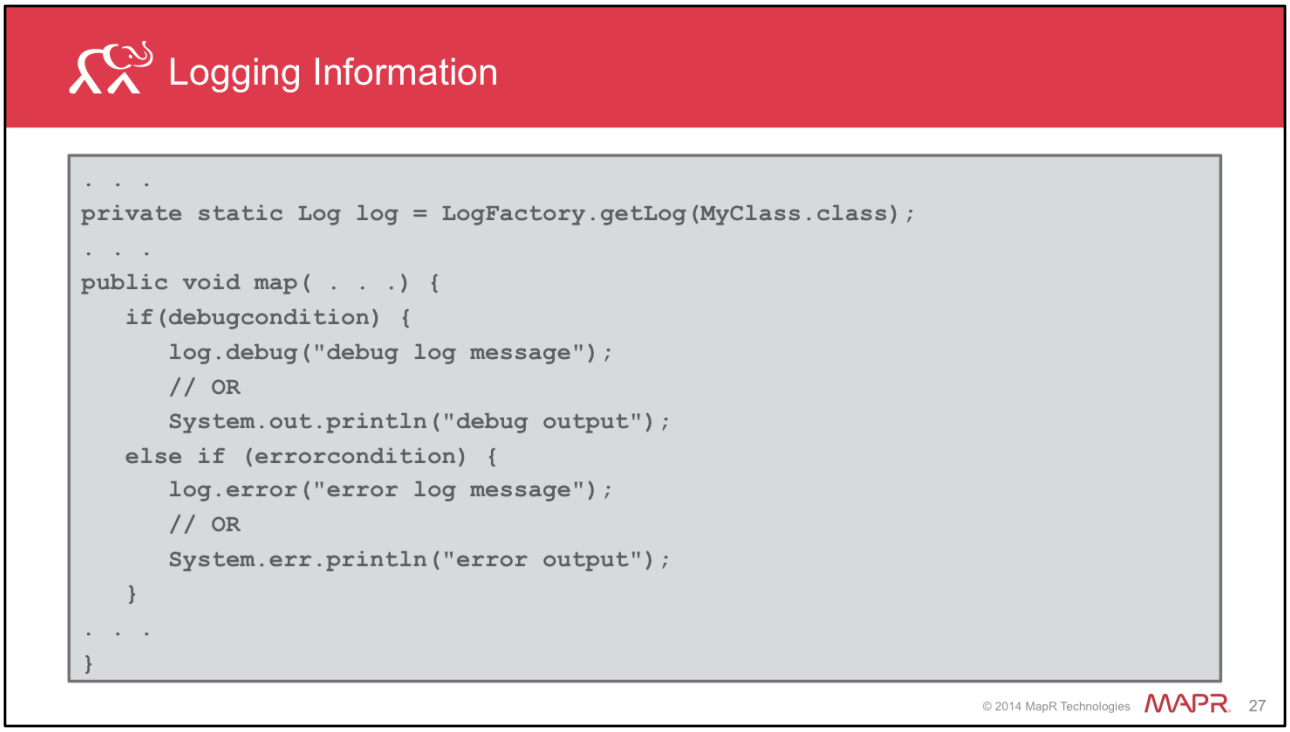


The Apache Commons Logging (JCL) framework is a lightweight, pluggable logging interface for Apache programs written in java. It provides both a robust, full-fledged logging subsystem (commons-logging.jar) as well as thin implementations for other logging tools such as Log4j and Avalon LogKit.

The definable levels in the implementation are trace, debug, info, warn, error, and fatal (in increasing order of arbitrary severity).

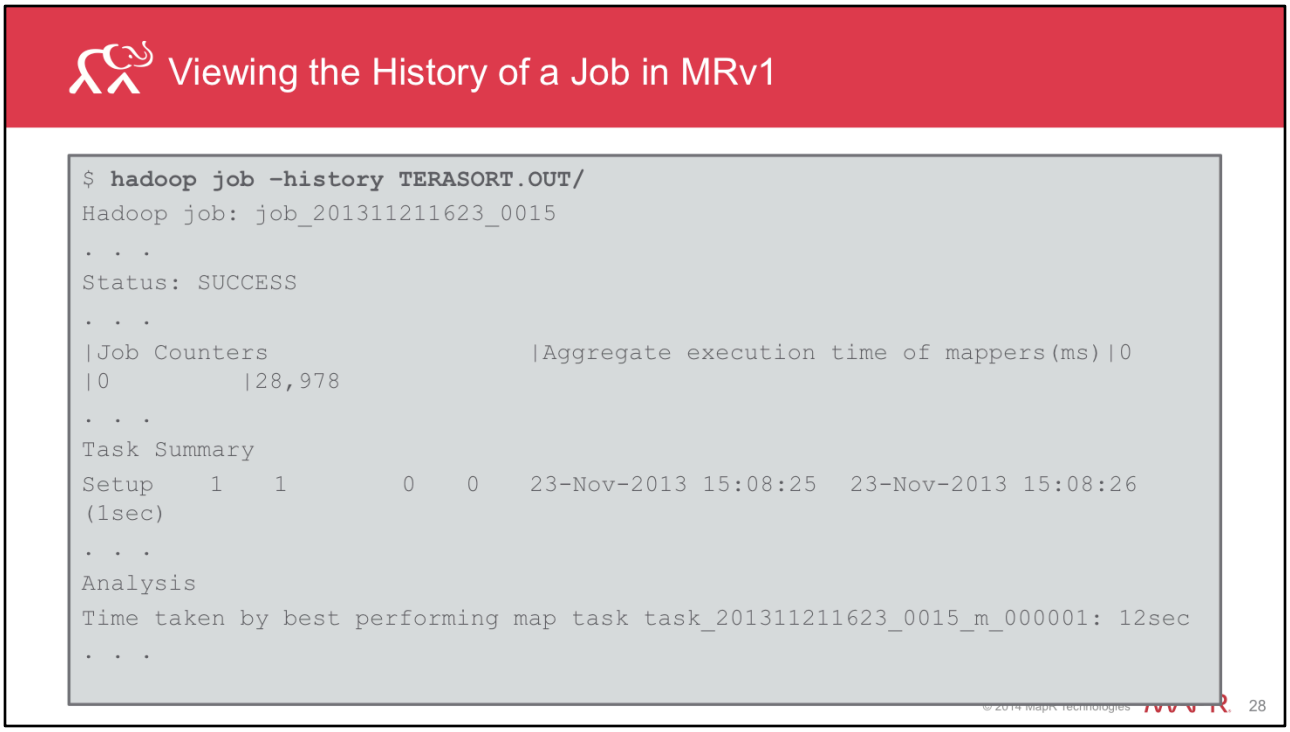
You configure the logging preferences for your Hadoop jobs in the commons-logging.properties file which you place in the CLASSPATH.

See http://commons.apache.org/proper/commons-logging for

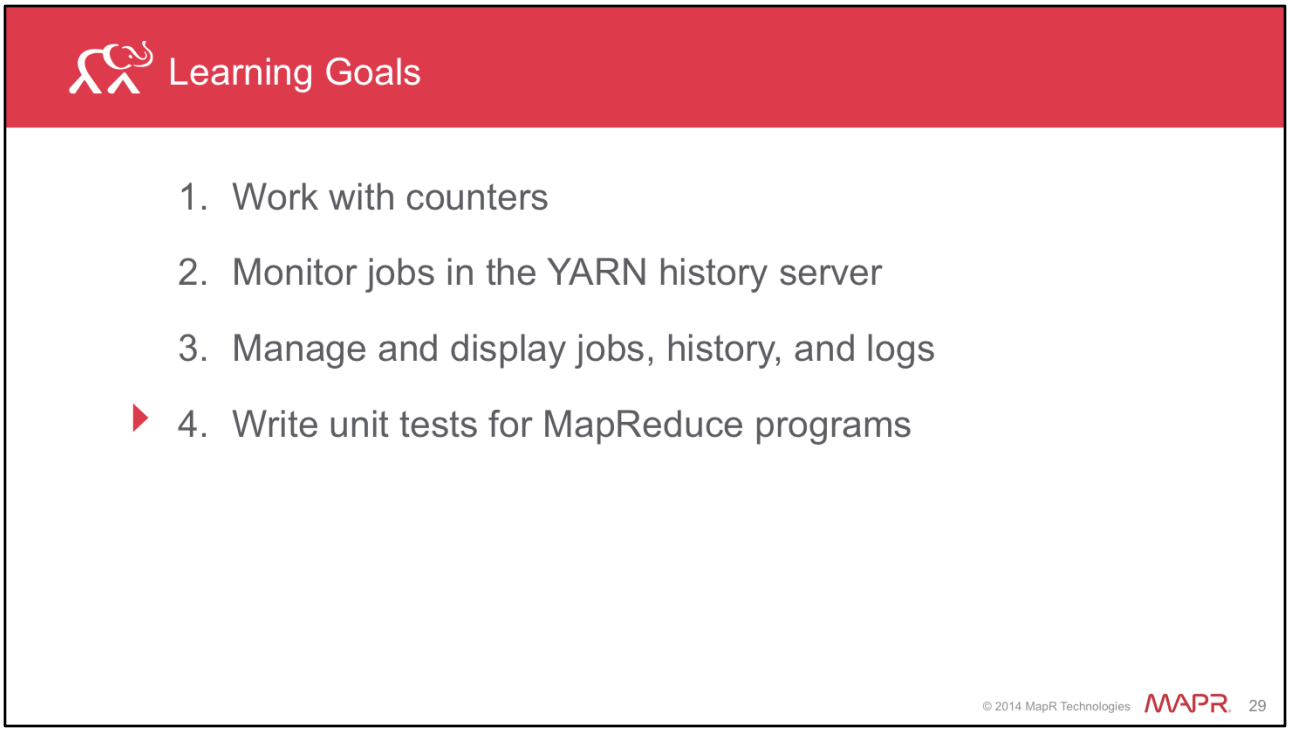


You can write to the configured log system in your map and reduce code. Note that the messages are syslog-style messages. As such, you specify the level of the message with one of the following:

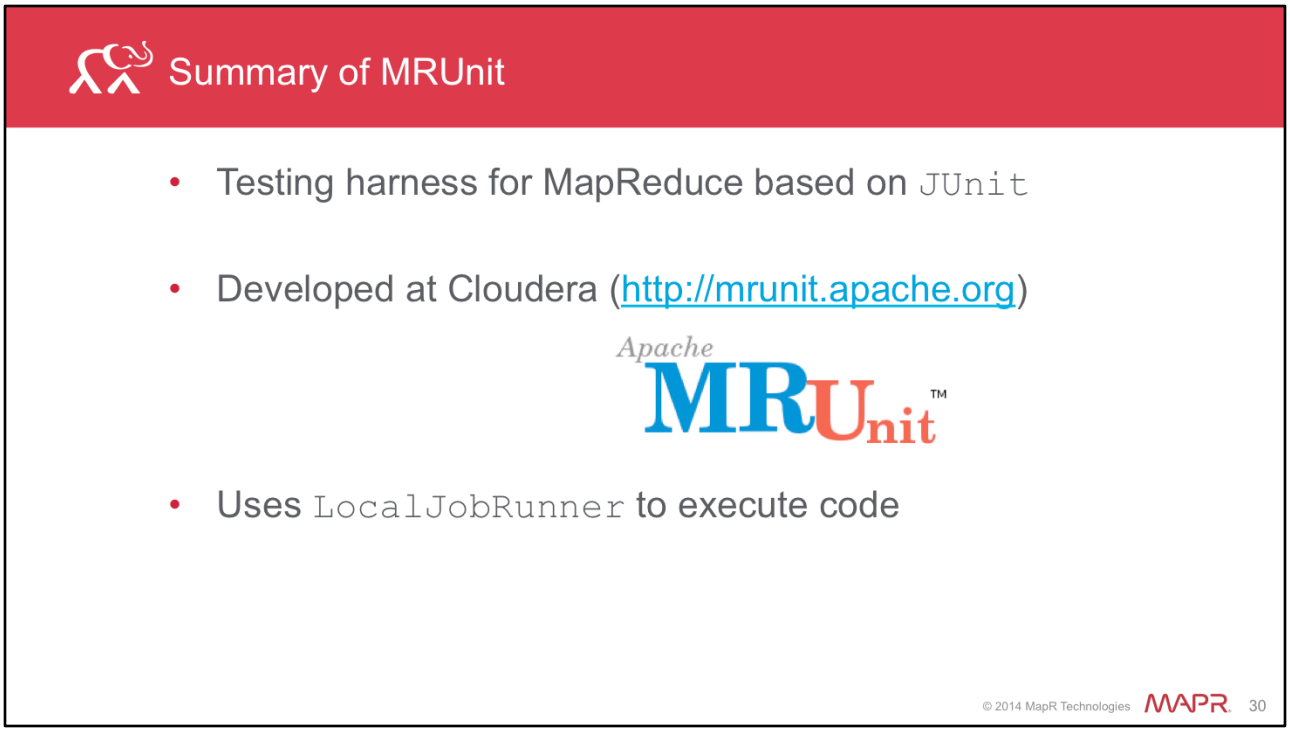
You can configure a log level in your code such that only messages from that level (and higher) are reported to the logging subsystem. Under normal operating circumstances, you should not need more detail than “info” from your jobs. But when you are debugging your code, you should enable “debug” level info from your jobs (mapred.map.child.log.level and mapred.reduce.child.log.level).



The history of a Hadoop job is stored in its output directory. You can display the history using the hadoop command, or you can use the normal operating system commands (e.g. view, cat, more, …) in a MapR environment.



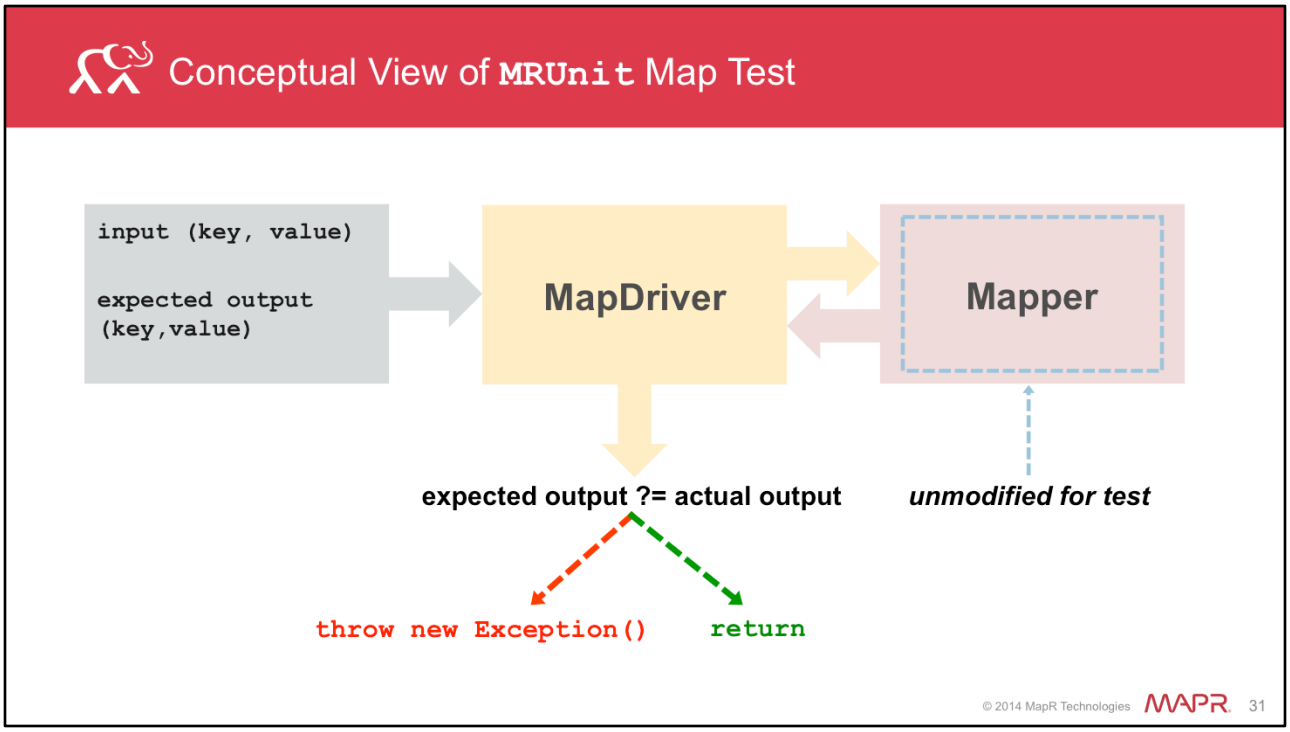
Now let's discuss how to write unit tests for MapReduce programs.



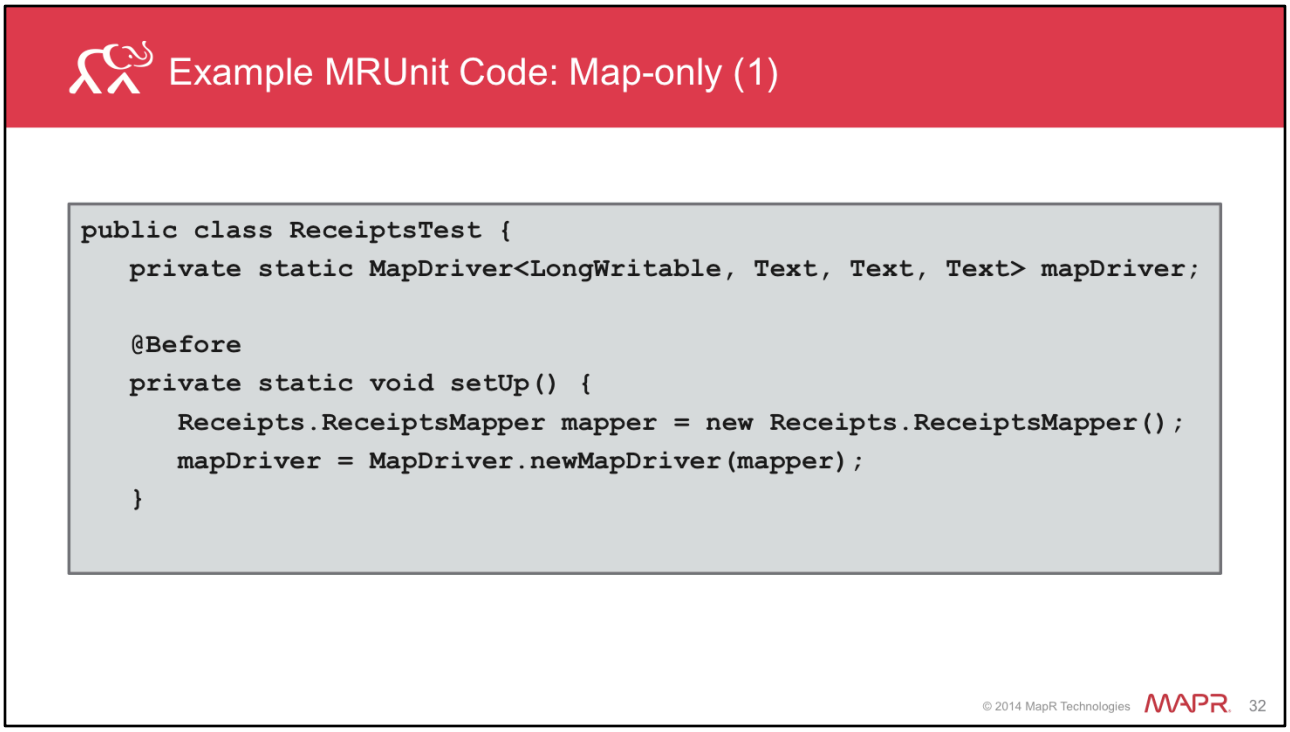
MRUnit allows programmers to test Mapper and Reducer class functionality without running code on a cluster. It is an extension of the Junit framework, and as such will work inside your development environment test harness.

MRUnit was developed at Cloudera but is an official Apache project available at http://mrunit.apache.org.

MRUnit uses the LocalJobRunner to execute either the Mapper or Reducer in isolation, or both, to simulate a single run with a sample data set. You set up your test case by defining one or more inpout records, asserting the output per the input, and executing the code in the LocalJobRunner. Successful code (expected output matches actual output) will silently exit, and failed code (mismatch between expected and actual output) will throw an exception, by default. You can also include other

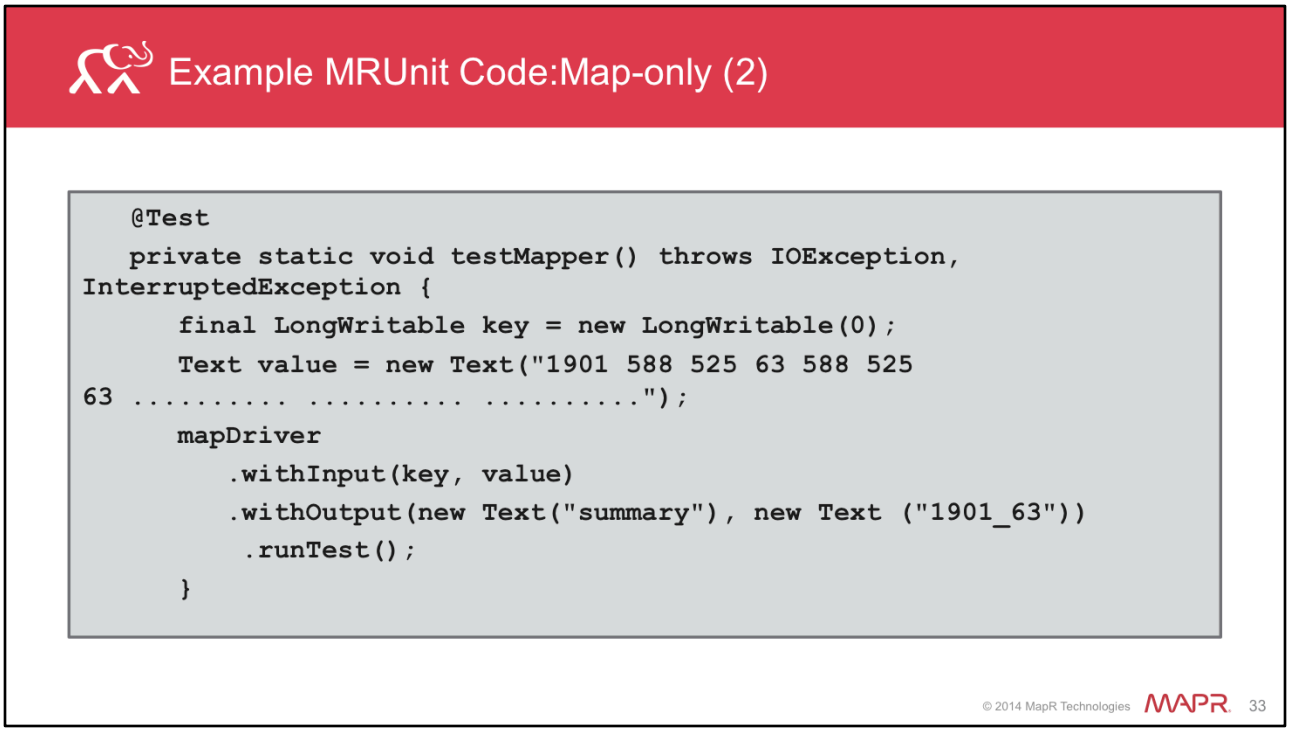


Here is a conceptual view of testing the Mapper class using MRUnit. You provide an input key and value along with the expected output from the map method for that key-value pair. The map driver then does a simple string comparison to compare the actual output with the expected output you provide. If they don't match, the driver throws an exception. If they do match, the driver exits silently.



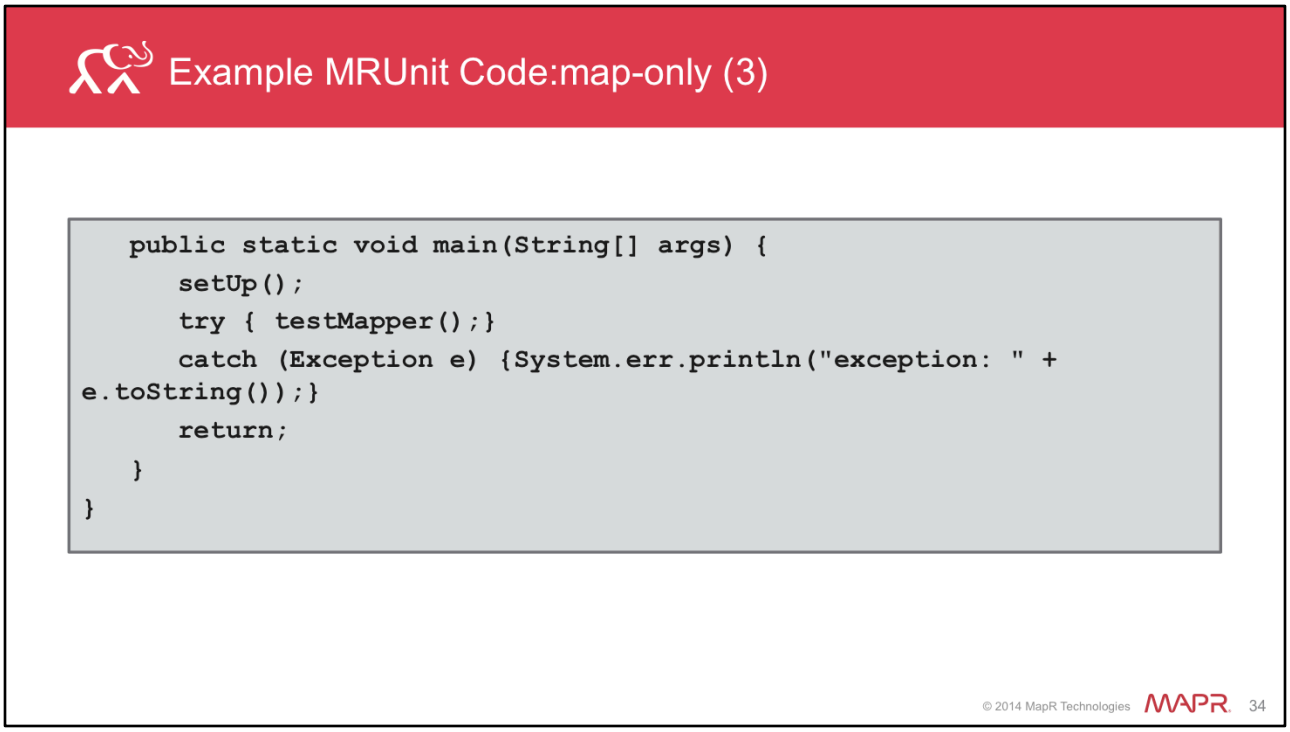
The code snippet above shows a class called "ReceiptsTest" which defines a driver for the MRUnit test harness to test the Mapper only.

The Java annotation "@Before" indicates the method to run before executing the test. In this case, it's the setUp() method which instantiates a new Mapper object that we are testing.

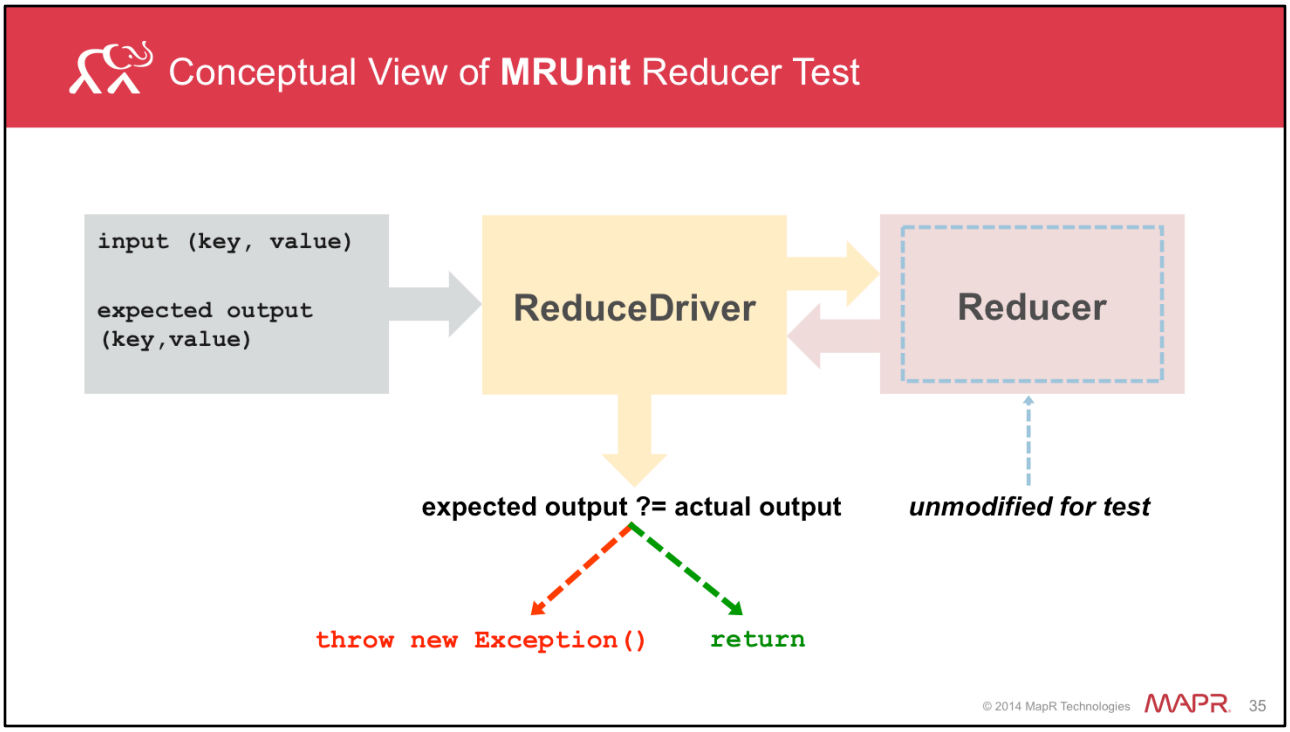


The Java annotation "@Test" indicates the method to execute when testing the code. In this case, the name of the test method is called testMapper(). In this example, we define a single input record which includes a LongWritable(0) – first record – and a sample Text value. The code executes the driver using this record as input and expecting the output to be "summary 1901_63".

The runTest() method calls the MRUnit test harness to execute the job locally. This method will throw an exception if the test fails (for any reason).



The code above shows the main method of the driver calling the 2 aforementioned methods to set up and execute the test. The code catches any exceptions thrown by the runTest() method and prints the trace to standard error. If there are no errors or exceptions, then the code exits silently. Properly equipped development environments will execute this test on build.



Here is a a conceptual view of testing the Reducer class using MRUnit. You provide an input key and list of values along with the expected output from the reduce method for that key-value pair. The reduce driver then does a simple string comparison to compare the actual output with the expected output you provide. If they don't match, the driver throws an exception. If they do match, the driver exits silently.