Developing Hadoop Applications

Overview:

Developing Hadoop Applications ESS 300 – MapReduce Essentials
Build Hadoop MapReduce Applications
Manage and Test Hadoop MapReduce Applications
Launch Jobs and Advanced Hadoop MapReduce Applications

A few notes on the labs in this course:

- Most of the labs will be performed as user `hdoop`.
- You'll want to complete all steps of the labs, because later labs will often build on earlier ones

Welcome – MapReduce Essentials

Lesson 1: Introduction to MapReduce.

Learning Goals

1.1 Describe a Brief History of MapReduce

1.2 Discuss How MapReduce Works at a High-level

1.3 Define How Data Flows in MapReduce

By the end of this lesson, you will be able to:

- Describe a brief history of MapReduce
- Discuss how MapReduce works at a high-level
- And define how data flows in MapReduce.

Learning Goals

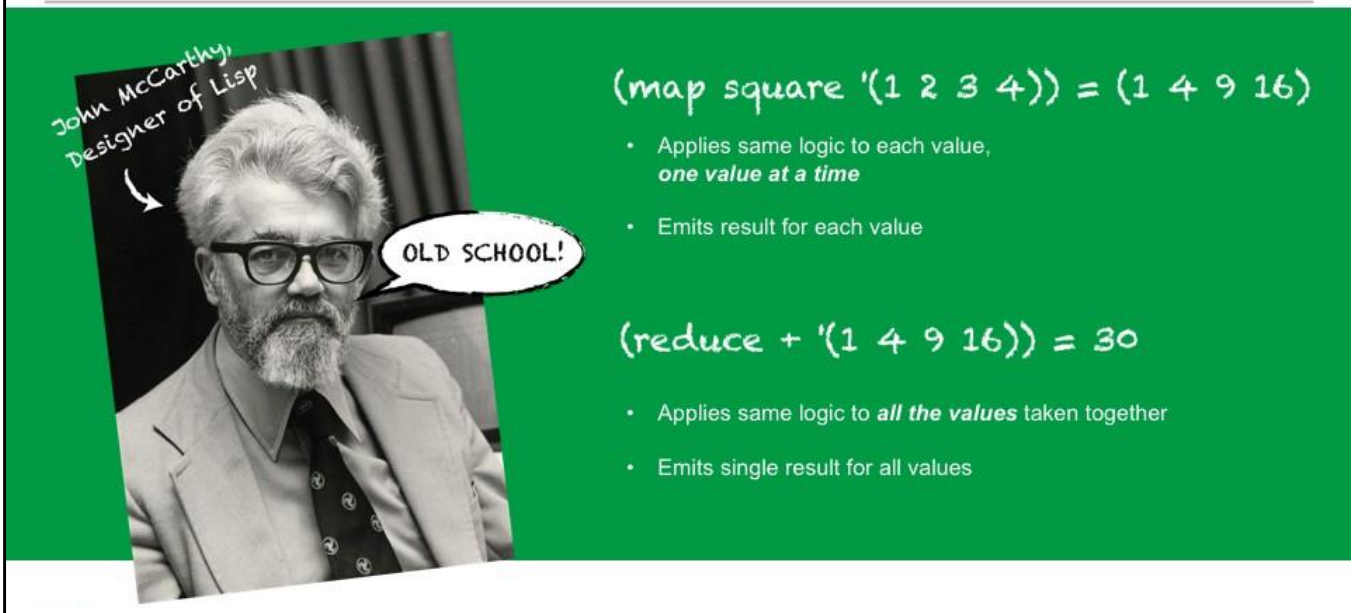
1.1 Describe a Brief History of MapReduce

1.2 Discuss How MapReduce Works at a High-level

1.3 Define How Data Flows in MapReduce

First, let's look at some of the history of MapReduce.

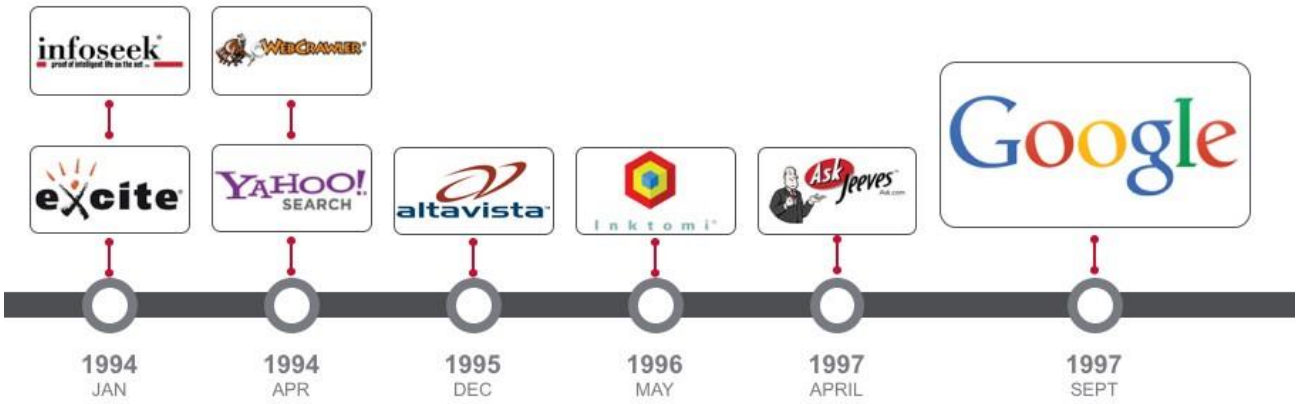
Lisp Map-Reduce



It should be noted that MapReduce was not invented at Google (or Hadoop). Lisp, for example, has used a map and reduce set of functionality since the 1970's.

Here, we see how Lisp uses the MapReduce model. This example has a map of the square function on an input list from 1 to 4. The square function, since it is mapped, will apply to each of the inputs and produce a single output per input. In this case 1, 4, 9, and 16. The addition function reduces the list and produces a single output which is the sum of the input.

Web Search Engines



So how did we get here? Google is the poster child for the power of the MapReduce paradigm which underlies Hadoop. Google was the 19th search engine to enter an already crowded market. You might recall some of these old search brands, but you might not… because within a few short years, Google emerged and dominated the search market.

Simplified Web Search Engine Approach

This is a high-level approach used to describe how a search engine functions. Without getting into the details of each step, it is straightforward to understand conceptually.

First, it crawls the web. This involves using so-called "spiders" to crawl the web, following links within web pages to get to other web pages. Overall, this is the most time-consuming step.

Second, it sorts the pages by URL.

Next, it removes any junk. A good search engine should remove "junk" from a known list of bad sites, or contextually based on the what is inside a given page.

Then it ranks all provided results by sorting the URLs for a given word or set of words based on frequency, number of hits, freshness of page… etc.

And finally, it creates an index to display to the user. (For each word, create a list of URLs that contain that word.)

Word Count Algorithm from Google White Paper

```
map(String key, String value):
   // key: document name
   // value: document contents
   for each word w in value:
      EmitIntermediate(w,"1");

reduce(String key, Iterator values):
   // key: a word
   // values: a list of counts
   int result = 0;
   for each v in values:
      result +=ParseInt(v);
      Emit(AsString(result));
```

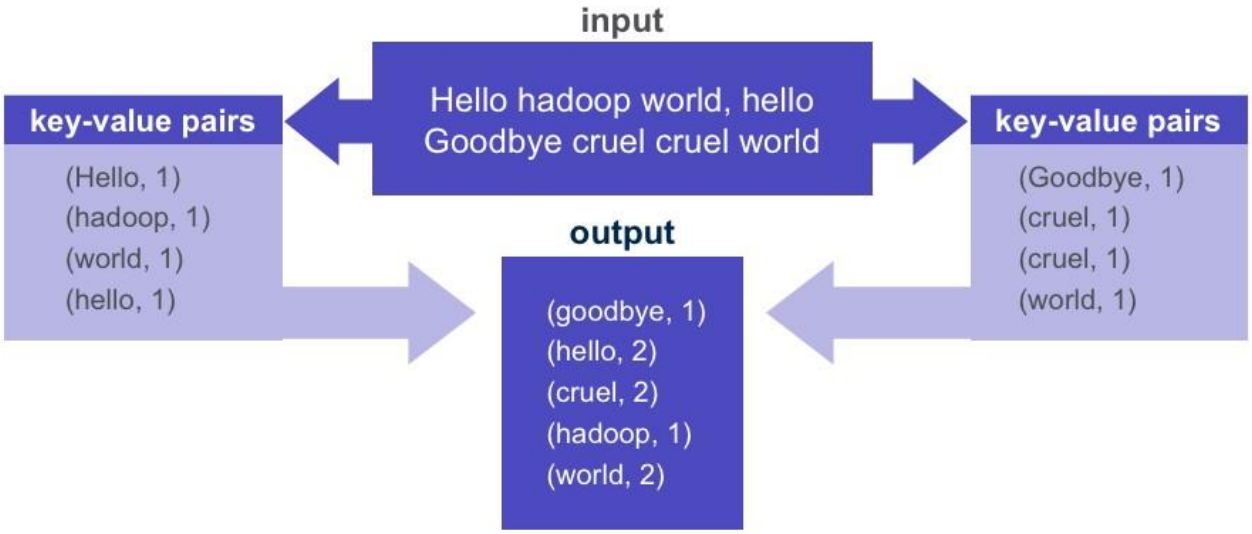
MapReduce: Simplified Data Processing on Large Clusters Jeffrey Dean (jeff@google.com) & Sanjay Ghemawat (sanjay@google.com) , Google,Inc.

The word count algorithm shown here is taken directly from the seminal paper on MapReduce from Dean and Ghemawat. Algorithms for counting words existed before this paper was published, but the mechanism for doing it using the MapReduce framework was novel. The algorithm is quite simple, which is true of most MapReduce programs as we will later see.

The map method takes as input a key and a value, where the key represents the name of a document and the value is the contents of the document. The map method loops through each word in the document and emits a 2-tuple representing word, and 1 in this example.

The reduce method takes as input a key and a list of values, where the key represents a word. The list of values is the list of counts for that word. In this example, the value is a list of 1's. The reduce method loops through the counts and sums them. When the loop is done, the reduce method emits the final count as a string.

Word Count Example



The input for our trivial word count example is shown above.

Each mapper in the map phase takes an input list (e.g. “Hello hadoop world, hello”) and maps it to a list of key-value pairs (e.g. (Hello, 1), (hadoop, 1), (world, 1), (hello, 1)). Note that even in the input list “Hello hadoop world, hello,” there is a key and a value, though the key is not obvious. We will discuss this later.

The reducer in the reduce phase takes an input of a key and the list of values associated with that key (e.g. (world, 1), (world, 1)) and emits a single output (e.g. (world, 2)). Note, the standard word count of MapReduce is not case sensitive.

Learning Goals

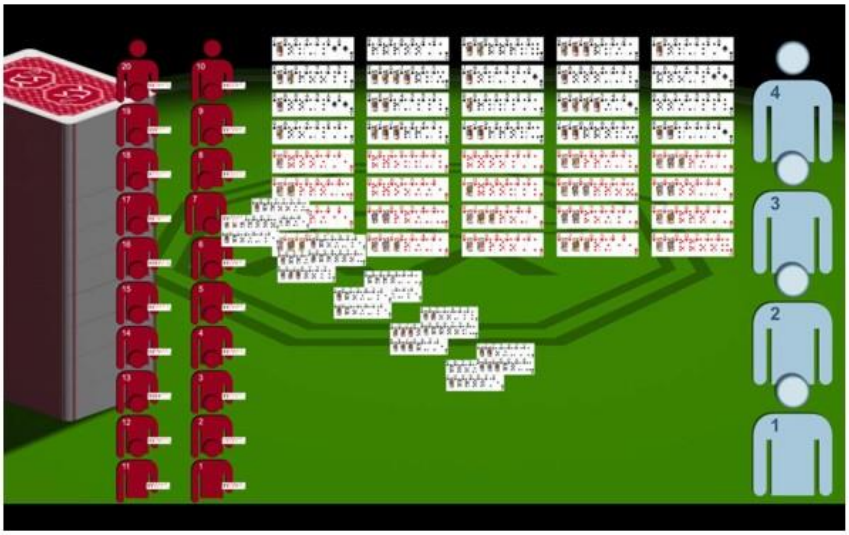
1.1 Describe a Brief History of MapReduce

1.2 Discuss How MapReduce Works at a High-level

1.3 Define How Data Flows in MapReduce

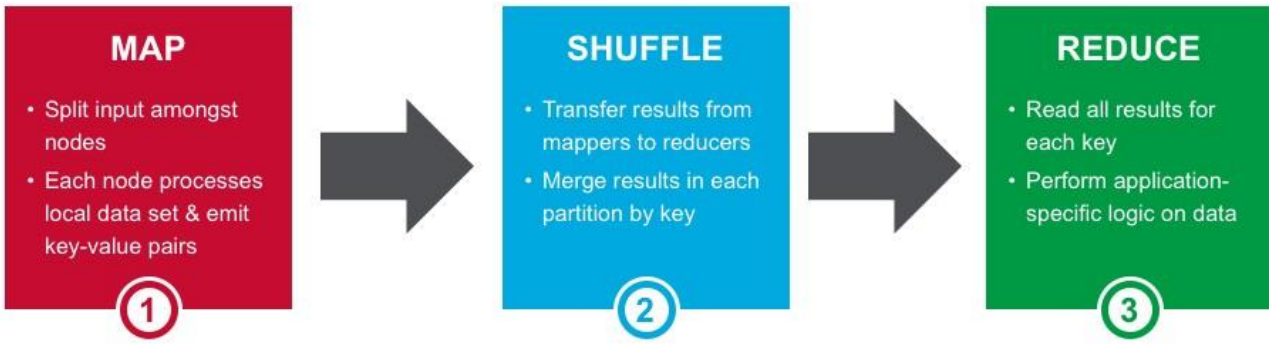
In this section, we discuss how MapReduce works at a high-level and understand the model conceptually.

Cards Game Animation



https://www.youtube.com/watch?v=A_8d55ZfWo8

Describe a Summary of MapReduce

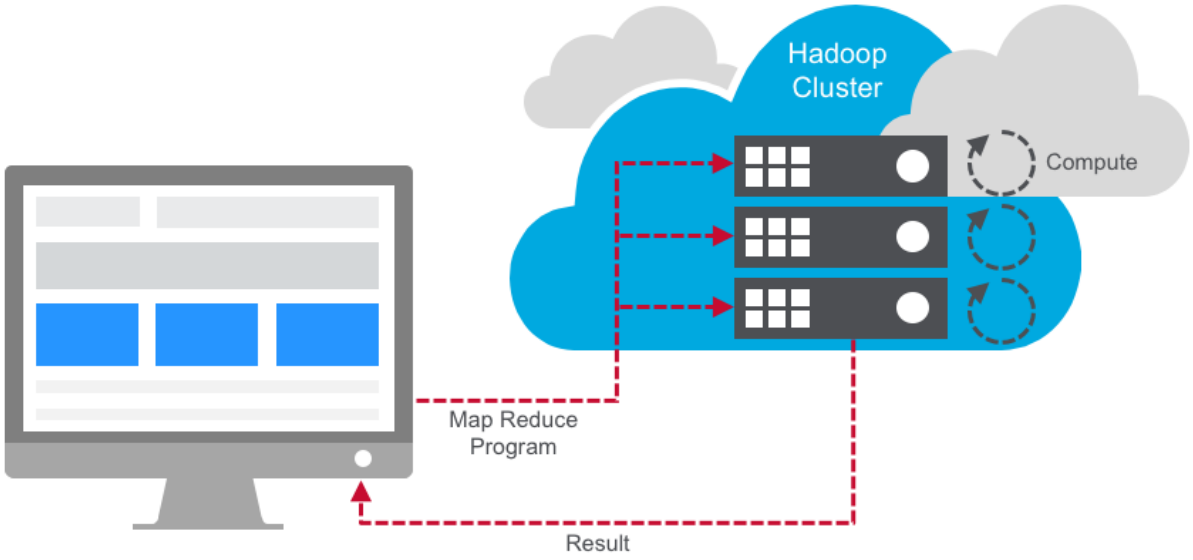


Let's discuss a high-level summary of the MapReduce computational model. As shown here, there are three phases in MapReduce: map, shuffle, and reduce.

The data in the map phase is split amongst the Task Tracker nodes where the data is located. Each node in the map phase emits key-value pairs based on input one record at a time. The shuffle phase is handled by the Hadoop framework, transferring and merging the result of the mapper outputs, to the reducers as partitions.

The data in the reduce phase is divided into the partitions in which each reducer reads a key and iterable list of values associated with that key. Reducers emit zero or more key-value pairs based on the application logic.

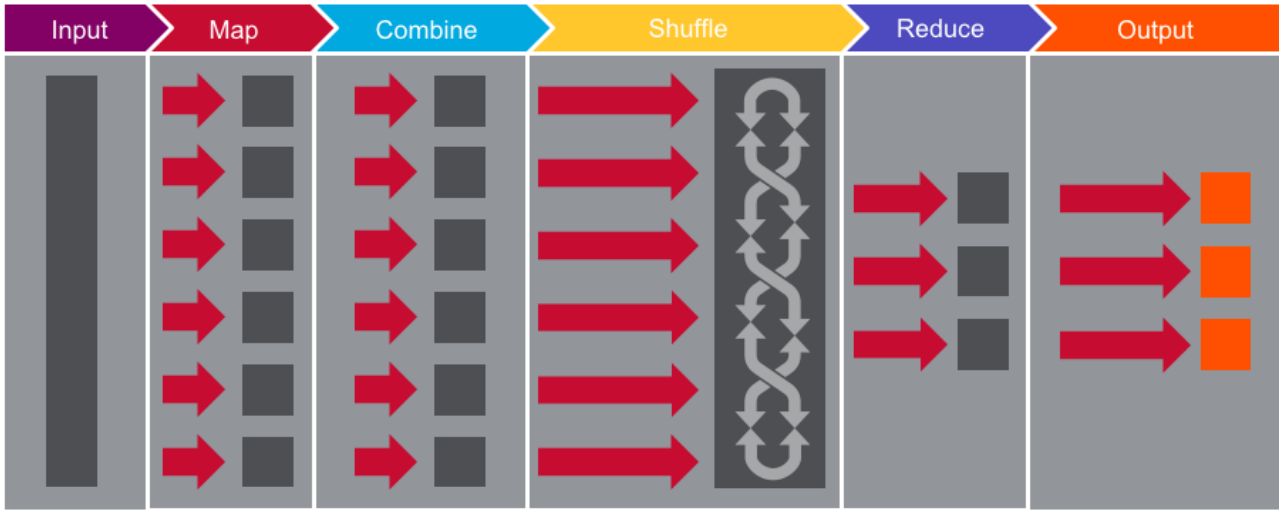
Describe the Hadoop Runtime Model



Recall that the MapReduce model is based on sending compute to where data resides.

We collect source data on the Hadoop cluster, either by bulk copying data in, or better yet, by simply accumulating data in the cluster over time. When we kick off a MapReduce job, Hadoop sends map and reduce tasks to appropriate servers in the cluster, and the framework manages all the details of data passing between nodes. Much of the compute happens on nodes with data on local disks, which minimizes network traffic. Once completed, we can read back the result from the cluster.

MapReduce Flow



Let's step through the flow of a MapReduce job.

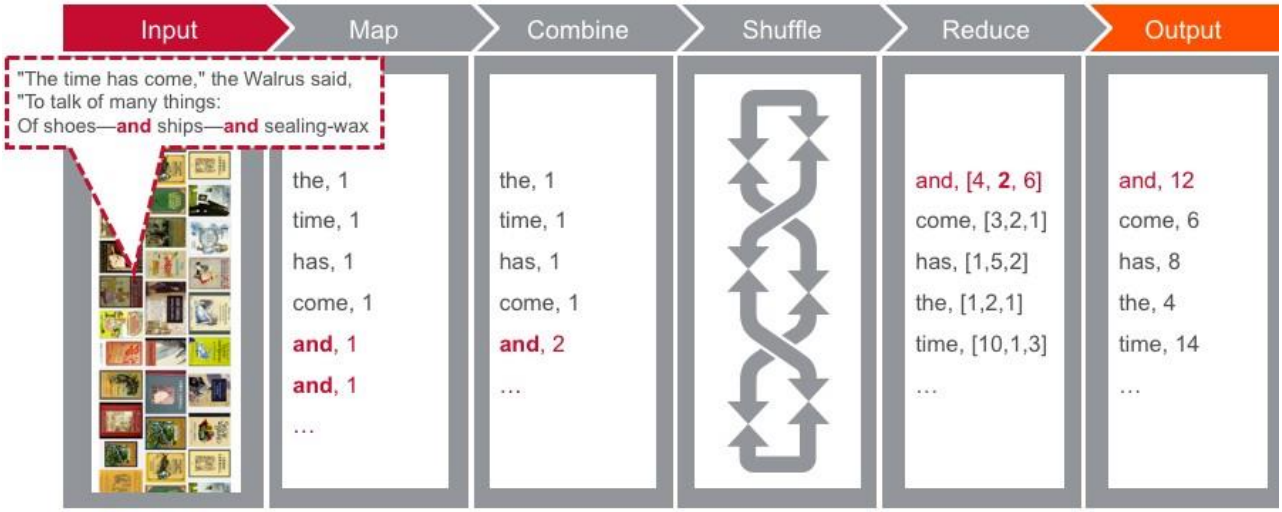
We start off with our input, which could be one or many files.

- The framework logically breaks up the input into "splits." Each split contains many records and records can be any type of information, like text, audio data, structured records, etc. Each split typically corresponds to a block of data on a node where the data resides, but the programmer doesn't need to be aware of this.
- Each split is processed by a map task. Each record in a split is passed, independently of other records, to the map() method. The map() method receives each record as a key-value pair, although in some cases it might only be interested in either the keys or the values. The Mapper emits key-value pairs in response to the input record. It can emit zero records, or it can emit lot of records.
- The map output is partitioned such that all records of a particular key go to the same reduce task. This phase involves a lot of copying and coordination between nodes in the cluster, but again, the programmer doesn't need to worry about these details.
- If the reduce method is like addition, where summing subtotals of terms is equivalent to summing all individual terms, it's more efficient to split the reduce step and do some of it before shuffle. In this case, a combiner method is used, which is often the same method as the reducer. This cuts down the number of records that need to be copied from node to node.
- Whether or not a combiner is invoked, the framework will send the intermediate results from the
  mappers to the reducers. The reduce() method receives a single key and a list of all values associated with that key. The reducer, based on your logic, will emit 0 or more key-value pairs which constitute the final results of this MapReduce job.
- Finally, the framework collects the reducer outputs so you can access the results.

Most of the parts here are handled by the framework. We only have to provide the code in the map and reduce columns.

Note that, at this level of detail, the boxes don't represent nodes in the cluster. We're talking only about the logical flow of data. The framework issues map tasks and reduce tasks in parallel. For example, several map tasks might run concurrently on a single node.

MapReduce Example: WordCount



Let's step through an example word-count program using MapReduce to count the occurrences of each word in a set of input text files.

WordCount is the "Hello World" program for MapReduce, though counting strings may be of real use in your programs.

As input, let's say we have the complete collection of Lewis Carroll's books, and we want to count the occurrence of every word.

Hadoop divides this into "input splits."

One of the nodes contains Tweedledee's poem, "The time has come to talk of many things" and so forth. In the case of text input, every line of text is a record and each record gets fed individually to the map method. The key is the byte-offset into the file, and the value is the text. In this case, the key isn't useful for this program, so we'll ignore it.

The map method tokenizes the input string, and outputs a key-value pair for every word. The key is the word, and the value is simply 1. As you can see, the word "and" shows up twice, and it produces two distinct key-value pairs.

In this case, some combining occurs before shuffle-and-sort. The combiner aggregates multiple instances of the same key coming out of the mapper into a subtotal. So in our case, the two instances of the string "and" get combined, and only one record of a value 2 goes through the shuffle.

The framework sorts records by key and, for each key, sends all records to a particular reducer. This particular reducer gets the keys: **and**, **come**, **has**, **the** & **time**. You can see that the combined value 2 for "and" shows up here. The other values come from other map tasks.

Then begins the reduce phase, which just has to sum up the values from the Mapper.

Finally, the framework gathers the output and deposits it into the file system where we can read it. (Of course, if we really counted all words in a book, there would be a lot more occurrences of these words than shown here.)

So, there we have it: our first example of a MapReduce program.

Knowledge Check

Knowledge Check

Select which are true of a MapReduce job:

1. The data in the map phase is split amongst the task tracker nodes where the data is located
2. The shuffle phase is handled by the Hadoop framework.
3. Output from the mappers is sent to the reducers as partitions.
4. Reducers emit zero or more key-value pairs based on the application logic
5. All of the above

Lab 1.2: Run `wordcount`

Now that we've described the process of how to run wordcount in MapReduce. You may want to try running wordcount on a single text file, a set of text files, or even a binary file.

Learning Goals

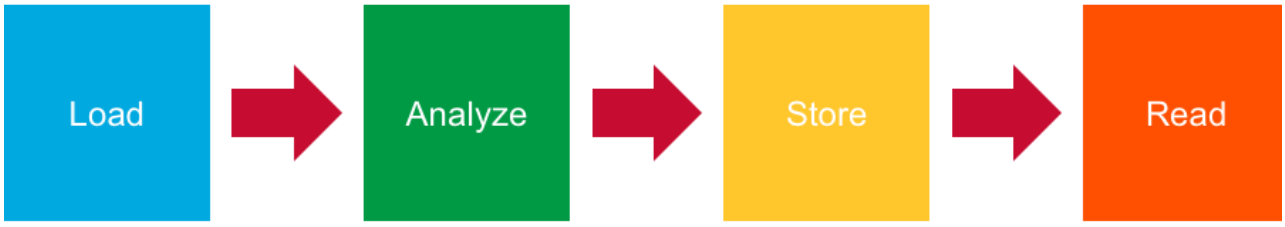
1.1 Describe a Brief History of MapReduce

1.2 Discuss How MapReduce Works at a High-level

1.3 Define How Data Flows in MapReduce

In this section, we illustrate data and execution flows in the MapReduce framework.

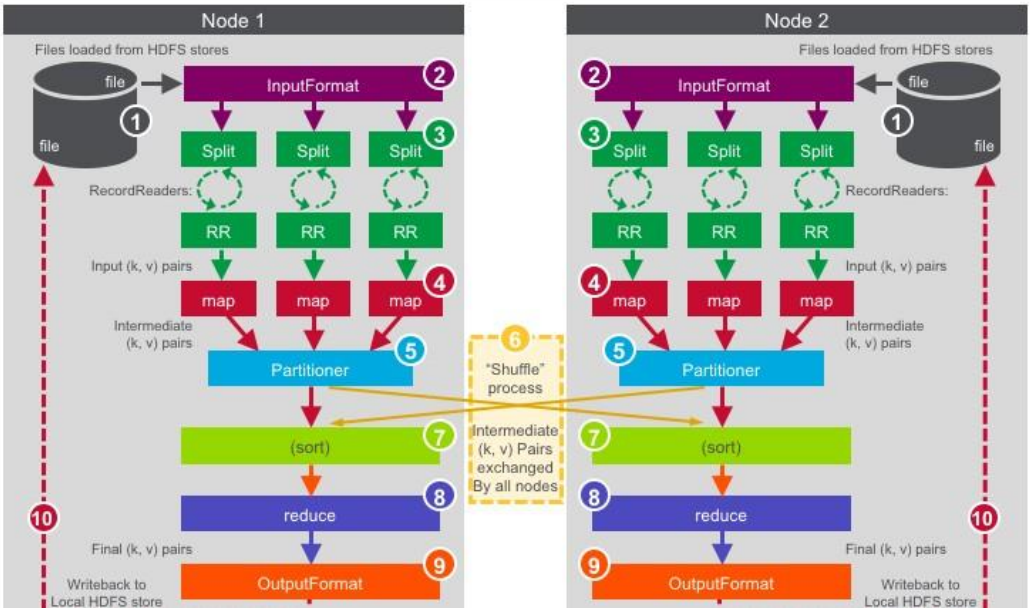
Define a Simplified MapReduce Workflow



Let's look at a simplified MapReduce workflow. First, load the data into the cluster (or you can also use MapR-FS to store the data in production). Then analyze the data with MapReduce and store the results in MapR-FS. You can then perform your business logic analysis on the results that are read in from the cluster.

Note that MapReduce is a simplified programming model and as such, you usually need to write a few MapReduce jobs in order to to completely analyze your data.

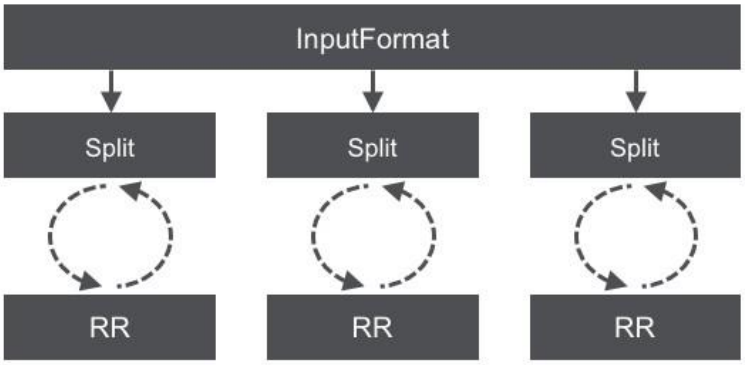
Summary of Execution and Data Flow

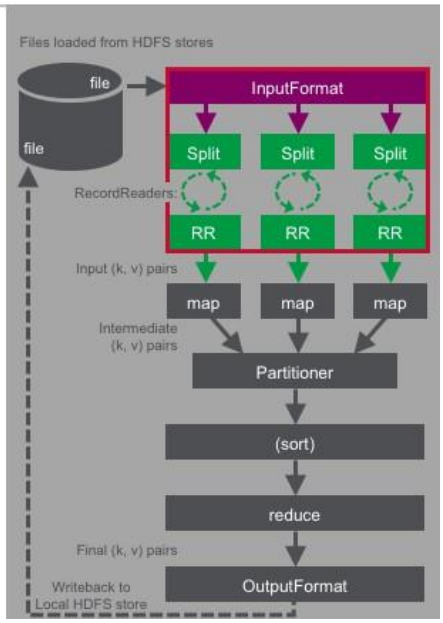


Here is the flow of data in a MapReduce execution:

1. Data is loaded from the Hadoop file system
2. Next the job defines the input format of the data
3. Data is then split between different map() methods running on all the nodes
4. Record readers then parse out the data into key-value pairs that serve as input into the map() methods
5. The map() method produces key-value pairs that are sent to the partitioner
6. When there are multiple reducers, the mapper creates one partition for each reduce task
7. The key-value pairs are then sorted by key in each partition
8. The reduce() method takes the intermediate key-value pairs and reduces them to a final list of key-value pairs
9. The job defines the output format of the data
10. And data is written back to the Hadoop file system

InputFormat Class





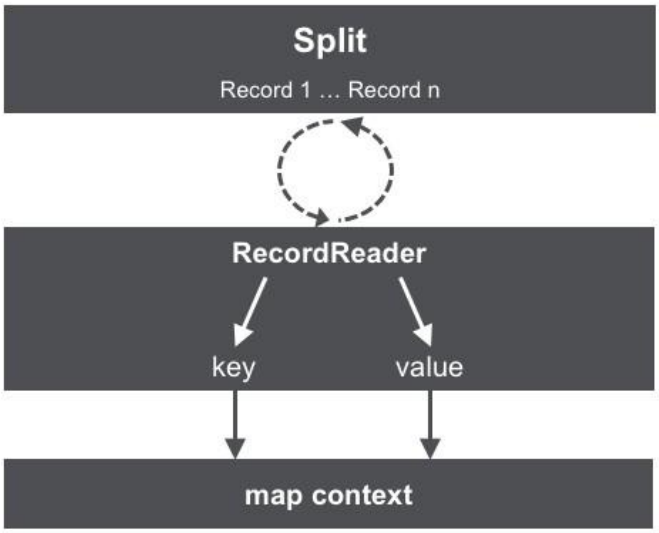
The InputFormat object is responsible for validating the job input, splitting the files amongst the mappers, and instantiating the RecordReader.

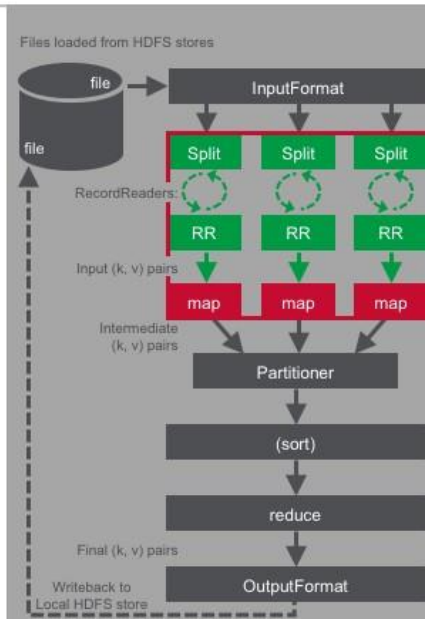
By default, the size of an input split is equal to the size of the block. In Hadoop, the default block size is 64M. In MapR, the equivalent structure is called “chunk” and has default size 256M.

Each InputSplit references a set of records, each of which will be broken up into a key and value for the mapper. The work of splitting up the data is done before your map() process even begins running. The job execution framework will attempt to place the mapping task as close to the data as possible.

Once the task is assigned to a node, that TaskTracker passes the InputSplit to the RecordReader constructor. The RecordReader reads records one by one and passes them to the map() method as key- value pairs. By default, the RecordReader considers the entire line up to the new line as a single record. You can extend the RecordReader and InputFormat classes to define different records in your input file, for example, multi-line records. When the InputSplit has no more records, the RecordReader stops.

Mapper Class



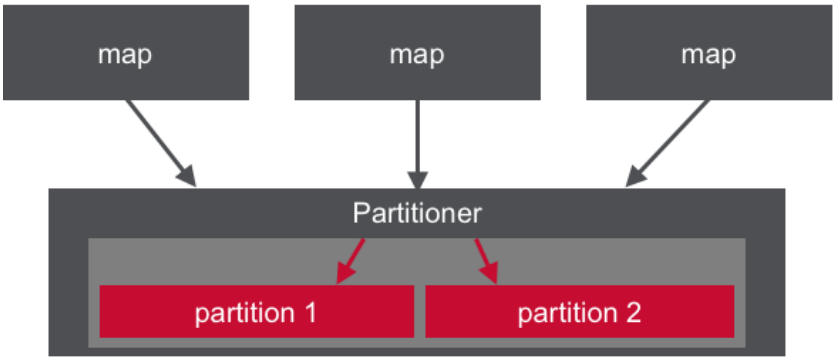


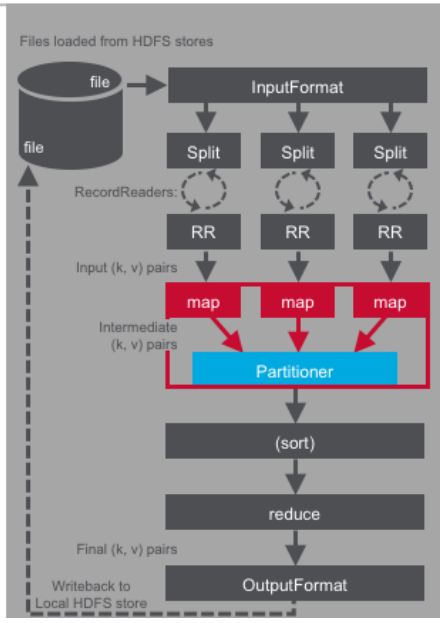
The map() method is implemented as part of the Mapper class. The map() method has three arguments: writable key, writable value, and context.

The default RecordReader (for text input) defines the key for the map() method as the byte offset of the record in the input file, and the value is simply the line at the byte offset.

The map method tokenizes the input value and then processes those tokens. The logic of what the map method actually does with each token in the value is completely up to you – the programmer. The map context object collects the output from all the calls to the map method and then passes that data to the partitioner, described next.

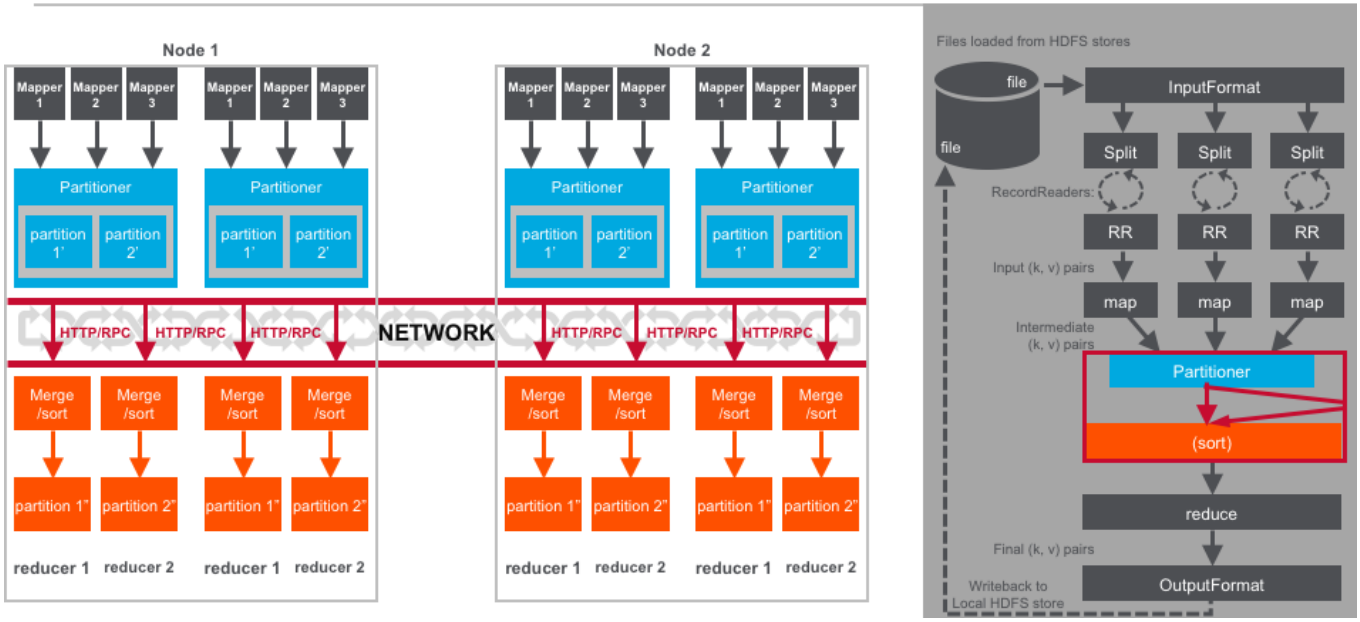
Partitioner Class





The partitioner takes the output generated by the map() method, hashes the record key, and creates partitions based on the hashed key. Each partition is earmarked for a specific reducer, so all the records with the same key will be written to the same partition (and therefore sent to the same reducer). This is the behavior of the default partitioner – you could override that and provide your own behavior.

Shuffle Phase

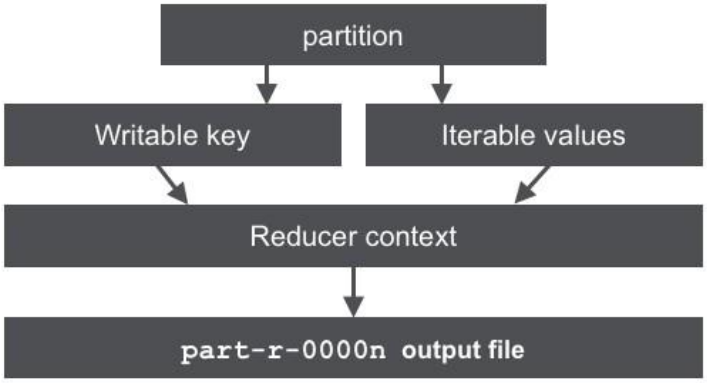


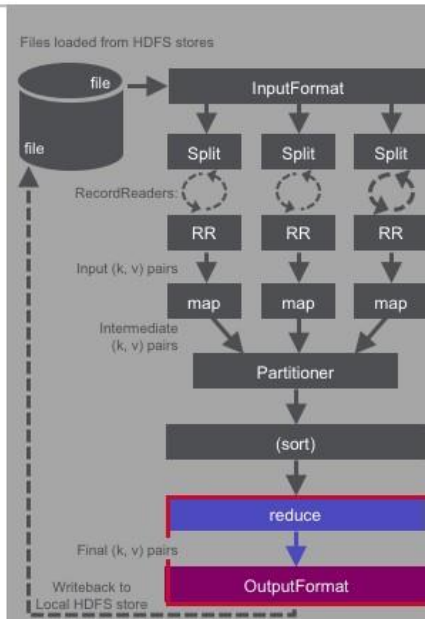
The partitions now are sorted and merged in preparation for sending to the reducers. Once an intermediate partition is complete, it is sent to the reducer over the network using a network protocol like HTTP or RPC. This is the part of the MapReduce program that is the most network intensive.

When a map task is finished, it signals its parent TaskTracker which in turn notifies the JobTracker. For a given job, the JobTracker knows the mapping between map outputs and TaskTrackers. The reducer periodically asks the JobTracker for map until it has retrieved them all.

TaskTrackers wait until they are told to delete the intermediate results by the JobTracker, which is some time after the job has completed. The map outputs are copied to the reduce TaskTracker's memory if they are small enough or written to disk. When the in-memory buffer reaches a threshold size or reaches a threshold number of map, it is merged and spilled to disk.

Reducer Class





The reduce() method is called for each key and the list of values associated with that key in the partition. The reduce() method processes each iterated value and writes the key and reduced list to the context. The OutputComitter in the context creates one output file per reducer that is run.

Results From MapReduce Job

- `_SUCCESS`
- `_logs/history*`
- `part-r-00000, part-r-00001, . . .`
- `part-m-00000, part-m-00001, . . .`

The results of a MapReduce job are written to a directory specified by the user. The contents of this directory are identified here. An empty file named _SUCCESS is created to indicate that the job completed successfully (though not necessarily without errors).

The history of the job is captured in the _logs/history* files.

The output of the reduce() method itself is captured in individual files (named part-r-00000 part-r-00001), one for each reducer. By keeping the files distinct for each reducer, there is no need to account for concurrency. Note that if you run a map-only job, the names of your output files will be different, starting with part-m instead of part-r, as shown here.

Next Steps

Build Hadoop MapReduce Applications

Lesson 2: Job Execution Framework

Build Hadoop MapReduce Applications, Lesson 2: Job Execution Framework.

Learning Goals

Learning Goals

2.1 Describe the MapReduce v1 job execution framework

2.2 Compare MapReduce v1 to MapReduce v2 (YARN)

2.3 Describe how jobs execute in YARN

2.4 Monitor jobs in YARN

Welcome to Lesson Two. In this lesson you will to compare and contrast MapReduce v1 to MapReduce v2 (YARN)
We will describe the MapReduce Job Execution framework. We will then define YARN or MapReduce v2, and describe the architecture. We will see how jobs are executed and managed in YARN and how it varies from MapReduce v1.

Learning Goals

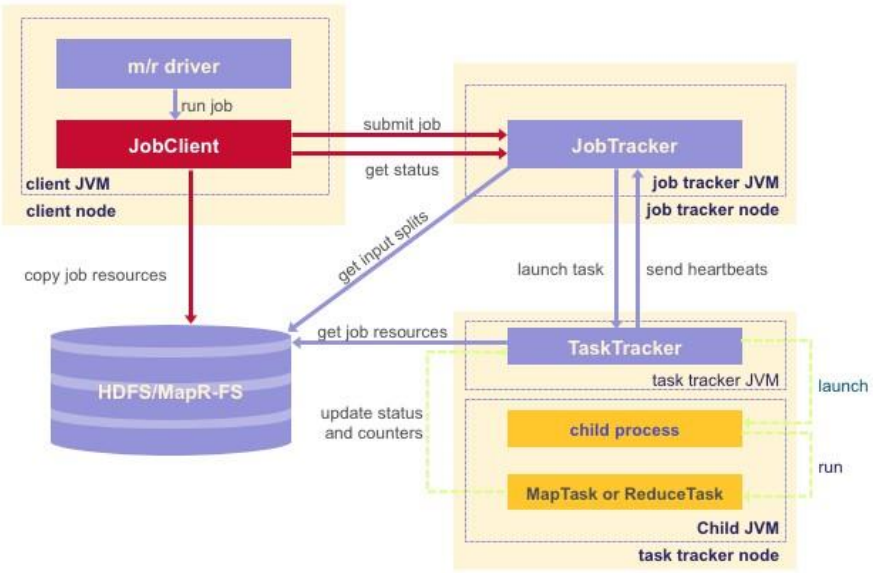
2.1 Describe the MapReduce v1 job execution framework

2.2 Compare MapReduce v1 to MapReduce v2 (YARN)

2.3 Describe how jobs execute in YARN

2.4 Monitor jobs in YARN

MapReduce Job Execution Framework



Now let's look at how Hadoop executes MapReduce jobs.

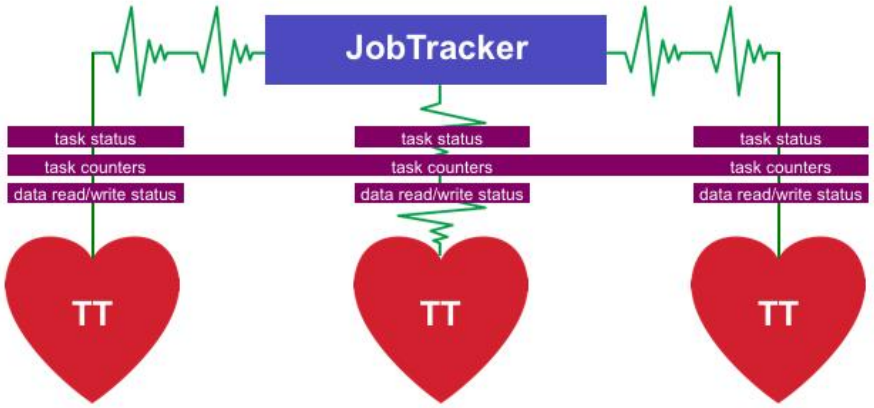
To begin, a user runs a MapReduce program on the client node which instantiates a JobClient object.

Next, the JobClient submits the job to the JobTracker.

Then the job tracker instantiates a Job object which gets sent to the appropriate task tracker. The task tracker launches a child process which in turns runs the map or reduce task.

Finally the task continuously updates the task tracker with status and counters.

How the Heartbeat is Used



When task trackers send heartbeats to the job tracker, they include other information such as task status, task counters and data read/write status.

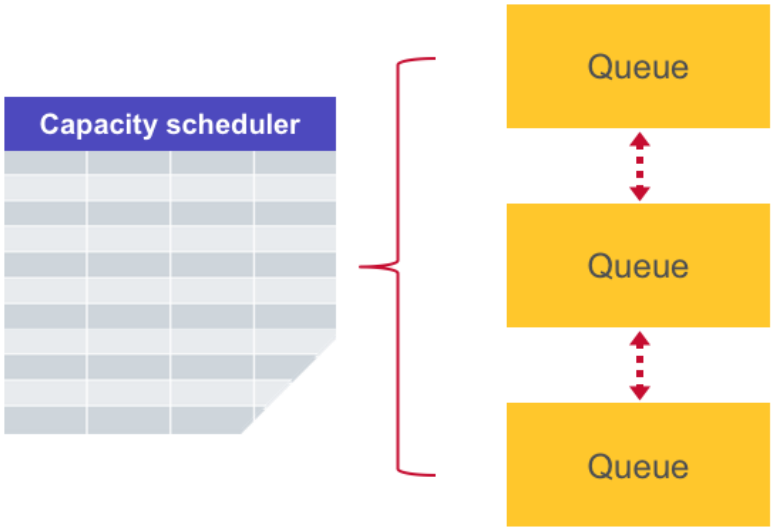
Heartbeats tell Job Tracker which Task Trackers are alive.

When Job Tracker stops receiving heartbeats from a Task Tracker then

Job Tracker reschedules tasks on failed Task Tracker to other Task Trackers Job Tracker marks Task Tracker as down and won't schedule subsequent tasks.

Similarly, data nodes piggyback their heartbeats with status reports (called block reports). Usually, a task tracker serves as a data node too.

Hadoop Job Scheduling

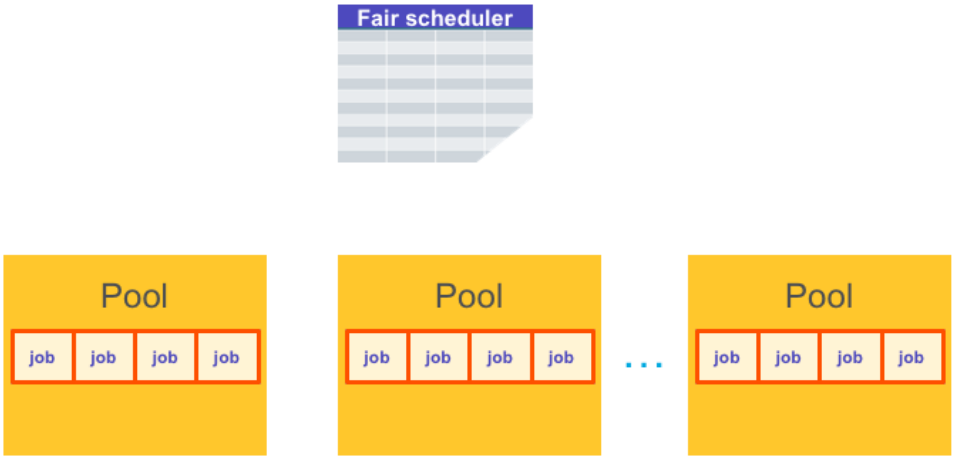


Two schedulers are available in Hadoop. The Fair scheduler is the default. Resources are shared evenly across pools and each user has its own pool by default. You can configure custom pools and guaranteed minimum access to pools to prevent starvation. This scheduler supports pre-emption.

There is also a Capacity scheduler where resources shared across queues. An Admin configures hierarchical queues to reflect organizations and their weighted access to resources. You can also configure soft and hard capacity limits to users within a queue. Queues have ACLs to prevent rogues from accessing queue. This scheduler supports resource-based scheduling and job priorities.

The effective scheduler is defined in the HADOOP_HOME/conf/mapred-site.xml file.

Hadoop Fair Scheduler

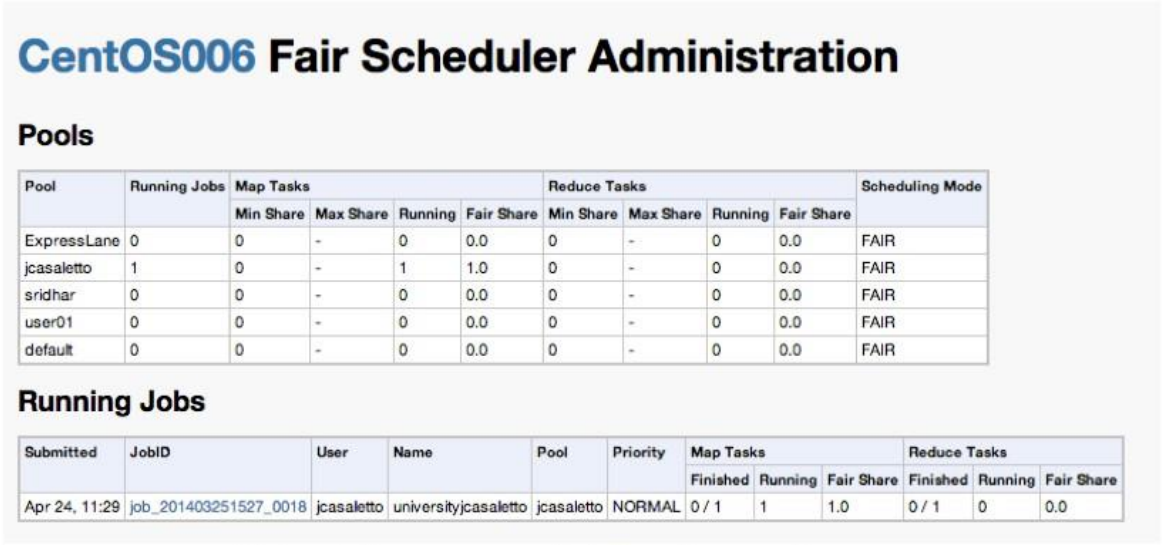


The goal of the fair scheduler is to give all users equitable access to pool resources. Each pool has configurable guaranteed capacity in slots. Jobs are placed in flat pools. The default is 1 pool per user.

The Fair Scheduler algorithm works by dividing each pool's minimum map and reduce tasks among jobs. When a slot is free, the algorithm will allocate a job below the minimum share or most starved. Jobs from "over-using" pools can be preempted.

The Fair Scheduler was developed at Facebook.

Fair Scheduler Web UI



CentOS006 Fair Scheduler Administration

Pools

Pool	Running Jobs	Map Tasks				Reduce Tasks				Scheduling Mode
		Min Share	Max Share	Running	Fair Share	Min Share	Max Share	Running	Fair Share	
ExpressLane	0	0	-	0	0.0	0	-	0	0.0	FAIR
jcasaletto	1	0	-	1	1.0	0	-	0	0.0	FAIR
sridhar	0	0	-	0	0.0	0	-	0	0.0	FAIR
user01	0	0	-	0	0.0	0	-	0	0.0	FAIR
default	0	0	-	0	0.0	0	-	0	0.0	FAIR

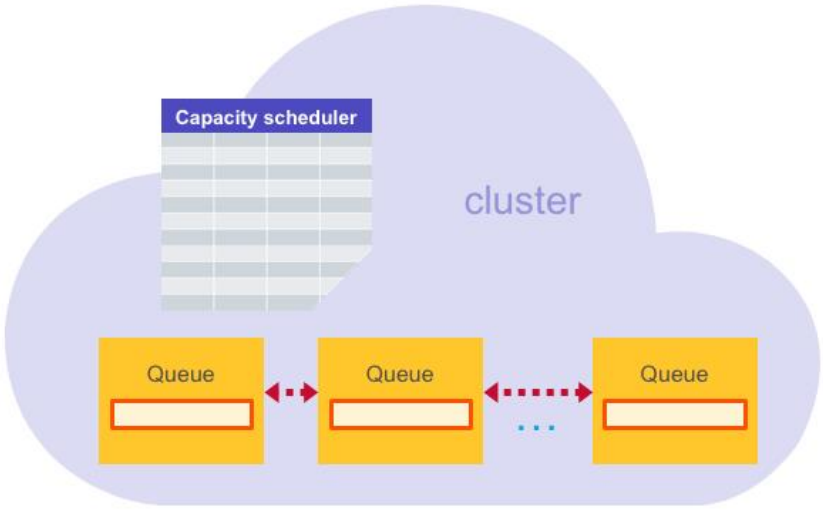
Running Jobs

Submitted	JobID	User	Name	Pool	Priority	Map Tasks			Reduce Tasks		
						Finished	Running	Fair Share	Finished	Running	Fair Share
Apr 24, 11:29	job_201403251527_0018	jcasaletto	universityjcasaletto	jcasaletto	NORMAL	0 / 1	1	1.0	0 / 1	0	0.0

Scheduler runs on JobTracker node

You can examine the configuration and status of the Hadoop fair scheduler in the MCS, or by using the Web UI running on the job tracker.

Hadoop Capacity Scheduler



The goal of the capacity scheduler give all queues access to cluster resources. Shares assigned to queues as percentages of total cluster resources

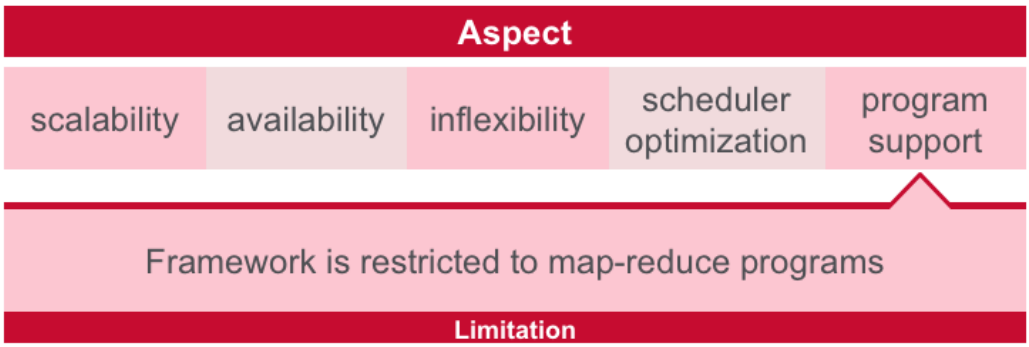
Each queue has configurable guaranteed capacity in slots.

Jobs are placed in hierarchical queues. The default is 1 queue per cluster.

Jobs within queue are FIFO. You can configure capacity.scheduler.xml for per-queue or per- user

This scheduler was developed at Yahoo.

Limitations in the Hadoop Execution Framework



There are some limitations in the MapReduce v1 framework. MaprReduce v2 or YARN addresses some of these issues. Click on each to learn more.

Learning Goals

Learning Goals

2.1 Describe the MapReduce v1 job execution framework

2.2 Compare MapReduce v1 to MapReduce v2 (YARN)

2.3 Describe how jobs execute in YARN

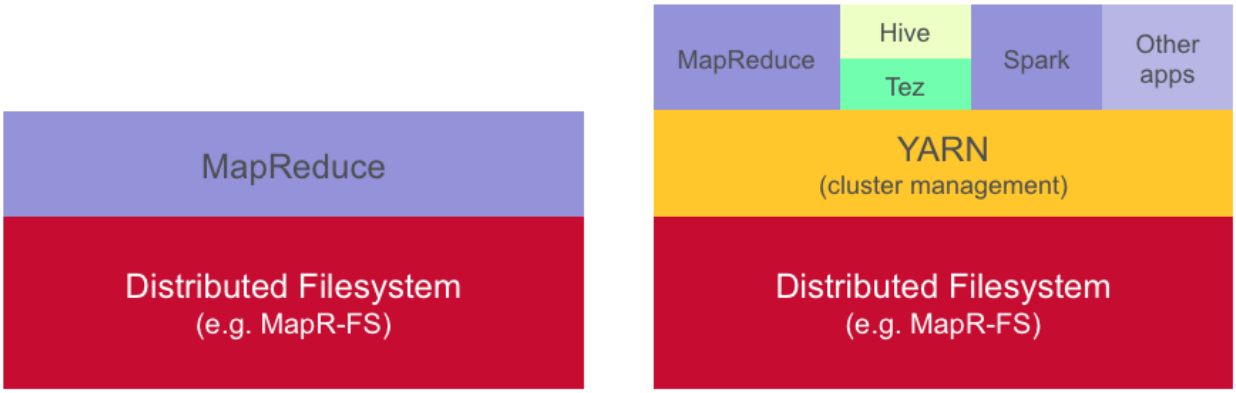
2.4 Monitor jobs in YARN

Motivation for YARN

1. **Map and reduce slot configuration is not dynamic**
   - Inflexibility leads to underutilization
   - ***No slot configuration in YARN***
2. **Hadoop framework only supports MapReduce jobs**
   - No support for non-MR apps
   - ***YARN supports MR and non-MR apps***
3. **Single job tracker has scalability limitations**
   - ~4000 nodes per cluster max
   - ***YARN equivalent of JT has multiple instances per cluster for scale***
4. **YARN uses same API and CLI as MRv1 (with similar Web UIs)**

The motivation for the development of YARN or MapReduce v2 was to further support and overcome some limitations of MapReduce v1. For example, Map and Reduce slot configuration is not dynamic. This inflexibility leads to underutilization. There is no slot configuration in YARN allowing it to be more flexible. Another limitation of MapReduce v1, is that the Hadoop framework only supports MapReduce jobs and not an non-MapReduce application. YARN supports both. Finally, the single Job Tracker in MapReduce v1 has a upper limit of 4000 nodes which limits scalability. The YARN equivalent of Job Tracker supports multiple instances per cluster to scale. Yarn uses the same API and CLI as MapReduce v1 which is similar to Web Uis.

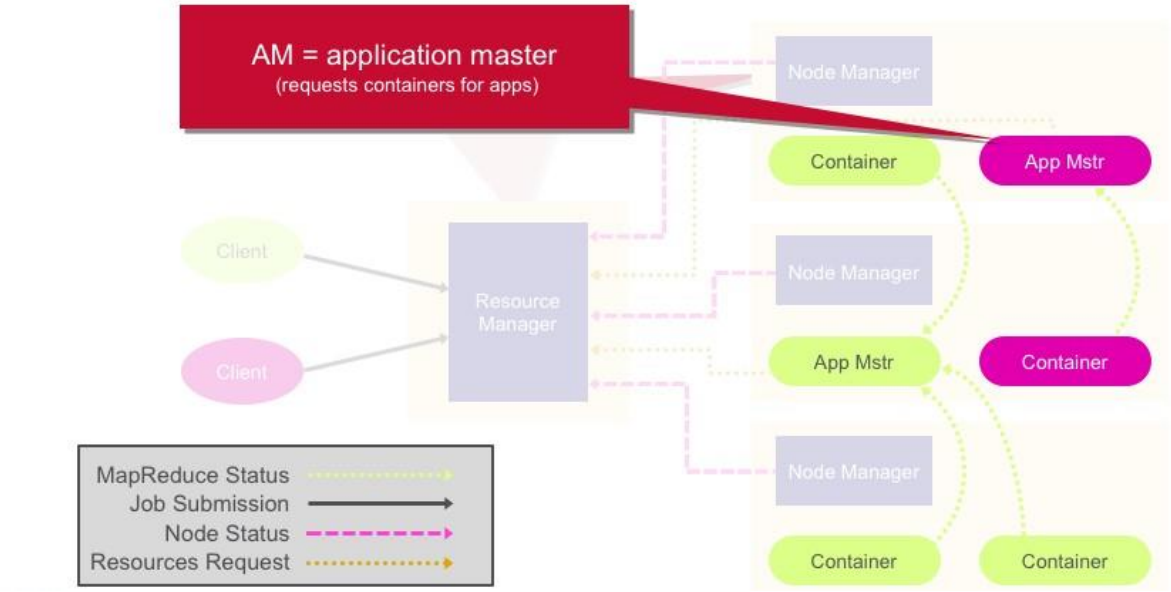
Differences MRv1 and MRv2



The main advancement in YARN architecture is the separation of resource management and job management, which were both handled by the same JobTracker process in Hadoop 1.x. Cluster resources and job scheduling are managed by the ResourceManager. Resource negotiation and job monitoring are managed by an ApplicationMaster for each application running on the cluster. In MapReduce, each node advertises a relatively fixed number of map slots and reduce slots. This can lead to resource under-utilization. For example, if there is a heavy reduce load and limited map slots are available, then some map slots cannot accept reduce tasks (and vice versa).

YARN generalizes resource management for use by new engines and frameworks, allowing resources to be allocated and reallocated for different concurrent applications sharing a cluster. Existing MapReduce applications can run on YARN without any changes. At the same time, because MapReduce is now merely another application on YARN, MapReduce is free to evolve independently of the resource management infrastructure.

YARN Architecture



Click on each to learn more.

RM = resource manager (creates/deletes containers, tracks NMs) NM = node manager (launches apps and AMs / reports status) AM = application manager (requests containers for apps) Container = logical envelope for resources (CPU, memory)

Knowledge Check

Knowledge Check

Which of the following are true of YARN?

A. There is no slot configuration in YARN

B. The YARN equivalent of Job Tracker supports multiple instances per cluster to scale.

C. YARN supports MapReduce and non-MapReduce jobs

D. All of the above

Learning Goals

Learning Goals

2.1 Describe the MapReduce v1 job execution framework

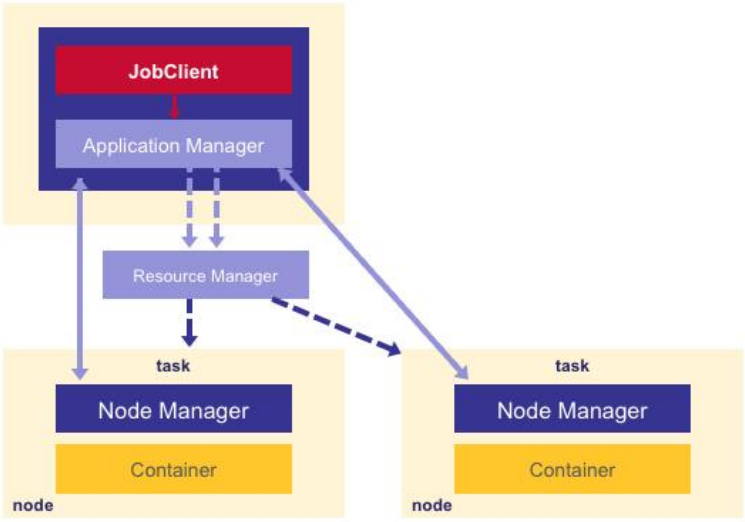
2.2 Compare MapReduce v1 to MapReduce v2 (YARN)

2.3 Describe how jobs execute in YARN

2.4 Monitor jobs in YARN

In this section, we describe how jobs execute in YARN.

MapReduce Job Lifecycle in MRv2



1. User submits app request by passing config for Application Manager to Resource Manager.
2.Resource Manager allocates a container for Application Manager on a node. Tells Node Manager in charge of that node to launch the Application Manager container.
3. Application Manager registers back with Resource Manager. Asks for more containers to
run tasks.
4. Resource Manager allocates the containers on different nodes in the cluster.
5.Application Manager talks directly to the Node Managers on those nodes to launch the containers for tasks.
6. Application Manager monitors the progress of the tasks.
7.When all the application's tasks are done, Application Manager unregisters itself from Resource Manager.
8. Resource Manager claims back the previously allocated containers for the application.

Non-MapReduce Job Lifecycle in MRv2

1. User submits app request by passing config for AM to RM
2. RM starts AM and allocates container
3. AM launches container and monitors it
4. When AM is done, it unregisters from RM

Let's discuss the steps of the starting and stopping of a job in MapReduce v2 that does not use MapReduce.

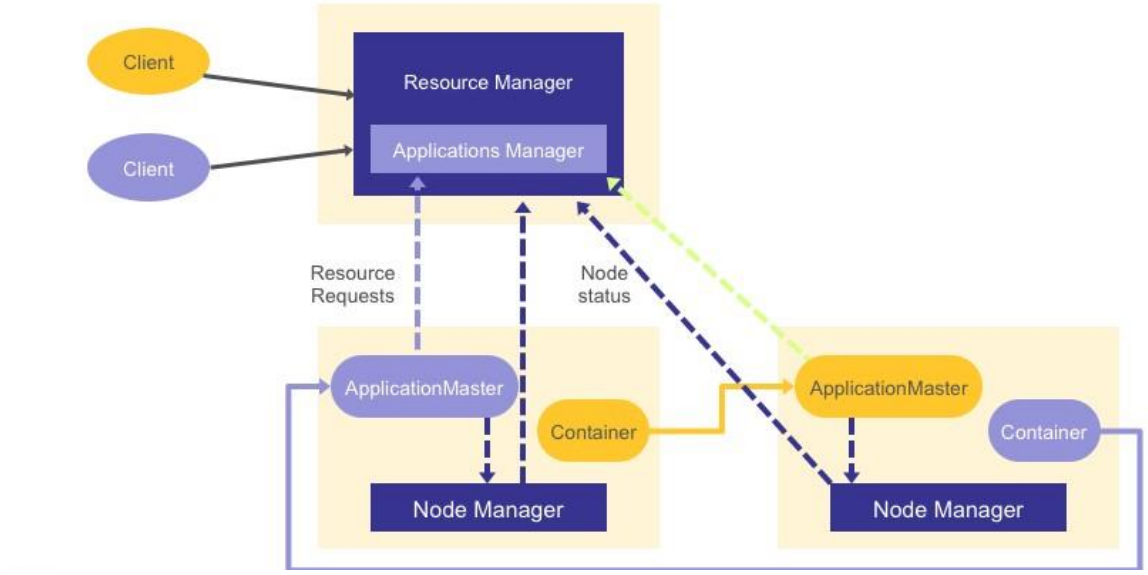
First the user submits an app request by passing configuration to the Application Master and to the Resource Manager.

Next, the Resource Manager starts the Application Master to allocate a container to the job.

Then the Application Master launches the container and monitors it.

When the Application Master is done, it then unregisters from the Resource Manager.

Describe How a Job is Launched in MRv2



Let's now describe in more detail how a job is launched in YARN.

First a client submits an application to the YARN Resource Manager, including the information required for the Container Launch Context.

Next the Application Manager, which is in the Resource Manager, negotiates a container and bootstraps the Application Master instance for the application.

Then the Application Master registers with the Resource Manager and requests containers.

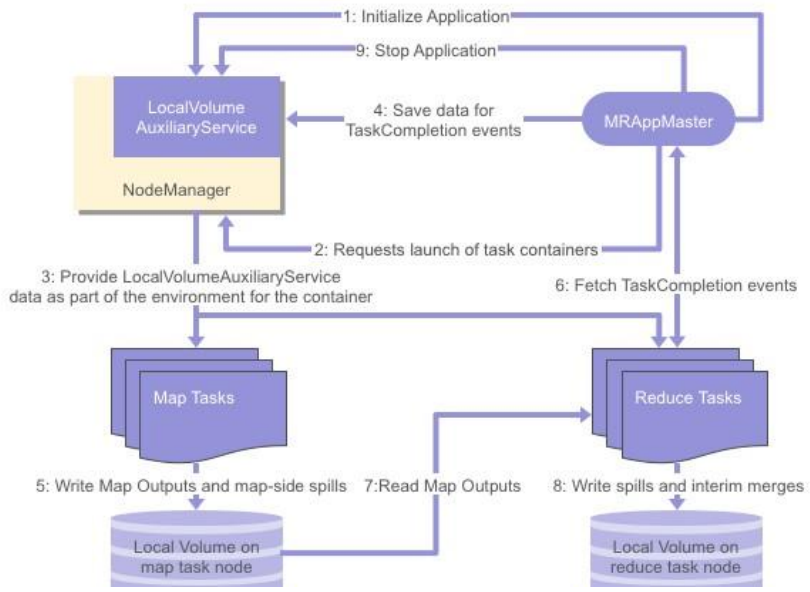
After that the Application Master communicates with Node Managers to launch the containers it has been granted, specifying the CLC for each container.

The the Application Master manages application execution. During execution, the application provides progress and status information to the Application Master. The client can monitor the application's status by querying the Resource Manager or by communicating directly with the Application Master.

The Application Master reports completion of the application to the Resource Manager.

Finally the Application Master un-registers with the Resource Manager, which then cleans up the Application Master container.

MapR Direct Shuffle in YARN



Now let's describe the MapR Direct Shuffle work flow in YARN. The Application Master service initializes the application by calling initialize Application() on the LocalVolumeAuxiliaryService.

Next the Application Master service requests task containers from the Resource Manager.

Then the Resource Manager sends the App Master information that AppMaster uses to request containers from the NodeManager.

After that the Node Manager on each node launches containers using information about the node's local volume from the LocalVolumeAuxiliaryService.

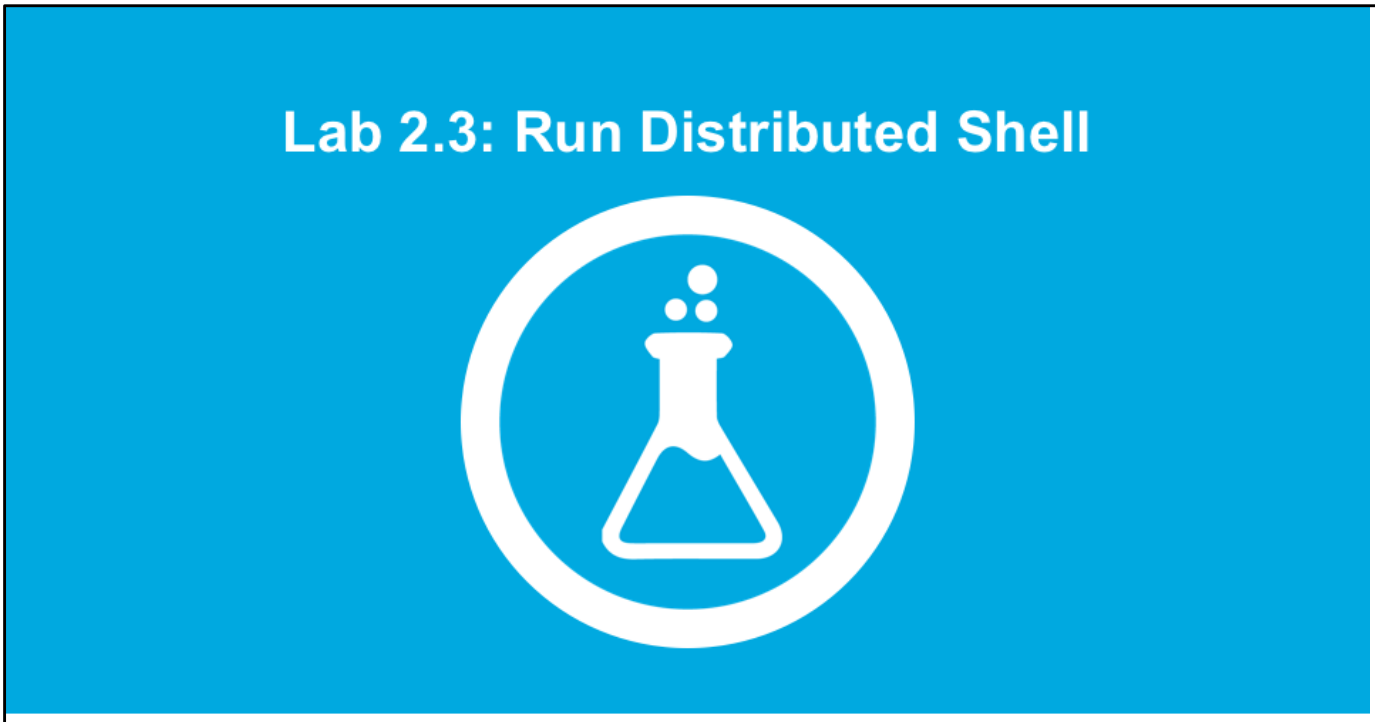
Data from map tasks is saved in the AppMaster for later use in TaskCompletion events which are requested by reduce tasks.

As the map tasks completes, map outputs and map-side spills are written to the local volumes on the map task nodes, generating Task Completion events.

ReduceTasks fetch Task Completion events from the Application Manager. The task Completion events include information on the location of map output data, enabling reduce tasks to copy data from MapOutput locations.

Reduce tasks read the map output information. Spills and interim merges are written to local volumes on the reduce task nodes.

Finally the Application Master calls stopApplication() on the LocalVolumeAuxiliaryService to clean up data on the local volume.



Lab 2.3: Run Distributed Shell

Learning Goals

Learning Goals

2.1 Describe the MapReduce v1 job execution framework

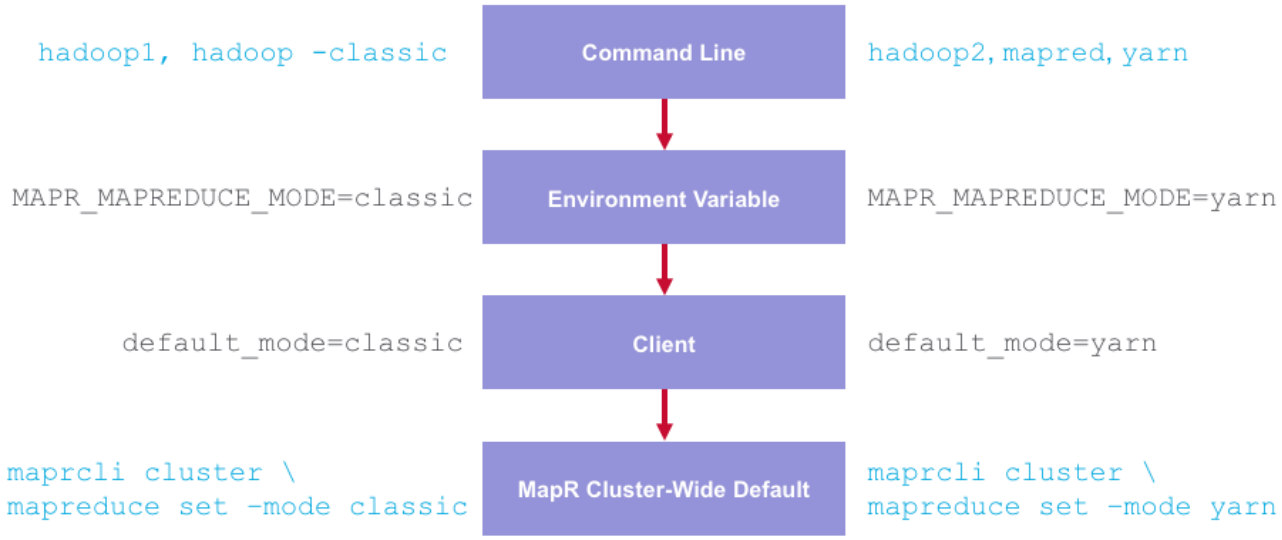
2.2 Compare MapReduce v1 to MapReduce v2 (YARN)

2.3 Describe how jobs execute in YARN

2.4 Monitor jobs in YARN

In this section, we describe how to monitor your jobs in a YARN environment.

MRv1 and MRv2 on the Same MapR Cluster



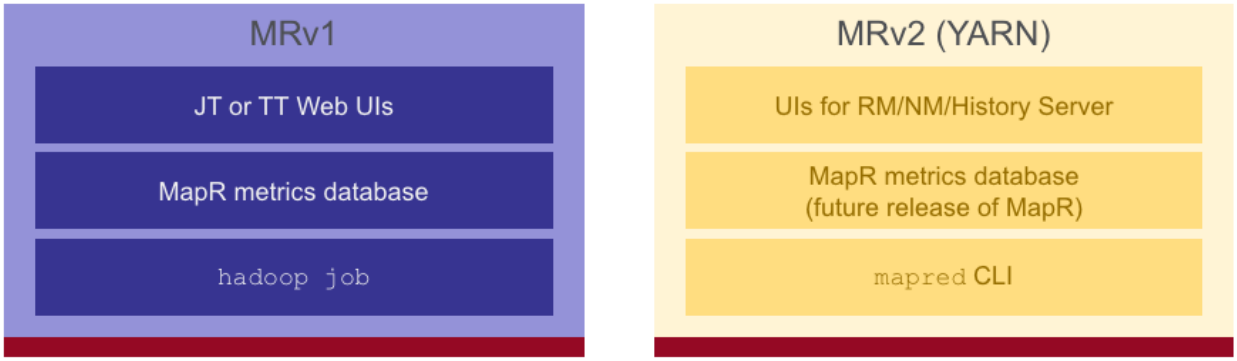
Users submitting and managing jobs can control which version of MapReduce to use by invoking a particular command. For MRv1, users run hadoop1 or use hadoop with the –classic option set. For MRv2, users run hadoop2 or mapred or yarn. The hadoop command itself is linked to MRv2 when both versions are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

The default setting on the cluster itself may be defined using the maprcli command. You can also perform this operation within the MCS.

Differences in MRv2 Job Management

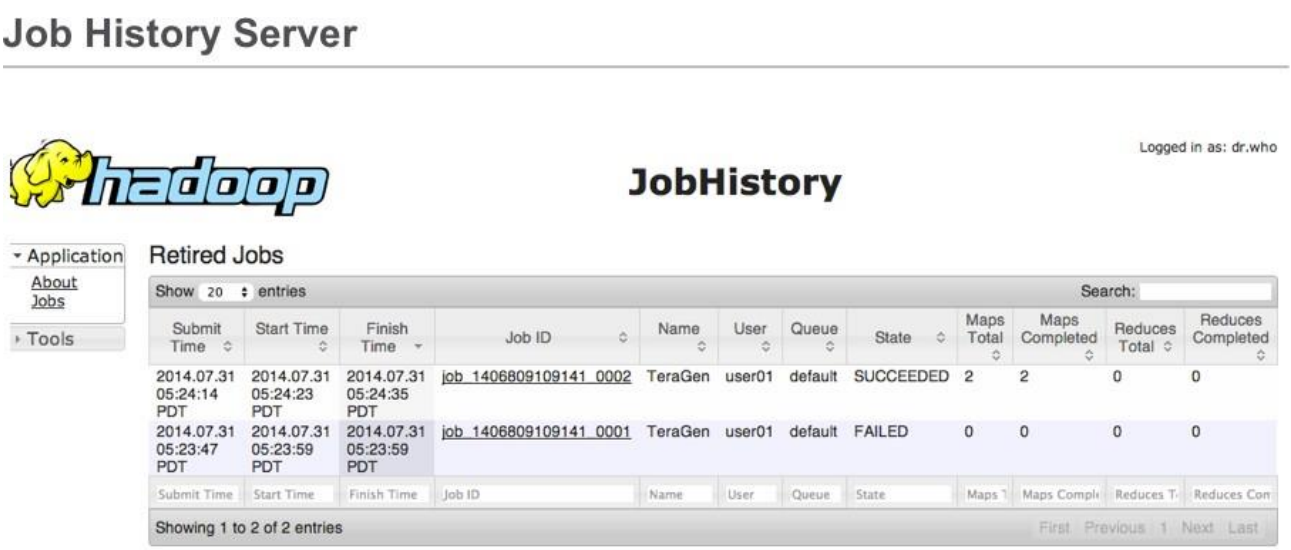


Let's look at the high-level differences of YARN job management.

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.

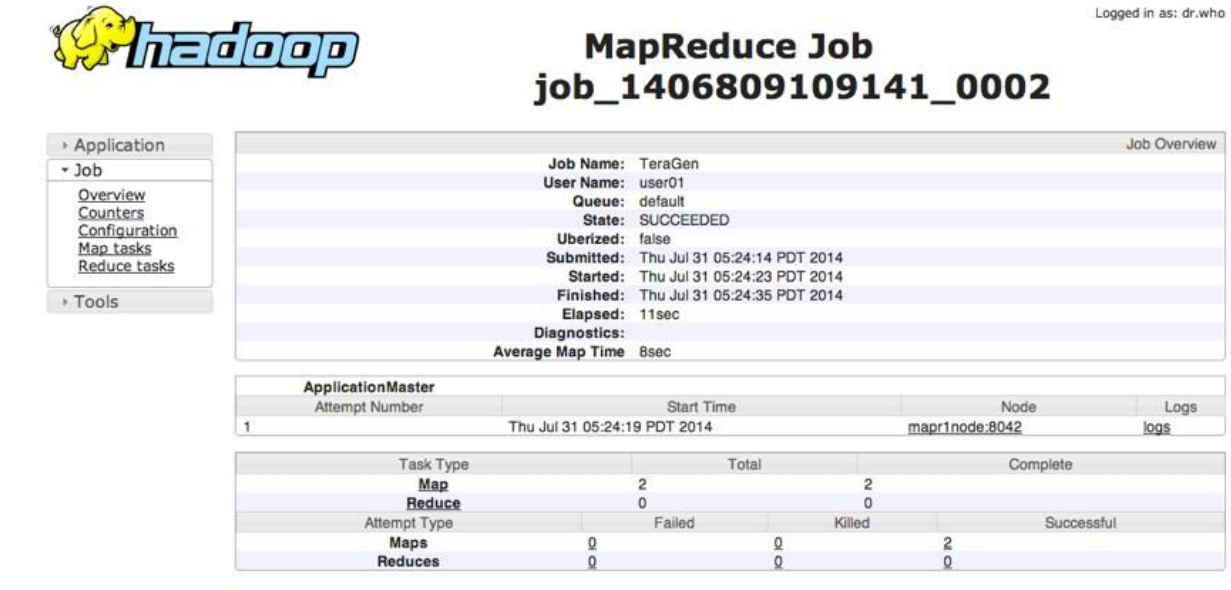
In MRv1, you can use the metrics database (available through the MCS) to monitor jobs. The metrics database will be available for YARN in a subsequent release of MapR software.

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.

Job History (3)

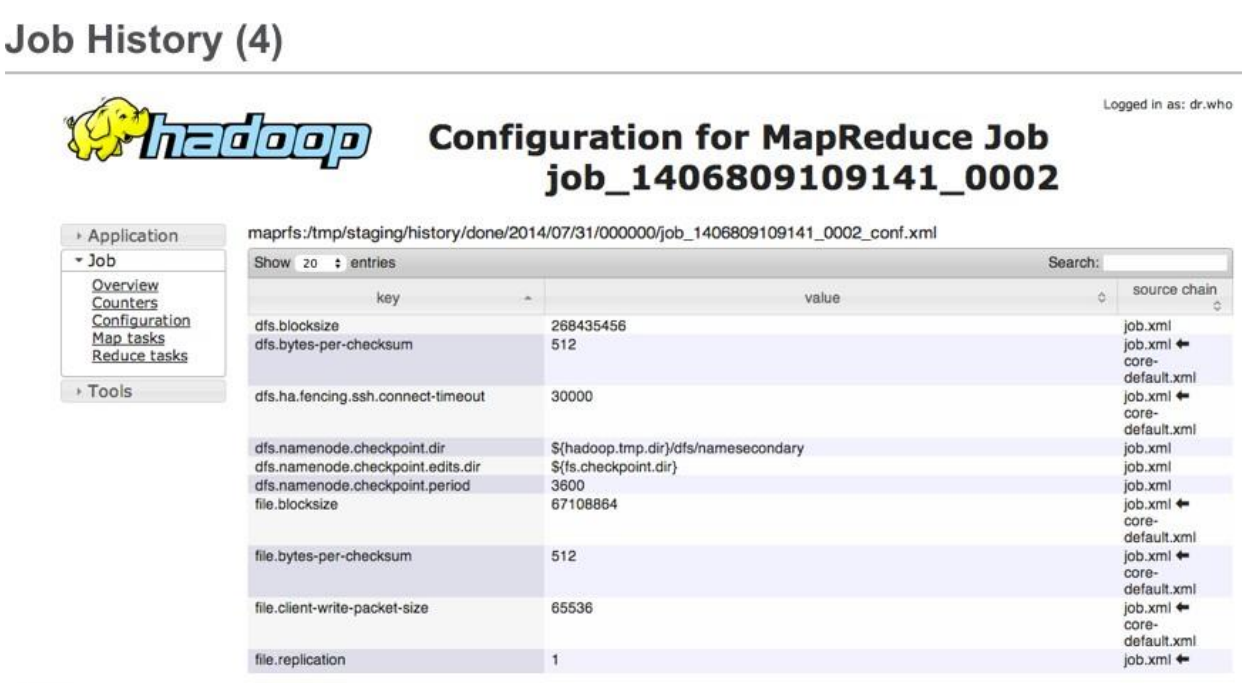
Logged in as: dr.who

Counters for job_1406809109141_0002

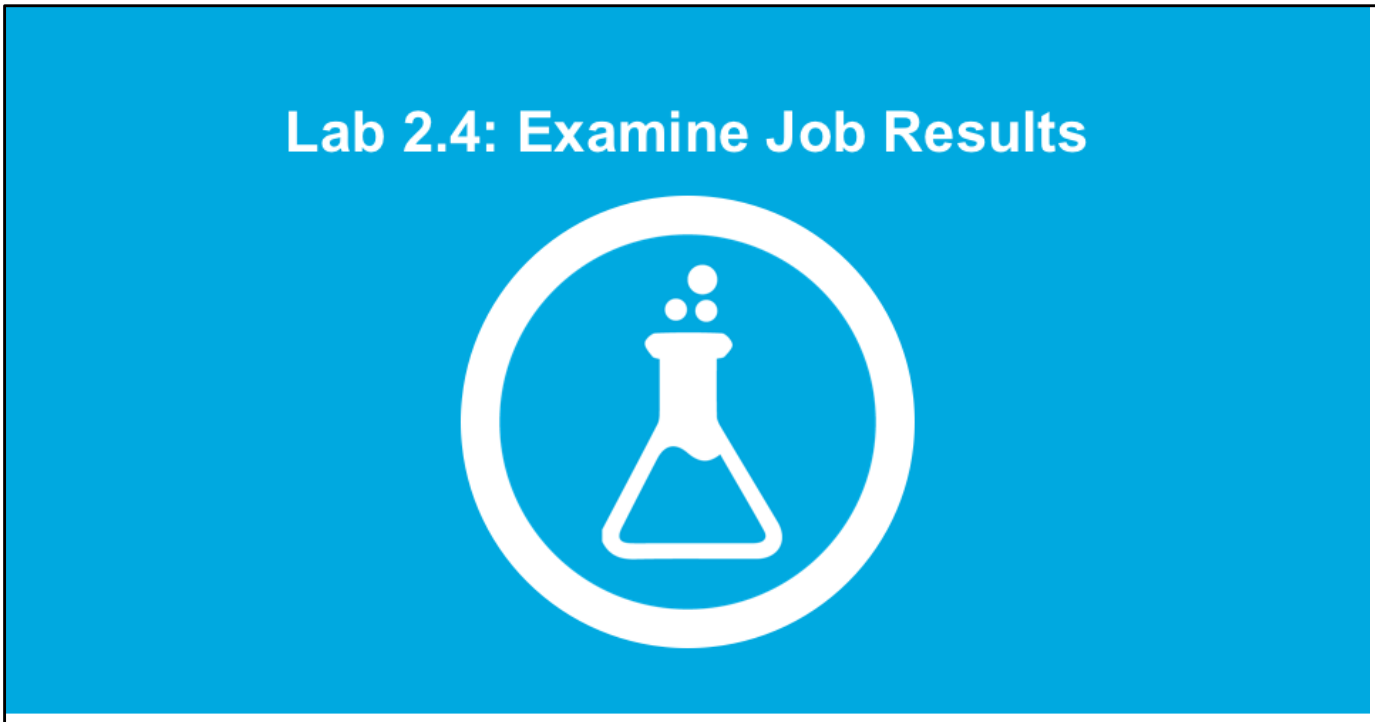
- Application
- Job
  - Overview
  - Counters
  - Configuration
  - Map tasks
  - Reduce tasks
- Tools

Counter Group	Counters			
File System Counters	Name	Map	Reduce	Total
	FILE: Number of bytes read	0	0	0
	FILE: Number of bytes written	130398	0	130398
	FILE: Number of large read operations	0	0	0
	FILE: Number of read operations	0	0	0
	FILE: Number of write operations	0	0	0
	MAPRFS: Number of bytes read	164	0	164
	MAPRFS: Number of bytes written	100000	0	100000
	MAPRFS: Number of large read operations	0	0	0
	MAPRFS: Number of read operations	14	0	14
	MAPRFS: Number of write operations	2012	0	2012
Job Counters	Name	Map	Reduce	Total
	Launched map tasks	0	0	2
	Other local map tasks	0	0	2
	Total time spent by all maps in occupied slots (ms)	0	0	16223
Map-Reduce Framework	Name	Map	Reduce	Total
	CPU time spent (ms)	320	0	320
	Failed Shuffles	0	0	0
	GC time elapsed (ms)	56	0	56
	Input split bytes	164	0	164
	Map input records	1000	0	1000
	Map output records	1000	0	1000

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.



Lab 2.4: Examine Job Results in History Server Web UI

Next Steps

Developing Hadoop Applications

Lesson 3: Write a MapReduce Program

Congratulations! You have completed Lesson Two! You should now have an understanding of how Hadoop executes MapReduce v1, what the motivation was to develop version two based on some limitations of version one. You should understand the differences between MR version one and version two. You should know how YARN executes and manages jobs. In the next lesson you will learn how to write a simple MapReduce program.

Build Hadoop MapReduce Applications

Lesson 3: Write a MapReduce Program

Build Hadoop MapReduce Applications, Lesson 3: Write a MapReduce Program.

Learning Goals

3.1 Summarize programming problem

3.2 Design and implement

- Mapper class
- Reducer class
- Driver class

3.3 Build and execute code

Welcome to Lesson Three. In this lesson we will learn about the programming problem, how to design, implement the Mapper, Reducer, and Driver classes. We will also build, execute code, and examine the output.

Learning Goals

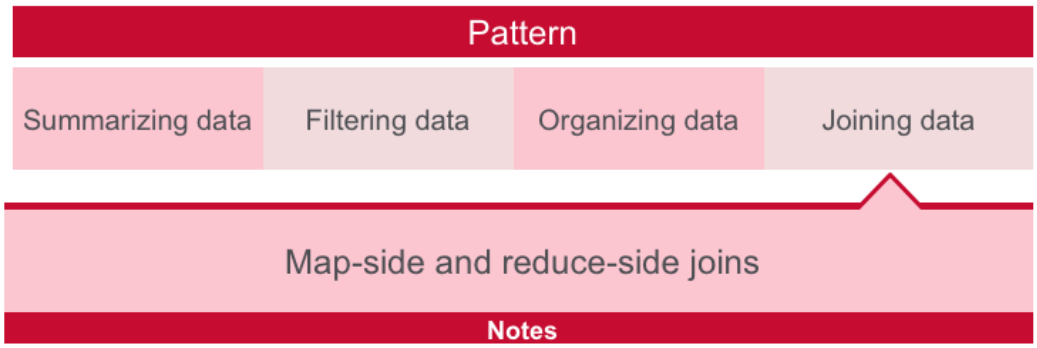
3.1 Summarize programming problem

3.2 Design and implement

- Mapper class
- Reducer class
- Driver class

3.3 Build and execute code

MapReduce Design Patterns



MapReduce may be used for a wide variety of batch-processed tasks.

Some MapReduce Programming Tips

Start with a template for the driver, mapper, and reducer classes

Modify the template to suit the needs of your application

Understand the flow and transformation of data:

- **Identify appropriate types for keys and values**

Let's look at some tips for MapReduce programming.

Most of the non-logic part of your MapReduce code is the same across all your applications. This includes import statements, class definitions, and method signatures. Construct a template and modify it according to the needs of each application rather than starting from scratch every time.

The most important aspect of writing MapReduce applications is understanding how data is transformed as it executes in the MapReduce framework. There are essentially four transformations from start to finish:

First, how data is transformed from the input files and fed into the mappers Then how is data transformed by the mappers
Next how data is sorted, merged, and presented to the reducer Last, how the reducers transform the data and write to output files

Similarly, it is important to use the appropriate types for your keys and values. There are several discussions in this course which provide details about which type of data to use for your input and output. Above all else, you must ensure that your input and output types match up, or your MapReduce code will not work. Since the input and output formats are derived from the same base class, your code may well compile but then fail at runtime.

Example Data Set

2000	2025191	1788950	236241	1544607	1458185	86422	480584	330765	149819
2001	1991082	1862846	128236	1483563	1516008	-32445	507519	346838	160681
2002	1853136	2010894	-157758	1337815	1655232	-317417	515321	355662	159659
2003	1782314	2159899	-377585	1258472	1796890	-538418	523842	363009	160833
2004	1880114	2292841	-412727	1345369	1913330	-567961	534745	379511	155234
2005	2153611	2471957	-318346	1576135	2069746	-493611	577476	402211	175265
2006	2406869	2655050	-248181	1798487	2232981	-434494	608382	422069	186313
2007	2567985	2728686	-160701	1932896	2275049	-342153	635089	453637	181452
2008	2523991	2982544	-458553	1865945	2507793	-641848	658046	474751	183295
2009	2104989	3517677	-1412688	1450980	3000661	-1549681	654009	517016	136993
2010	2162706	3457079	-1294373	1531019	2902397	-1371378	631687	554682	77005
2011	2303466	3603059	-1299593	1737678	3104453	-1366775	565788	498606	67182
2012	2450164	3537127	-1086963	1880663	3029539	-1148876	569501	507588	61913

year

delta

We will use this data set for our programming exercise example. There are a total of 10 fields of information in each line. Our programming objective is to find the minimum value in the delta column and the year associated with that minimum. We will ignore all the other fields of data.

Learning Goals

Learning Goals

3.1 Summarize programming problem

3.2 Design and implement

- **Mapper class**
- Reducer class
- Driver class

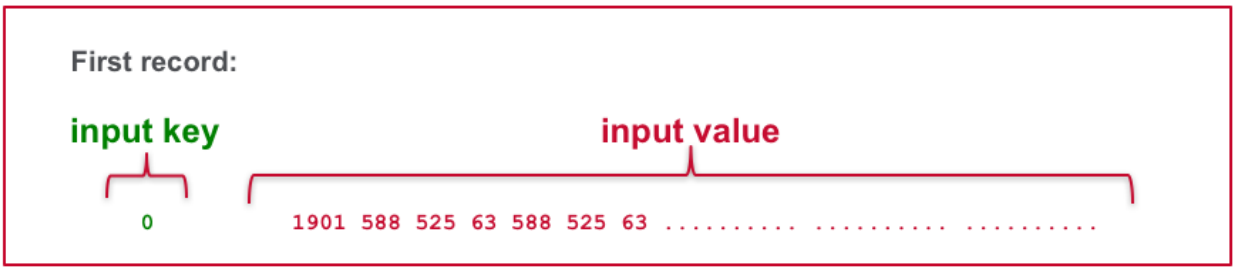
3.3 Build and execute code

Now let's design and implement the mapper class for our simple MapReduce program.

Input to the Mapper Class

Input format = `TextInputFormat`
Key = `LongWritable`
Value = `Text`

More detail later




```
StringTokenizer itr = new StringTokenizer(value.toString(),"\\s+");
```

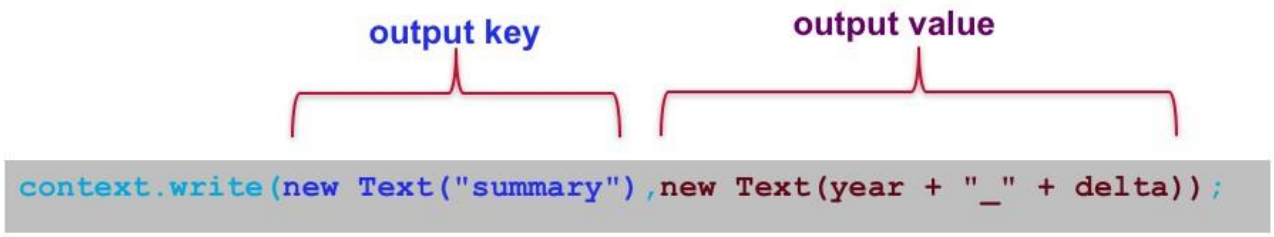
Now let's define and implement the Mapper class to solve the programming problem.

In this example, we will be using the TextInputFormat class as the type of input to the Mapper class. Using this format, the key from the default record reader associated with TextInputFormat is the byte offset into the file (LongWritable). We won't be using this key for anything in our program, so we will just ignore it. The value from the default record reader is the line read from the input file, in Text.

Let's look at the first record. The key of the first record is the byte offset to the line in the input file (the 0th byte). The value of the first record includes the year, number of receipts, outlays, and the delta (receipts – outlays). Recall that we are interested only in the first and fourth fields of the record value.

Since the record value is in Text format, we will use a StringTokenizer to break up the Text string into individual fields. Here we construct the StringTokenizer using white space as the delimiter.

Output of the Mapper Class



First few lines of output

```
summary    1901_63
summary    1902_77
summary    1903_45
summary    1904_-43
summary    1905_-23
summary    1906_25
```

After grabbing fields 1 and 4 from a record, the Mapper class emits the constant key ("summary") and a composite output value of the year (field 1) followed by an underscore followed by the delta (field 4). The first few lines of output are shown. From the first record in the data set, the mapper emits the key "summary" and the value 1901 underscore 63. From the second record in the data set, the mapper emits the same key "summary" and the value 1902 underscore 77. The mapper continues writing output for each record it reads from the file. Our file is rather small – it only goes from the year 1901 to the year 2012.

Note that since we hard-coded the key to always be the string "summary", there will be only one partition (and therefore only one reducer) when this mapreduce program is launched.

Design and Implement the Mapper Class

```
public class ReceiptsMapper  extends Mapper
<LongWritable,Text,Text,Text> {
   public void map(LongWritable key, Text value, Context context)
throws IOException, InterruptedException {
      StringTokenizer iterator = new
StringTokenizer(value.toString(),"\\s+");
      String year = iterator.nextToken();
      iterator.nextToken();
      iterator.nextToken();
      String delta= iterator.nextToken();
      context.write(new Text("summary"), new Text(year + "_" +
delta);
   }
}
```

Note that the code shown omits the import statements in order to conserve space in the text area.

The four arguments to the Mapper class represent input key type, input value type, output key type, and output value type. In this case, everything is of type Text except for the input key to the mapper, which is of type LongWritable to represent the byte offset of the record in the file.

The first two arguments to the map() method are the key and value which must match the types defined in the Mapper class definition. The third argument is the context which encapsulates the Hadoop job running context (configuration, record reader, record writer, status reporter, input split, and output committer). You can dereference those objects if you wish (e.g. update the task status through the status reporter or get a parameter value from the job configuration).

We create a StringTokenizer to iterate over the values in the record, separated by white

space. We call nextToken() which assigns the first field in the record to the year variable.

We then call nextToken() twice so that we can get to field 4, which is the delta. Last, we

emit the key (“summary”) and composite value (year_delta).


Knowledge Check

Knowledge Check

Which statements are true of Mapper class?

A. The four arguments to the Mapper class represent input key type, input value type, output key type, and output value type.

B. The mapper class calls the map() method

C. The first two arguments to the map() method are the key and value which must match the types defined in the Mapper class definition

D. All of the above

Learning Goals

3.1 Summarize programming problem

3.2 Design and implement

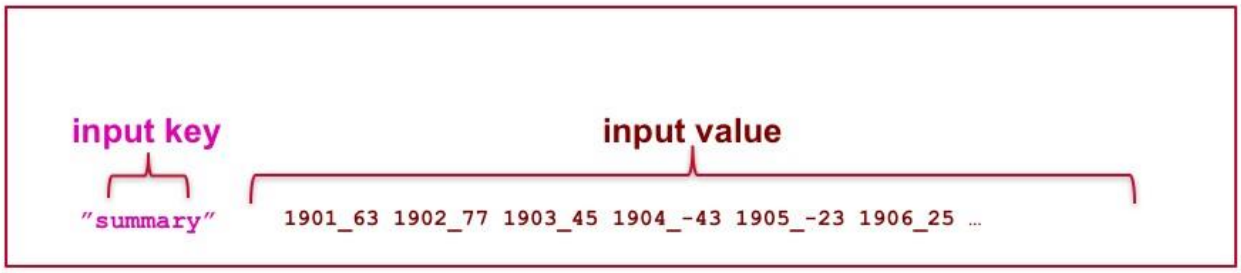
- Mapper class
- **Reducer class**
- Driver class

3.3 Build and execute code

Now let's design and implement the reducer class for our simple MapReduce program.

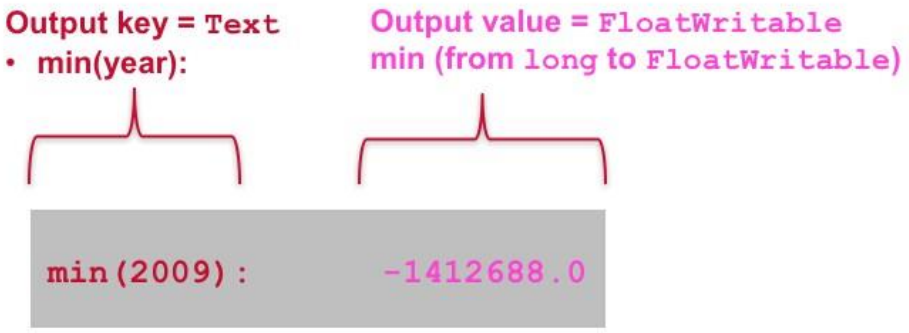
Input to the Reducer Class

- Mapper output key = `Text` → Reducer input key = `Text`
- Mapper output value = `Text` → Reducer input values = `Text`



Let's look at the input for the Reducer class. Recall that the output of the Mapper must match the input to the Reducer (both key and value types). Since the output key from the Mapper class is Text, the input key to the Reducer class must also be Text. Likewise, since the output value from the Mapper class is Text, the input value to the Reducer class must also be Text. Note there is a distinction between what is output from a single map() call and the whole set of intermediate results that the all the calls to map() produces. Specifically, the output of a single map() call is a single key-value pair. The Hadoop infrastructure performs a sort and merge operation on all those key-value pairs to produce a set of one or more partitions. When a call to reduce is made, it is made with all the values for a given key.

Output of the Reducer Class



The output of the reducer class must both conform to our solution requirements as well as match up to the data types specified in the source code. Recall that we defined the output key as Text. Note we are using FloatWritable as the output value type here because you will be calculating the mean value of the delta (a rational number) in the exercise associated with this lesson.

In the output shown here, the minimum delta was reported in the year 2009 as -1412688.

Design and Implement the Reducer Class (1)

```
public class ReceiptsReducer  extends Reducer
<Text,Text,Text,FloatWritable> {

   public void reduce(Text key, Iterable<Text> values, Context context)
throws IOException, InterruptedException {
     long tempValue = 0L, min=Long.MAX_VALUE;
     Text tempYear=null, tempValue=null, minYear=null, maxYear=null;
     String compositeString;
     String[] compositeStringArray;
```

Note in this example, code omits import statements in order to conserve space.

The four arguments to the Reducer class represent the type for input key, input value, output key, and output value. Recall that the output key and output value from the mapper must match the reducer's input key and input value by type.

The first two arguments to the reduce() method are the key and value, which match in type from the class definition. The third argument is the context which encapsulates the Hadoop job running context (configuration, status reporter, and output committer). As in the Mapper class, you can dereference those objects if you wish.

Design and Implement Reducer Class (2)

```
for (Text value: values) {
     compositeString = value.toString();
     compositeStringArray = compositeString.split("_");
     tempYear = new Text(compositeStringArray[0]);
     tempValue = new Long(compositeStringArray[1]).longValue();
     if(tempValue < min) {
        min=tempValue;
        minYear=tempYear;
     }
  }
  Text keyText = new Text("min" + "(" + minYear.toString() + "): ");
  context.write(keyText, new FloatWritable(min));
 }
}
```

We iterate over the all values associated with the key in the for loop.

We convert the Text value to a string (compositeString) so we can split out the year from the value (delta) for that year. We then convert that string into a string array (compositeStringArray) which splits out the compositeString variable based on the "_" character.

We pull out the year from the 0th element of the string array, and then we pull out the value as the "1th" element of the array.

We determine if we've found a global minimum delta, and if so, assign the min and minYear accordingly.

When we pop out of the loop, we have the global min delta and the year associated with the min. We emit the year and min delta.

Note that if the data set is partitioned into more than one partition, then we will have multiple output files, each with its "local" minimum calculated. In that case, we would need to do further processing to calculate the global minimum over the whole data set. It's for this reason we hardcoded the key – to guarantee we only have one partition and therefore one reducer.


Knowledge Check

Knowledge Check

Which statements are true of the Reducer class?

A. The four arguments to the Reducer class represent the type for input key, input value, output key, and output value

B. If the output value of the Mapper is Text, the input value to the Reducer can be LongWritable

C. If the output key of the Mapper is Text, the input key to the Reducer class must also be Text.

D. All of the above

E. A and C only

Learning Goals

3.1 Summarize programming problem

3.2 Design and implement

- Mapper class
- Reducer class
- **Driver class**

3.3 Build and execute code

Now let's design and implement the driver for our simple MapReduce program.

Implement the Driver (1)

```
public class ReceiptsDriver extends Configured implements Tool {
   public int run(String[] args) throws Exception {
      if (args.length != 2) {
         System.err.printf("usage: %s [generic options] <inputfile>
<outputdir>\n", getClass().getSimpleName());
         System.exit(1);
      }
    Job job = new Job(getConf(), "my receipts");
    job.setJarByClass(ReceiptsDriver.class);
    job.setMapperClass(ReceiptsMapper.class);
    job.setReducerClass(ReceiptsReducer.class);
```

Let's look at the first part of the driver class.

It first checks the count of the command-line arguments provided and prints a usage statement if the argument count is not 2. Note that we don't check for the existence of the input file or output directory. The Hadoop framework will fail if either the input file doesn't exist or the output directory exists and is non-empty.

After checking the invocation of the command, the code instantiates a new Job object with the existing configuration and names the job "my receipts".

The rest of the code above sets values for the job, including the driver, mapper, and reducer classes used.

Implement the Driver (2)

```
    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(FloatWritable.class);
    job.setMapOutputValueClass(Text.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    return job.waitForCompletion(true) ? 0 : 1;
    }
    public static void main(String[] args) throws Exception {
       Configuration conf = new Configuration();
       System.exit(ToolRunner.run(conf, new ReceiptsDriver(), args));
    }
}
```

Now let's look at the second part of the driver implementation as shown here.

We define the types for output key and value in the job as Text and FloatWritable respectively. If the mapper and reducer classes do NOT use the same output key and value types, we must specify for the mapper. In this case, the output value type of the mapper is Text, while the output value type of the reducer is FloatWritable.

There are 2 ways to launch the job – syncronously and asyncronously. The job.waitForCompletion() launches the job syncronously. The driver code will block waiting for the job to complete at this line. The true argument informs the framework to write verbose output to the controlling terminal of the job.

The main() method is the entry point for the driver. In it, we instantiate a new Configuration object for the job. We then call the ToolRunner static run() method.

Knowledge Check

Which statements are true of the Driver class?

A. The Driver class first checks the invocation of the command (checks the count of the command-line arguments provided)

B. It sets values for the job, including the driver, mapper, and reducer classes used.

C. In the Driver class, we can specify how we want to launch the job – synchronously or asychronously

D. All of the above

Learning Goals

Learning Goals

3.1 Summarize programming problem

3.2 Design and implement

- Mapper class
- Reducer class
- Driver class

3.3 Build and execute code

Now let's learn how to build and execute our simple MapReduce program.

Configure the Environment

```
$ cat ~/.profile

export HADOOP_HOME=/opt/mapr/hadoop/hadoop-<version>

export LD_LIBRARY_PATH=$HADOOP_HOME/lib/native/Linux-amd64-64

export PATH=$HADOOP_HOME/bin:$PATH

export CLASSPATH=$HADOOP_HOME/*:$HADOOP_HOME/lib/*

export HADOOP_CLASSPATH=$CLASSPATH
```

This example assumes you are a bash user for which the .profile file is the initialization file. If you use another shell, there is a different shell initialization file.

You are not obliged to define the HADOOP_HOME environment variable, but setting it as shown here allows you to reference the value of the HADOOP_HOME variable when defining other variables.

The LD_LIBRARY_PATH environment variable defines the path to your library files for executables. You are not obliged to define the LD_LIBRARY_PATH environment variable when running Hadoop jobs on a MapR cluster, but setting it as shown here uses libraries that are specifically compiled for the MapR distribution. Using Hadoop native libraries improves the performance of your MapReduce jobs by using compiled object code rather than Java byte codes.

The PATH environment variable defines a list of directories where your executables are located. You are not obliged to define the PATH variable, but setting it as shown in the slide above allows you to reference commands directly without providing the absolute path to the executable. The order in the PATH environment variable is important. If you wish to execute the hadoop command from the HADOOP_HOME (and not the one installed in your default PATH), you must put $HADOOP_HOME/bin first in your PATH statement.

You could optionally define your CLASSPATH environment variable in your shell environment to point to all the jars in the Hadoop distribution required to compile and run your MapReduce programs. If you don't specify the CLASSPATH variable in your shell environment, then you'll need to specify it at compjile time and run time. Similarly, you can optionally define the HADOOP_CLASSPATH variable to make it easier to run MapReduce applications with the hadoop command.

Build the Jar

```
$ mkdir classes

$ javac -d classes ReceiptsMapper.java

$ javac -d classes ReceiptsReducer.java

$ jar -cvf Receipts.jar -C classes/ .

$ javac -classpath $CLASSPATH:Receipts.jar -d classes
ReceiptsDriver.java

$ jar -uvf Receipts.jar -C classes/ .
```

This example shows the commands used to compile our 3 classes. First we'll create the directory "classes" to contain our class files.

The javac commands compile the mapper and reducer classes, and puts the compiled class files into a directory called "classes".

The jar command puts the mapper and reducer classes into a jar file. We include the new jar file in the classpath when we build the driver, after which we add the driver class to the existing jar file.

Launch the Hadoop Job

```
$ hadoop jar Receipts.jar Receipts.ReceiptsDriver \
/user/user01/RECEIPTS/DATA/receipts.txt \
/user/user01/RECEIPTS/OUT
```

The hadoop command launches the Hadoop job for our MapReduce example.

The hadoop command will use the “Receipts.jar” file we just created which contains the driver, mapper, and reducer classes.

The input to the MapReduce program is the first argument (/user/user01/RECEIPTS/DATA/receipts.txt) and the output to the MapReduce program is the second argument (/user/user01//RECEIPTS/OUT). These are the values associated with args[0] and args[1] in the driver, respectively.

Examine the Output

```
$ hadoop fs -cat /user/user01/RECEIPTS/OUT/part-r-00000
min(2009):   -1412688.0
```

This output file was created by our Reducer. It contains the statistics that our solution asked for – minimum delta and the year it occurred.

Lab 3.3: Modify a MapReduce Program

Open your lab guide to complete lab 3.3.

Congratulations! You have now completed Lesson Three! You should now know about the programming problem, how to design, implement the Mapper, Reducer, and Driver classes. You should also be able to build, execute code, and examine the output. In the next lesson you will examine the Mapper input processing and Reducer output processing data flow as well as explore the Mapper, Reducer, and Job class API.

Manage and Test Hadoop MapReduce Applications

Lesson 4: Use the MapReduce API

Learning Goals

Learning Goals

4.1 API overview

4.2 Examine

- Mapper input processing data flow
- Reducer output processing data flow

4.3 Explore the Mapper, Reducer and Job class API

Welcome to Lesson Four. In this lesson you will review the MapReduce API, examine the Mapper input processing and Reducer output processing data flow as well as explore the Mapper, Reducer, and Job class API.

Learning Goals

4.1 API overview

4.2 Examine

- Mapper input processing data flow
- Reducer output processing data flow

4.3 Explore the Mapper, Reducer and Job class API

Hadoop Version Support on MapR

Each MapR software release supports and ships with a specific version of Hadoop. For example, MapR 3.0.1 shipped with API versions 0.20.2 while MapR 4.0.1 supports both 0.20.2 and 2.4 including YARN.

Note that Apache does not keep older versions of the API available online, so you'll have to download the Hadoop source to access the documentation. It's a good idea to download the source code anyway – sometimes the best documentation is the source itself.

mapred and mapreduce

Aspect	mapred	mapreduce
Supported on MapR	yes	yes
Deprecated	no	no
YARN-compatible	yes	yes
Types	interfaces	abstract classes

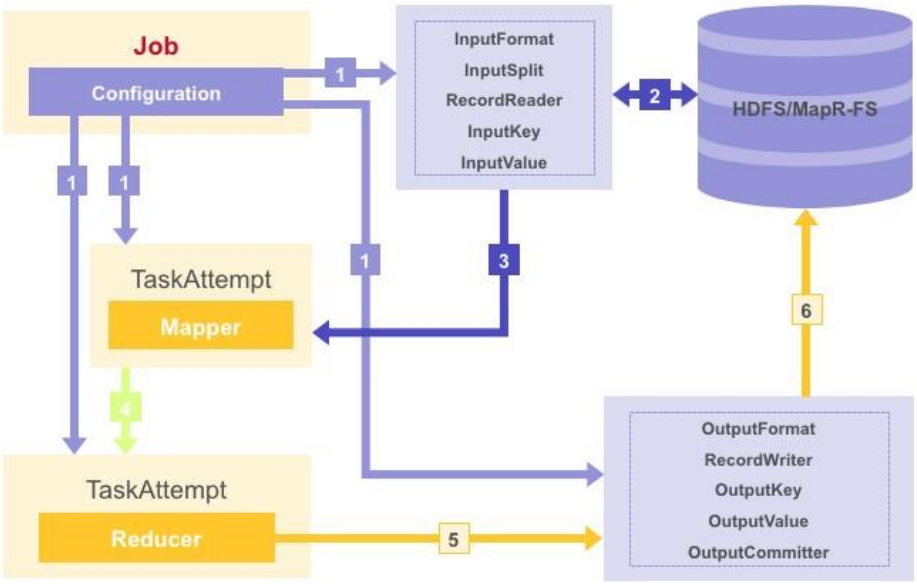
Objects	`OutputCollector`
	`Reporter`
	`JobConf`
		`Context`

Methods	`map() reduce()`	`map() reduce()`
		`cleanup() setup()`
		`run()`

Output files	`part-nnnnn`	`part-r-nnnnn` `part-m-nnnnn`
Reducer input values	`java.lang.iterable`	`java.lang.iterator`

There are 2 API packages to choose when developing MapReduce applications: org.apache.hadoop.mapred and org.apache.hadoop.mapreduce. The mapred package was deprecated in the Apache Hadoop project when the mapreduce package was introduced. Shortly thereafter, it was un-deprecated. Use the API which is most flexible for your use. Note that this course will use a mixture of each API. Also note that when you work with existing
MapReduce libraries, they may use one or the other version, so it's important to recognize the differences. Perhaps the biggest difference between them are the setup(), run(), and cleanup() methods in the mapreduce package. Exposing these methods provides a lot more flexibility and power to the programmer.

Interactions Between Objects



Let's discuss the relationships between the objects in a running Hadoop job.

The driver class instantiates a Job object and it's configuration, and then uses set() methods to define each element of the Configuration object.

The Mappers read data from the Hadoop file system to do its processing. The Mappers leverage the InputFormat, InputSplit, RecordReader, InputKey, and InputValue types to process input.

The framework shuffles output from the mappers to the reducers. The reducers write data to the Hadoop file system after they are done processing. The reducers leverage the outputformat, recordwriter, outputkey, and outputvalue, and outputcommitter types to process output.

WritableComparable Types

```
public interface WritableComparable extends Writable, Comparable
{
    void readFields(DataInput in);
    void write(DataOutput out);
    int compareTo(WritableComparable o)
}
```

Writable is an interface from Hadoop which is used to serialize keys and values before they are written to a stream (network or storage). Each java primitive has a corresponding Writable class (for example the integer java primitive is represented as a Hadoop IntWritable). The IntWritable and LongWritable classes have a variable length Writable type that can make your memory footprint more compact wherever possible.

All keys and values must implement the Writable interface, and all keys must further implement the Comparable interface.

Since key types and value types are often interchanged, there is an interface called WritableComparable which extends both interfaces.

The compareTo() method returns -1, 0, or 1 depending on the lexicographic comparison between an object and the input to the method. The framework relies on this method to sort key-value pairs by key.

Learning Goals

Learning Goals

4.1 API overview

4.2 **Examine**

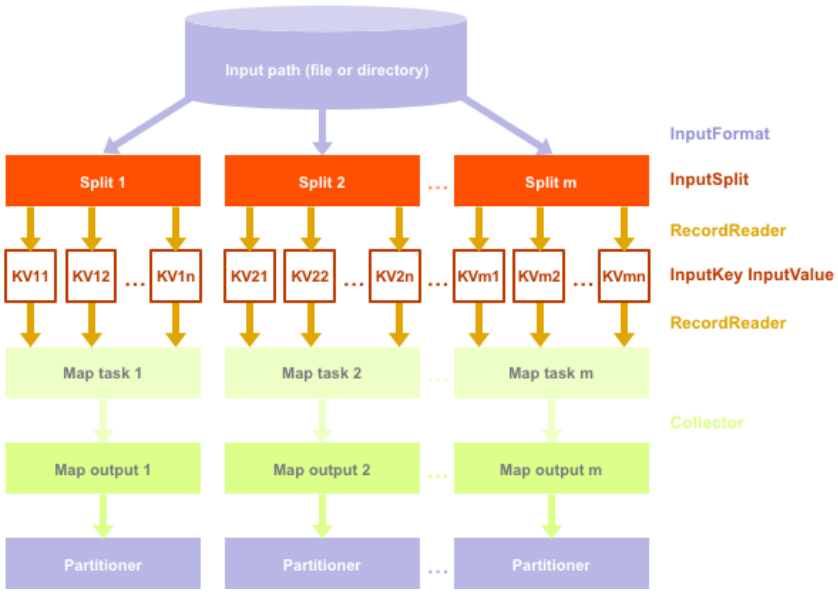
Mapper input processing data flow

Reducer output processing data flow

4.3 Explore the Mapper, Reducer and Job class API

In section we describe the Mapper input processing data flow.

Mapper Input Flow



Input files are split using an implementation of the InputFormat class. Key-value pairs from the input splits are generated for each split using the RecordReader class. The InputKey and InputValue objects are subclasses of the Writable class. All the key-value pairs from a given input split are sent to the same Mapper. The map() method is called once for each key-value pair, and all the results for each mapper are collected and sent to the partitioner which breaks out results based on hashed key values. These results are stored in the file system local to the Mapper task where they await pickup by the Hadoop framework running on the Reducer nodes to perform the shuffle and sort phase.

InputFormat Class

```
public abstract class InputFormat<K, V> {

 // returns array of InputSplits for job
 public abstract List<InputSplit> getSplits(JobContext) throws
IOException, InterruptedException;

 // create new record reader
 public abstract RecordReader<K, V> createRecordReader(InputSplit
split, TaskAttemptContext context) throws IOException,
InterruptedException;

}
```

Implementations include:

- `TextInputFormat` (for single-line records in text files)
- `SequenceFileInputFormat` (for binary files)

The InputFormat class performs the following:

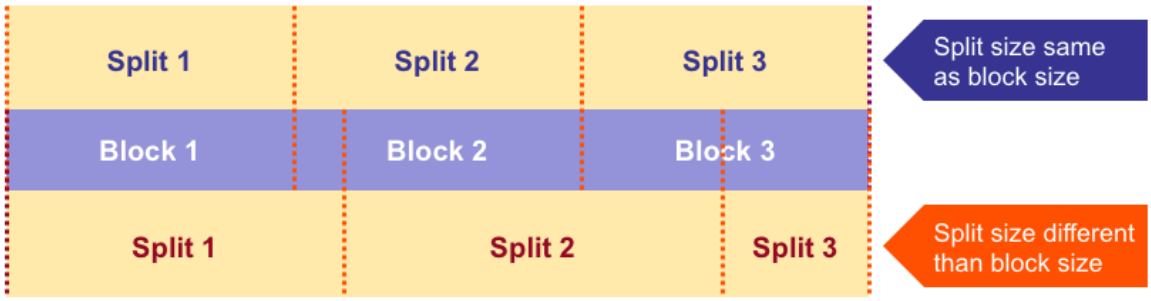
1. Validate the input files and directories that exist for the job. It throws IOException on invalid input.
2. Split the input files into InputSplits
3. Instantiate the RecordReader to be used for parsing out records from each input split

File-based input formats split the input into logical splits based on the total size of the input files.

TextInputFormat is a subclass of FileInputFormat you can use for plain text files. TextInputFormat breaks files into lines (terminated by new-line character). The key is the offset into the file and the value is the line of text up to the new-line character. Note that the TextInputFormat class is not parameterized -- the key (LongWritable) and value (Text) are hard-coded.

SequenceFileInputFormat is a subclass of FileInputFormat you can use for Hadoop compressed binary files. SequenceFileInputFormat breaks files into key/value pairs (as defined by the Reader). Sequence files are either uncompressed, record-compressed, or block-compressed. Unlike TextInputFormat, the SequenceFileInputFormat class is parameterized on the key and value types.

Input Split Boundaries



The graphic above depicts a common situation where the input splits may not line up exactly with block sizes. The defaultsize of an input split is the same as the default block size, but it is configurable. Note that an input split may be smaller, larger, or the same size as the block size, depending on the configuration the user provides to the mapreduce job.

There are a few reasons you might change the size of an input split. Adjusting it smaller or larger than the block size will influence the number of mappers that are launched in the job (recall one mapper is instantiated per input split).

InputSplit Class

```
public abstract class InputSplit {

   // returns number of bytes in the split
   public abstract long getLength() throws IOException;

   // returns array of nodes
   public abstract String{[] getLocations() throws IOException;

}
```

Implementations include:
- `FileSplit` (breaks each file into splits)
- `CombineFileSplit` (breaks multiple files into single split)

InputSplit is a logical representation of a subset of the data to be processed by an individual Mapper.

Note that InputSplit is a logical representation of a part of a file, and that split may not (most likely does not) contain an integral number records. In other words, the last record of an input split may be incomplete, as may be the first record of an input split. Processing whole records is the responsibiliity of the RecordReader.

The FileSplit represents a section of an input file of length getLength(). The FileSplit is returned by a call the InputFormat.getSplits(JobContext) and passed to InputFormat class to instantiate a RecordReader for the split in the context of the Mapper task.

The mapred package includes an implementation for CombineFileSplit. CombineFileSplit combines multiple files into a single split (default is that each file less than the split size is its own split).

RecordReader Interface

```
public interface RecordReader<K, V> {
    //Reads the next key/value pair from the input for processing.
    boolean next(K key, V value) throws IOException;
    //Creates an object of the appropriate type to be used as a key and value.
    K createKey();
    V createValue();
    //returns the current position in the input
    long getPos() throws IOException;
    //Close this InputSplit
    public void close() throws IOException;
    float getProgress() throws IOException;
}
```

Implementations include:

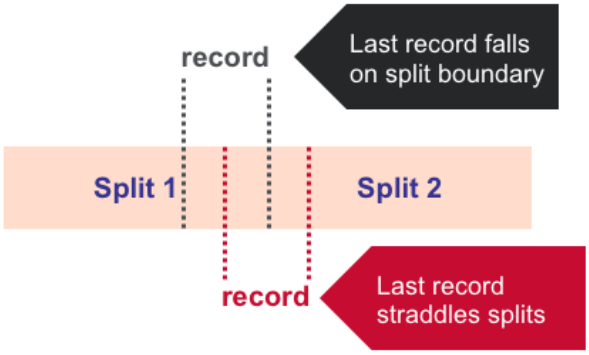
- LineRecordReader
- SequenceFileRecordReader

The record reader breaks up the data in an input split into key-value pairs and presents them to the Mapper assigned to that split.

Note that records won't necessarily start and end on whole split boundaries (or block boundaries for that matter). It is the job of the RecordReader to determine its position and process the record boundaries and presenting the tasks with keys and values.

LineRecordReader is used for Text files and SequenceFileRecordReader is used for Sequence files.

Identifying Record Boundaries



It is the most common case that a record does not end on a split boundary. Usually, a record starts in one split and is terminated in the next split. The record reader logic accounts for this. Users do not need to configure anything with default record readers, but custom record readers that a programmer implements must comprehend this phenomenon.

Knowledge Check

Knowledge Check

Which are true of the Mapper input flow?

A. Input files are split and key-value pairs generated for each split and sent to the same Mapper.

B. The map() method is called for each key-value pair once and output sent to the partitioner.

C. Results from the mapper are stored in the file system of the Control node

D. 1, 2 & 3

E. 1 & 2 only

Learning Goals

4.1 API overview

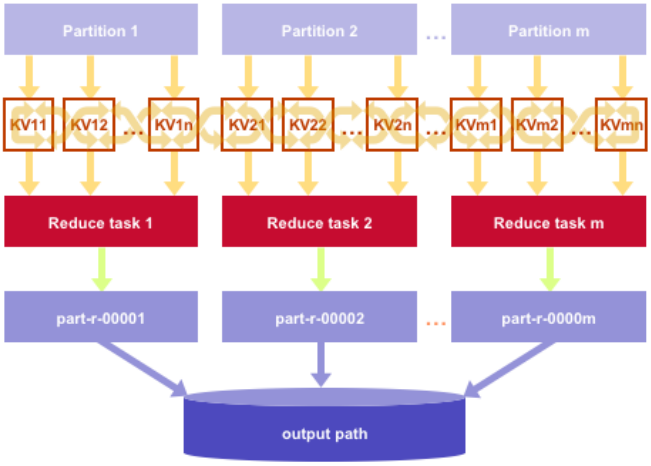
4.2 Examine

- Mapper input processing data flow
- **Reducer output processing data flow**

4.3 Explore the Mapper, Reducer and Job class API

In this section, we describe how reducer output data is processed in a MapReduce job.

Reducer Output Flow



The Hadoop framework is responsible for taking the partitions from each of the Mappers, shuffles and sorts the results, and presenting a sorted set of key-value pairs as single partition to each Reducer. The key (or a subset of the key) is used to derive the partition, typically by a hash function. The total number of partitions is the same as the number of reduce tasks for the job.

Each reducer takes its partition as input, processes one iterable list of key-value pairs at a time, and produces an output file called 'part-r-0000x' (where x denotes the number of the reducer). The output directory for this file is specified in the Job configuration and must reside on the HDFS/Mapr-FS file system. The directory cannot both exist and already contain reduce task output files, or the OutputFormat class will generate an IOException.

The RecordWriter (encapsulated in the OutputCommitter object) is responsible for writing the Reducer output to the file system.

OutputFormat Class

```
public abstract class OutputFormat<K, V> {
   public abstract RecordWriter<K, V>
getRecordWriter(TaskAttemptContext context) throws IOException,
InterruptedException;

   public abstract void checkOutputSpecs(JobContext context) throws
IOException, InterruptedException;

   public abstract OutputCommitter
getOutputCommitter(TaskAttemptContext context) throws IOException,
InterruptedException;
}
```

The OutputFormat class is analogous to the InputFormat class, but rather than checking the input files and directories and instantiating a record reader, the OutputFormat class checks the output directory and instantiates a record writer.

Additionally, the OutputFormat class has the responsibility of committing the output of the mapreduce job based on whether the job was successful or not. A job that does not run to 100% successful completion will only generate an _logs directory as output. All other partial reducer output will not be written.

RecordWriter Class

```
public abstract class RecordWriter<K, V> {

   public abstract void write(K key, V value) throws IOException,
InterruptedException;

   public abstract void close(TaskAttemptContext context) throws
IOException, InterruptedException;

}
```

RecordWriter writes the key-value pairs to the output files.

The TextOutputFormat.LineRecordWriter implementation irequires a java.io.DataOutputStream object to write the key-value pairs to the HDFS/MapR-FS file system. If the output is configured to be compressed to storage, then the LineRecordWriter will invoke the configured codec to encode the data prior to writing to the stream. The separator between the key and value is configurable – the default is the tab character.

Knowledge Check

Knowledge Check

Which are true of the Reducer output flow?

A. Each reducer takes its partition as input, processes one iterable list of key-value pairs at a time.

B. Produces an output file called 'part-r-0000x

C. The output directory for this file is specified in the Job configuration and must reside on the HDFS/Mapr-FS file system

D. 1, 2 & 3

E. 1 & 3 only

Learning Goals

Learning Goals

4.1 API overview

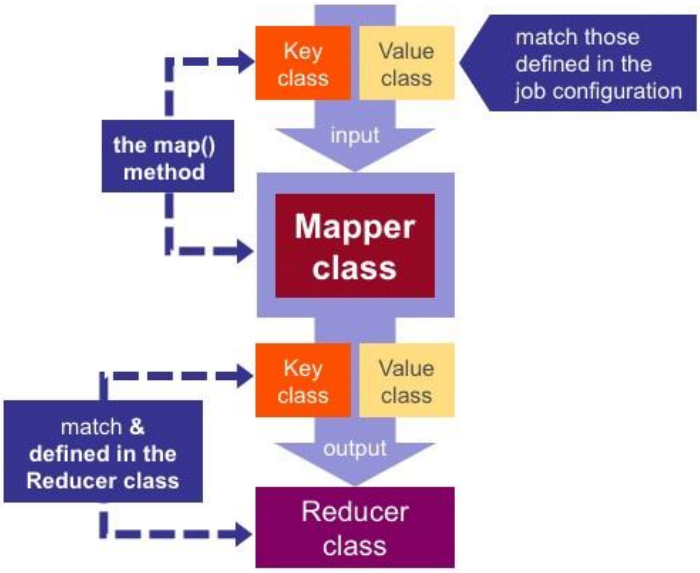
4.2 Examine

- Mapper input processing data flow
- Reducer output processing data flow

4.3 Explore the Mapper, Reducer and Job class API

In this section, we will now explore the Mapper, Reducer, and Job class API.

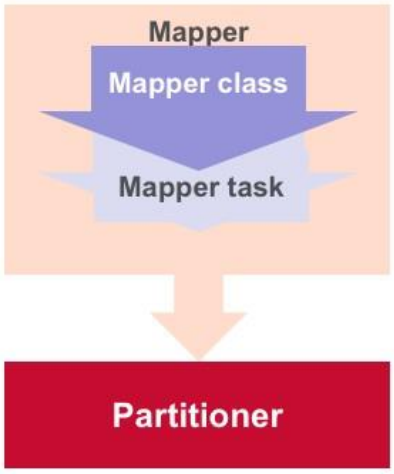
Input and Output Keys and Values

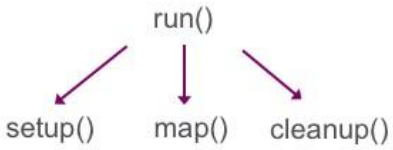


The Mapper class is parameterized with the key and value types for input and output. Here are a few rules regarding input and output keys and values for the Mapper class:

1. The input key class and input value class in the Mapper class must match those defined in the job configuration
2. The input key class and input value class in the map() method must match those defined in the Mapper class
3. The output key class and output value class in the Mapper must match the input key class
   and input value class defined in the Reducer class

Mapper Class





The default run() method for the Mapper class invokes setup() once for each Mapper/split. The run() method then calls the map() method once for each record. You can override the default run() method if you wanted to do some debugging, or if you implement a multi-threaded Mapper. The setup() method is called when the task starts (i.e. before the map() method is called).

The cleanup() method is called when the mapper task finishes (i.e. when all the records in the split have been processed on a mapper). One use case for using the setup() and cleanup() method is to open and close a file, HBase, or JDBC connection. This is a much more streamlined approach if I/O is performed during the task than to open and close a stream within the map() or reduce() method. Note however that the default run() method does not catch exceptions thrown in the map or reduce code.

The primary method you implement in the Mapper class is the map() method. All output

from the mapper class is sent to the partitioner.

Mapper Example

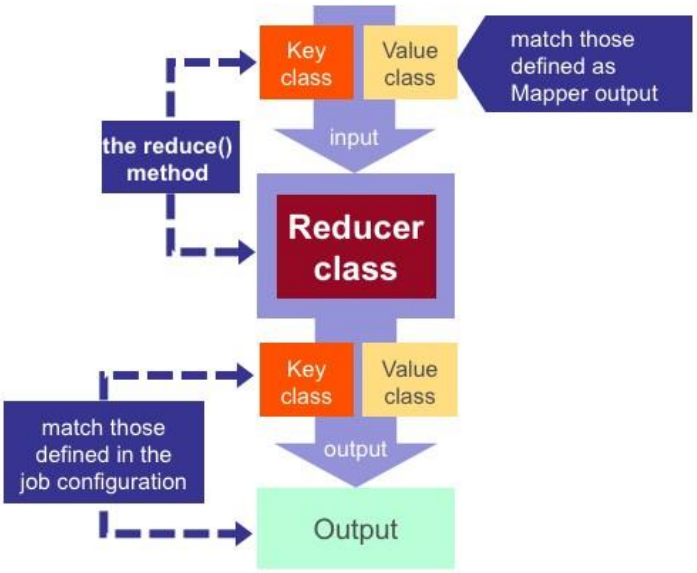
```
public  class MyWordcountMapper extends Mapper<LongWritable, Text,
Text, IntWritable> {
   private Text word = new Text();
   private final static IntWritable one = new IntWritable(1);
   public void map(LongWritable key, Text value, Context context)
throws IOException, InterruptedException {
      String line = value.toString();
      StringTokenizer iterator = new StringTokenizer(line);
      while (iterator.hasMoreTokens()) {
         word.set(iterator.nextToken());
         context.write(word, one);
      }
   }
}
```

This example Mapper iterates over a line and emits each word (as a key) and a “1” (as a value) as part of a word count application.

The LongWritable input key class in this case is provided as the byte offset into the input file to the beginning of the record (line). The input value and output key classes are both defined as Text, and the output type is defined as IntWritable. In this word count case, the output of the mapper will be the key (word) and value (cardinality of word in record).

The Mapper.Context object passed to the map() method contains, among other things, a reference to the job and its configuration. It also contains information specific to this (map) task.

Reducer Class



Now let's discuss the Reducer class. We see the class and method summaries for the Reducer class. The Reducer class is parameterized with the key and value types for input and output. Here are a few rules regarding input and output keys and values for the Reducer class:

1. The input key class and input value class in the Reducer must match the output key class and output value class defined in the Mapper class
2. The output key class and output value class in the Reducer must match those defined in the job configuration

The behavior of the cleanup(), run(), and setup() methods are identical as those described for the Mapper class.

Reducer Example

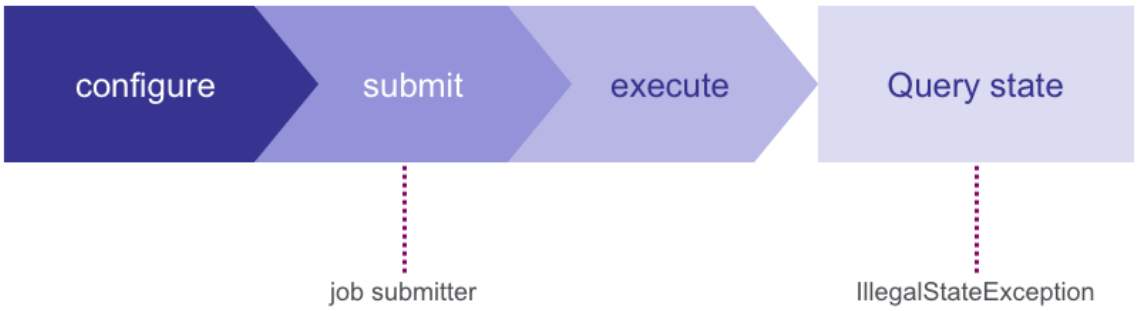
```
public class MyWordcountReducer extends
Reducer<Text,IntWritable,Text,IntWritable> {
   public void reduce(Text key, Iterable<IntWritable> values, Context
context) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable value : values) {
         sum += value.get();
      }
      context.write(key, new IntWritable(sum));
   }
}
```

This example Reducer iterates over a list of values for a given key. It emits each word (as a key) and a sum (as a value) as part of a word count application.

The Text input (key) class and Iterable<IntWritable> input value class match the output key class and output value class, respectively, from the Mapper.

The Reducer.Context object passed to the reduce() method contains, among other things, a reference to the job and its configuration. It also contains information specific to this (reduce) task.

Job Class



Now let' s review the job submitter's view of the Job. It allows the user to configure the job, submit it, control its execution, and query the state. The set methods only work until the job is submitted, afterwards they will throw an IllegalStateException.

Job Object

```
Configuration conf = new Configuration();
Job job = new Job(conf, "mywordcount");
```

The example here constructs a new Job object using the string name “mywordcount”. If you construct and job without specifying the name, a default name will be given to your job (which is the name of the driver class which instantiates the Job object).

Job Methods

void	**setGroupingComparatorClass**(Class<? extends RawComparator> cls) Define the comparator that controls which keys are grouped together for a single call to Reducer.reduce(Object, Iterable, org.apache.hadoop.mapreduce.Reducer.Context)
void	**setInputFormatClass**(Class<? extends InputFormat> cls) Set the InputFormat for the job.
void	**setJarByClass**(Class<?> cls) Set the Jar by finding where a given class came from.
void	**setJobName**(String name) Set the user-specified job name.
void	**setMapOutputKeyClass**(Class<?> theClass) Set the key class for the map output data.
void	**setMapOutputValueClass**(Class<?> theClass) Set the value class for the map output data.
void	**setMapperClass**(Class<? extends Mapper> cls) Set the Mapper for the job.
void	**setNumReduceTasks**(int tasks) Set the number of reduce tasks for the job.
void	**setOutputFormatClass**(Class<? extends OutputFormat> cls) Set the OutputFormat for the job.
void	**setOutputKeyClass**(Class<?> theClass) Set the key class for the job output data.
void	**setOutputValueClass**(Class<?> theClass) Set the value class for job outputs.

The setJarByClass() method defines the class for the driver.

The setOutputKeyClass() and setOutputValueClass() methods apply to both the mapper and reducer classes.

If your mapper class has different output key and value classes, use the setMapOutputKeyClass() and setMapOutputValueClass() methods.

Job Methods

void	**setPartitionerClass**(Class<? extends Partitioner> cls) Set the Partitioner for the job.
void	**setReducerClass**(Class<? extends Reducer> cls) Set the Reducer for the job.
void	**setSortComparatorClass**(Class<? extends RawComparator> cls) Define the comparator that controls how the keys are sorted before they are passed to the Reducer.
void	**setWorkingDirectory**(Path dir) Set the current working directory for the default file system.
void	**submit**() Submit the job to the cluster and return immediately.
boolean	**waitForCompletion**(boolean verbose) Submit the job to the cluster and wait for it to finish.

Besides the remaining setters, the submit() and waitForCompletion() methods are how the user launches the job from the driver. The difference is that submit() is non-blocking (background) while waitForCompletion() is blocking (foreground).

Simplistic Driver Implementation

```
public class MyWordcountDriver {
   public static void main(String[] args) throws Exception {
      Configuration conf = new Configuration();
      Job job = new Job(conf, "mywordcount");
      job.setJarByClass(MyWordcountDriver.class);
      job.setMapperClass(MyWordcountMapper.class);
      job.setReducerClass(MyWordcountReducer.class);
      job.setOutputKeyClass(Text.class);
      job.setOutputValueClass(IntWritable.class);
      job.setInputFormatClass(TextInputFormat.class);
      job.setOutputFormatClass(TextOutputFormat.class);
      FileInputFormat.addInputPath(job, new Path(args[0]));
      FileOutputFormat.setOutputPath(job, new Path(args[1]));
      System.exit(job.waitForCompletion(true) ? 0 : 1);
   }
}
```

The driver class contains a main method which instantiates a new job and job configuration. We set various components of the job configuration (e.g. name of driver, mapper, and reducer classes; type of input and output key, type of input and output value) and then call waitForCompletion() to launch the job and block until it returns.

The drawback of putting all the driver code in one big main method is that you the programmer must then parse any additional parameters that get passed to the job. Fortunately, this work has already be done for us, as we see momentarily.

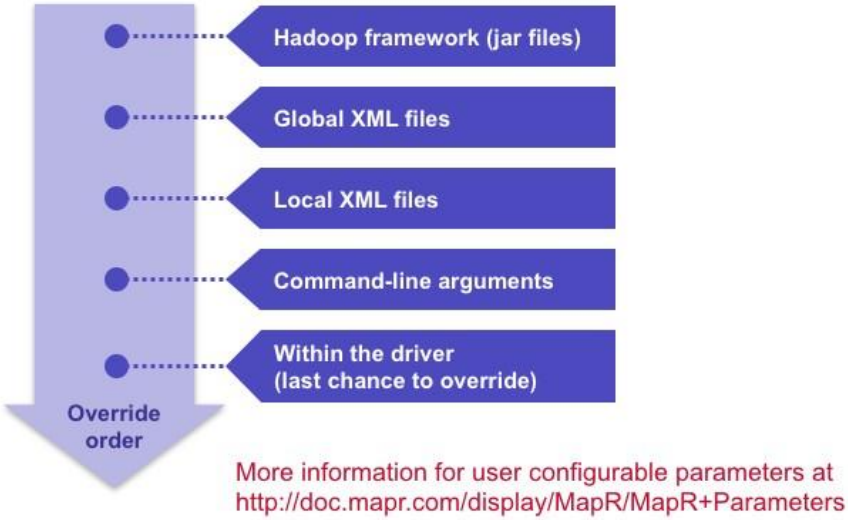
Best Way to Implement the Driver

```
public class MyWordcountDriver extends Configured implements Tool {
   public static void main(String[] args) throws Exception {
      Configuration conf = new Configuration();
      System.exit(ToolRunner.run(conf, new MyWordcountDriver (),
args));
   }

   public int run(String[] args) throws Exception {
      Job job = new Job(conf, "mywordcount");
      . . .
      job.waitForCompletion(true) ? 0 : 1;
}
```

JC-to-MC: show code in order. In this case, show the whole slide first (like you usually do), then show 1, then show both boxes numbered 2, then show the whole slide again (like you usually do).

Using ToolRunner allows you to make use of the GenericOptionsParser to pass Hadoop options as well as command-line arguments to your driver. This imposes a trivial modification to the previous driver implementation (by using a run method and main method that calls run), but it gives us a huge payoff. Now we can pass configuration parameters to our mapreduce jobs in XML files or directly at the command line.

How to Set Configuration Parameters



There are many configuration parameters. Some are related to the file system. Some are related to Mapreduce. Some may be custom to your application.

The slide above shows the order in which the parameters are defined and overwritten. As seen, the first place that the parameters get defined is in the hadoop framework itself inside the jar files. The last place that you can define the configuration parameters before the job launch is within the driver itself.

Lab 4.3: Write a MapReduce Program

Next Steps

Manage and Test Hadoop MapReduce Applications

Lesson 5: Managing, Monitoring, and Testing MapReduce Jobs

Congratulations! You have completed Lesson Four. You should now be able to Identify the correct MapReduce API for your environment.

Understand how Writable types require conversion in order to manipulate data.

You should know that each mapper is assigned a single complete input split, and each map method processes a single record and each reducer is assigned one or more complete partitions, and each reducer method processes all records from those partitions.

You should be able to use the ToolRunner interface for your driver code.

In the next lesson you will learn how to manage, monitor and test MapReduce jobs.

Manage and Test Hadoop MapReduce Applications

Lesson 5: Managing, Monitoring, and Testing MapReduce Jobs

Manage and Test Hadoop MapReduce Applications,
Lesson 5: Managing, Monitoring, and Testing MapReduce Jobs.

Learning Goals

Learning Goals

5.1 Work with counters

5.2 Use the MCS to monitor jobs

5.3 Manage and display jobs, history, and logs

5.4 Write unit tests for MapReduce programs

Welcome to Lesson 5. 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.

Learning Goals

5.1 Work with counters

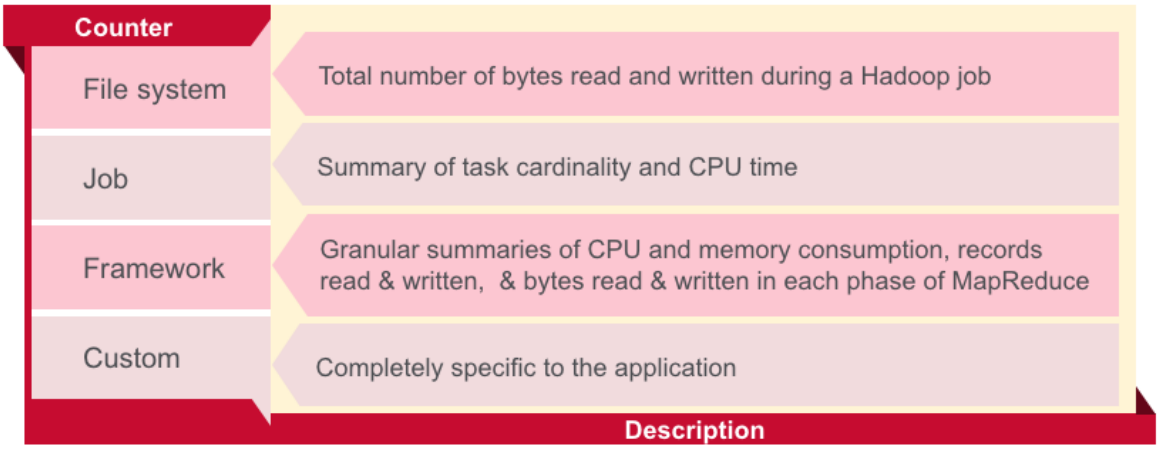
5.2 Use the MCS to monitor jobs

5.3 Manage and display jobs, history, and logs

5.4 Write unit tests for MapReduce programs

Let's begin by working with counters.

Summary of Counters



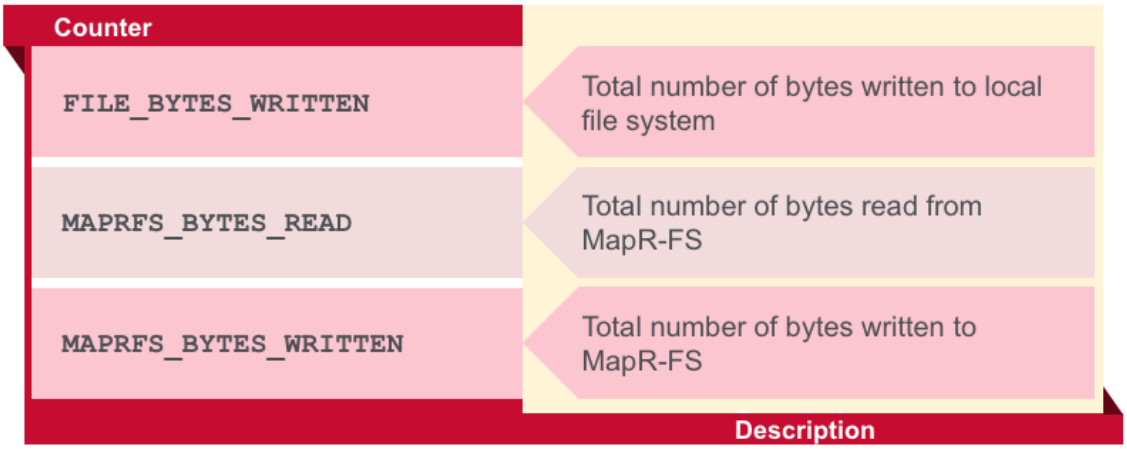
There are 4 categories of counters in Hadoop: file system, job, framework, and custom.

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.

File System Counters



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.

Job Counters

Counter	Description
DATA_LOCAL_MAPS	Total number of map tasks executed on local data
FALLOW_SLOTS_MILLIS_MAPS	Total time map tasks spend waiting after slots are reserved (pre-emption)
FALLOW_SLOTS_MILLIS_REDUCES	Total time reduce tasks spend waiting after slots are reserved (pre-emption)
SLOTS_MILLIS_MAPS	Total time map tasks spend executing
SLOTS_MILLIS_REDUCES	Total time reduce tasks spend executing
TOTAL_LAUNCHED_MAPS	Total number of map tasks launched, including failed tasks
TOTAL_LAUNCHED_REDUCES	Total number of reduce tasks launched, including failed tasks

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

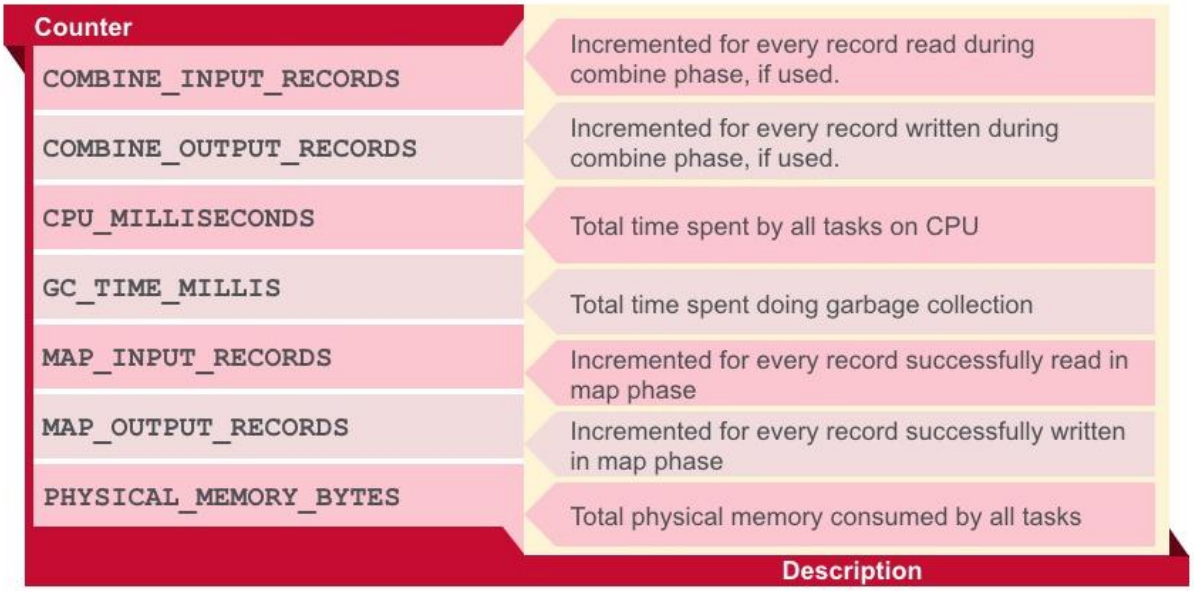
The DATA_LOCAL_MAPS indicates how many map tasks executed on local file systems. Optimally, all the map tasks will execute on local data to exploit locality of reference, but this isn't always possible.

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 clock time for the map and reduce tasks.

The TOTAL_LAUNCHED_MAPS counter defines how many map tasks were launched for the job, including failed tasks. Optimally, this number is the same as the number of splits for the job.

Framework Counters (1)



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.

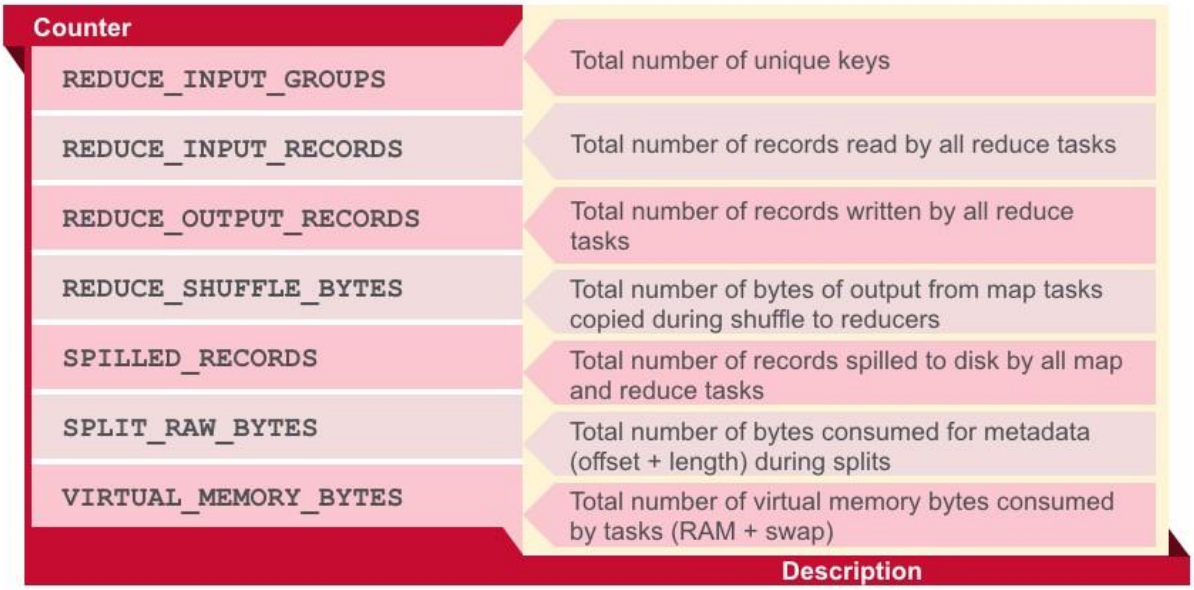
The CPU statistics are gathered from /proc/cpuinfo and indicate how much total time was spent executing map and reduce tasks for a job.

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 /proc/meminfo and indicate how much RAM (not including swap space) was consumed by all the tasks.

Framework Counters (2)



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.

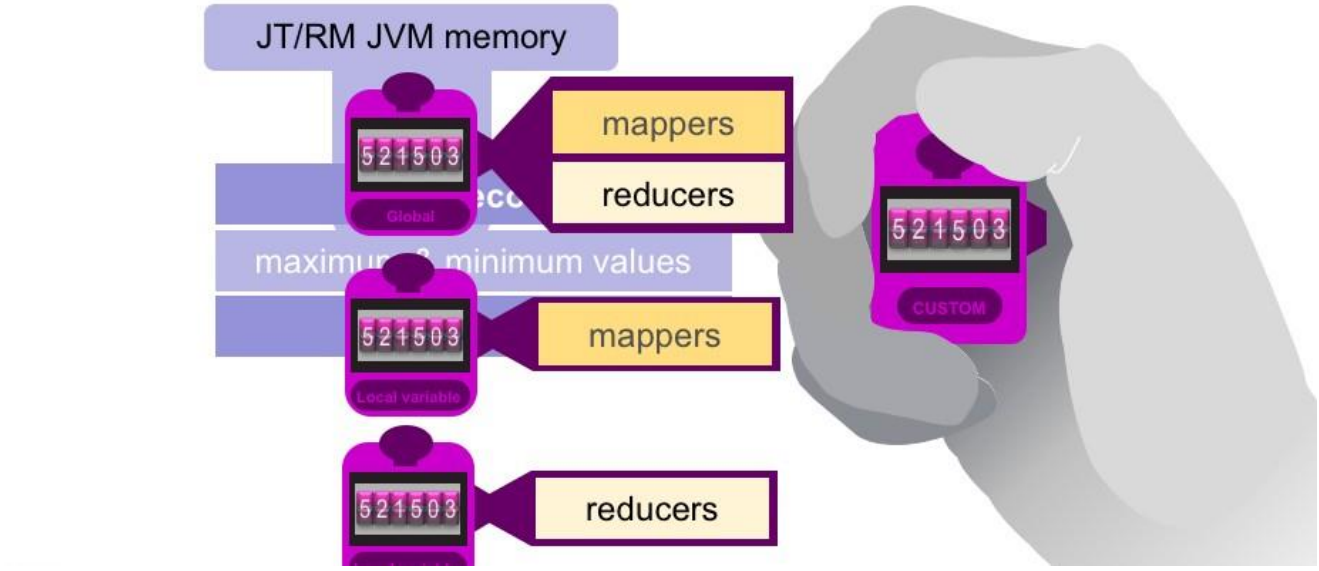
The REDUCE_SHUFFLE_BYTES counter indicates how many bytes the intermediate results from the mappers were transferred to the reducers. Higher numbers here will make the job go slower as the shuffle process is the primary network consumer in the MapReduce process.

The SPILLED_RECORDS counter indicates how much data the map and reduce tasks wrote (spilled) to disk when processing the records. Higher numbers here will make the job go slower.

The SPLIT_RAW_BYTES counter only increments for metadata in splits, not the actual data itself.

The VIRTUAL_MEMORY_BYTES counter shows how much physical memory plus swap space is consumed by all tasks in a job. Contrast this counter with PHYSICAL_MEMORY_BYTES. The difference between these two counters indicates how much swap space is used for a job.

Custom Counters



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 capacity.


Knowledge Check

Knowledge Check

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

A. The correct number of bytes were read and written

B. The correct number of tasks were launched and successfully ran

C. The amount of CPU and memory consumed is appropriate for your job and cluster nodes

D. The correct number of records were read and written

E. All of the above

Lab 5.1b: Use Custom Counters

Open your lab guide to perform lab 5.1a.

Lab 5.1a: Examine Default Job Output

Open your lab guide to perform lab 5.1b.

Learning Goals

Learning Goals

5.1 Work with counters

5.2 Use the MCS to monitor jobs

5.3 Manage and display jobs, history, and logs

5.4 Write unit tests for MapReduce programs

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

Displaying Jobs in MCS

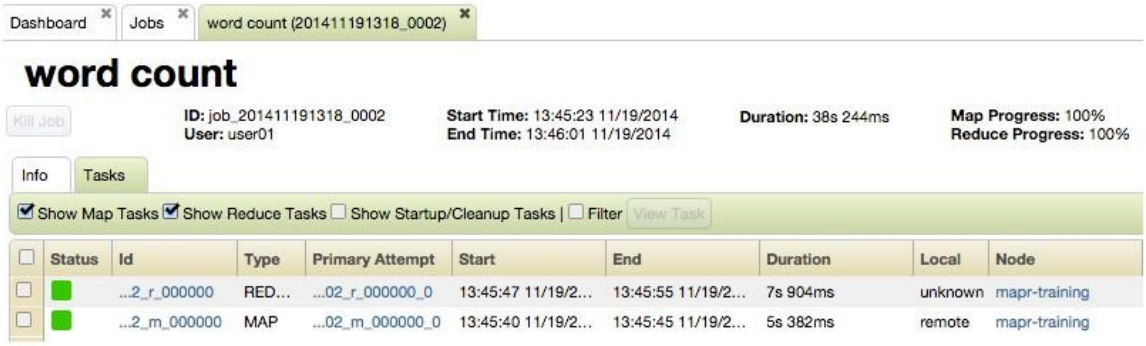
https://webserver:8443

Let's discuss using the MapR Console Server (MCS) to manage MapReduce jobs and tasks.

In addition to managing and monitoring your MapR cluster, you can use the MCS to show granular job and task information. You must configure the metrics database in order to display this information in the MCS. The first time you log in to the MCS as your user, you'll need to specify the URL for the metrics database (database-server:3306), username, password, and name of database (metrics).

The screen shot above shows the total number of jobs reported in the metrics database for this cluster (there's just one called "wordcount"). You can "click into" the details of any job run on the system.

Getting Job Summaries in MCS



The screen shot above shows summary information about this job, including the number of map and reduce tasks and how long they each took to run. You also see the status of each task. In this case, green means good.

Getting Task Summaries in MCS

You can dig into the tasks of a job by clicking on them in the GUI. The screen shot above shows more details around the reduce task for our “wordcount” job. The summaries show when certain parts of the task started and stopped, and what their status is.

Getting Task Details in MCS

Dashboard | Jobs | word count (201411191318_0002) | task_201411191318_0002_r_000000 | attempt_201411191318_0002_r_000000_0

attempt_201411191318_0002_r_000000_0

Start Time: 13:45:47 11/19/2014
End Time: 13:45:55 11/19/2014 **Duration:** 7s 614ms

Info

Task Attempt: 100% complete

Shuffle End	Sort End	GC Time	CPU Time	Physical Mem	Virtual Mem	Host	Log Location	Diagnostic Info
13:45:54 11/19/2014	13:45:54 11/19/2014	29ms	270ms	131.5MB	3.6GB		Log »	

MapReduce Framework Counters

Counter	Total
Combine Input Records	-
Combine Output Records	-
Map Input Bytes	-
Map Input Records	-
Map Output Bytes	-
Map Output Records	-
Map Skipped Records	-
Reduce Input Groups	36
Reduce Input Records	36
Reduce Output Records	36
Reduce Shuffle Bytes	1.3KB
Reduce Skipped Records	-
Spilled Records	36
Split Raw Bytes	-

MapReduce Throughput Counters

Counter	Map	Reduce
Input Bytes/Sec	-	-
Input Records/Sec	-	5
Output Bytes/Sec	-	-
Output Records/Sec	-	5
Shuffle Bytes/Sec	180B	-

File System Counters

Counter	Total
Local Bytes Read	-
Local Bytes Written	146,370
MapR-FS Bytes Read	2.7KB
MapR-FS Bytes Written	2.5KB

hnologies L5-20

You can retrieve very low-level, granular details for any task. The screen shot above shows the framework counters, throughput counters, and file system counters for this particular task. This information is useful for both debugging and analyzing the runtime profile of your mapreduce job.

Getting Log Files From MCS

Task Logs: 'attempt_201411191318_0002_r_000000_0'

stdout logs

stderr logs

syslog logs

```
2014-11-19 13:45:50,937 INFO mapred.Child [main]: JVM: jvm_201411191318_0002_r_1629066112 pid: 21822
2014-11-19 13:45:51,961 WARN mapreduce.Counters [IPC Client (1237272991) connection to /127.0.0.1:58375 from job_201411191318_0002]: Group org.apache.hadoop.mapr
2014-11-19 13:45:52,238 INFO Configuration.deprecation [main]: slave.host.name is deprecated. Instead, use dfs.datanode.hostname
2014-11-19 13:45:52,359 INFO Configuration.deprecation [main]: topology.script.number.args is deprecated. Instead, use net.topology.script.number.args
2014-11-19 13:45:53,130 INFO mapred.Child [main]: Starting task attempt_201411191318_0002_r_000000_0
2014-11-19 13:45:53,130 INFO Configuration.deprecation [main]: session.id is deprecated. Instead, use dfs.metrics.session-id
2014-11-19 13:45:53,135 INFO jvm.JvmMetrics [main]: Initializing JVM Metrics with processName=SHUFFLE, sessionId=
2014-11-19 13:45:53,362 INFO util.ProcessTree [main]: setsid exited with exit code 0
2014-11-19 13:45:53,375 WARN util.ProcfsBasedProcessTree [main]: /proc/<pid>/status does not have information about swap space used(VmSwap). Can not track swap u
2014-11-19 13:45:53,375 INFO mapred.Task [main]:  Using ResourceCalculatorPlugin : org.apache.hadoop.mapreduce.util.LinuxResourceCalculatorPlugin@12cf66f
2014-11-19 13:45:54,192 INFO mapred.Merger [main]: Merging 1 sorted segments
2014-11-19 13:45:54,193 INFO mapred.Merger [main]: Down to the last merge-pass, with 1 segments left of total size: 1362 bytes
2014-11-19 13:45:54,224 INFO mapred.Merger [main]: Merging 1 sorted segments
2014-11-19 13:45:54,225 INFO mapred.Merger [main]: Down to the last merge-pass, with 1 segments left of total size: 1362 bytes
2014-11-19 13:45:54,265 INFO mapred.Task [main]: Task:attempt_201411191318_0002_r_000000_0 is done. And is in the process of commiting
2014-11-19 13:45:54,270 WARN mapreduce.Counters [main]: Group FileSystemCounters is deprecated. Use org.apache.hadoop.mapreduce.FileSystemCounter instead
2014-11-19 13:45:55,329 INFO mapred.Task [main]: Task attempt_201411191318_0002_r_000000_0 is allowed to commit now
2014-11-19 13:45:55,336 INFO output.FileOutputCommitter [main]: Saved output of task 'attempt_201411191318_0002_r_000000_0' to /tmp/myout2
2014-11-19 13:45:55,378 INFO mapred.Task [main]: Task 'attempt_201411191318_0002_r_000000_0' done.
```

profile.out logs

You can display the log file associated with a given task by clicking the log link in the previous screen. The details in the log file include the following:

- Standard out generated from this task
- Standard error generated from this task
- Syslog log file entries generated by this task
- Profile output generated by this task
- Debug script output generated by this task

Note that debug scripts are optional and must be configured to run.

Learning Goals

Learning Goals

5.1 Work with counters

5.2 Use the MCS to monitor jobs

5.3 Manage and display jobs, history, and logs

5.4 Write unit tests for MapReduce programs

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

Tracking Status of Launched Jobs (1)

```
$ hadoop job -list
1 jobs currently running
JobId     State     StartTime UserName  Priority  SchedulingInfo
job_201404051135_0322    1    1396917391960  user20    NORMAL    NA

$ hadoop job -status job_201404051135_0322
Job: job_201404051135_0322
file:
maprfs:/var/mapr/cluster/mapred/jobTracker/staging/user20/.staging/job_201404051135_
0322/job.xml
tracking URL: http://ip-10-170-165-
141:50030/jobdetails.jsp?jobid=job_201404051135_0322
map() completion: 0.024717983
reduce() completion: 0.0
Counters: 19
<output omitted>
```

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.

Tracking Status of a Launched Job (2)

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.

Stopping a Launched Job

```
$ hadoop job -list
1 jobs currently running
JobId    State    StartTime    UserName Priority SchedulingInfo
job_201404051135_03221    1396917391960    user20   NORMAL   NA

$ hadoop job -kill job_201404051135_03221
```

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.

Modifying a Job Priority

When job is launched using API

```
. . .
   Configuration conf = new Configuration();
   conf.set("mapred.job.priority", "VERY_HIGH");
   Job job = new Job(conf, "receipts" + System.getProperty("user.name"));
. . .
```

When job is launched using a configuration file (or passing –D mapred.job.priority=X at job submission

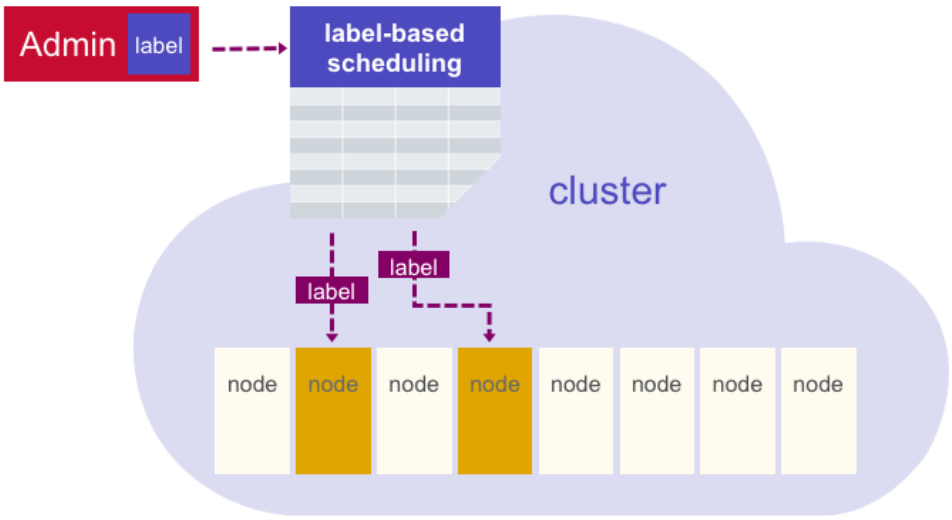
```
<configuration>
<property>
<name>mapred.job.priority</name><value>HIGH</value>
</property>
</configuration>
```

After job is submitted (but before it is run) using hadoop CLI

```
$ hadoop job -set-priority job_201311221648_0039 VERY_LOW
13/11/26 14:59:12 INFO fs.JobTrackerWatcher: Current running JobTracker is: ip-10-198-
41-188/10.198.41.188:9001
Changed job priority.
```

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 Feature: Label-Based Scheduling

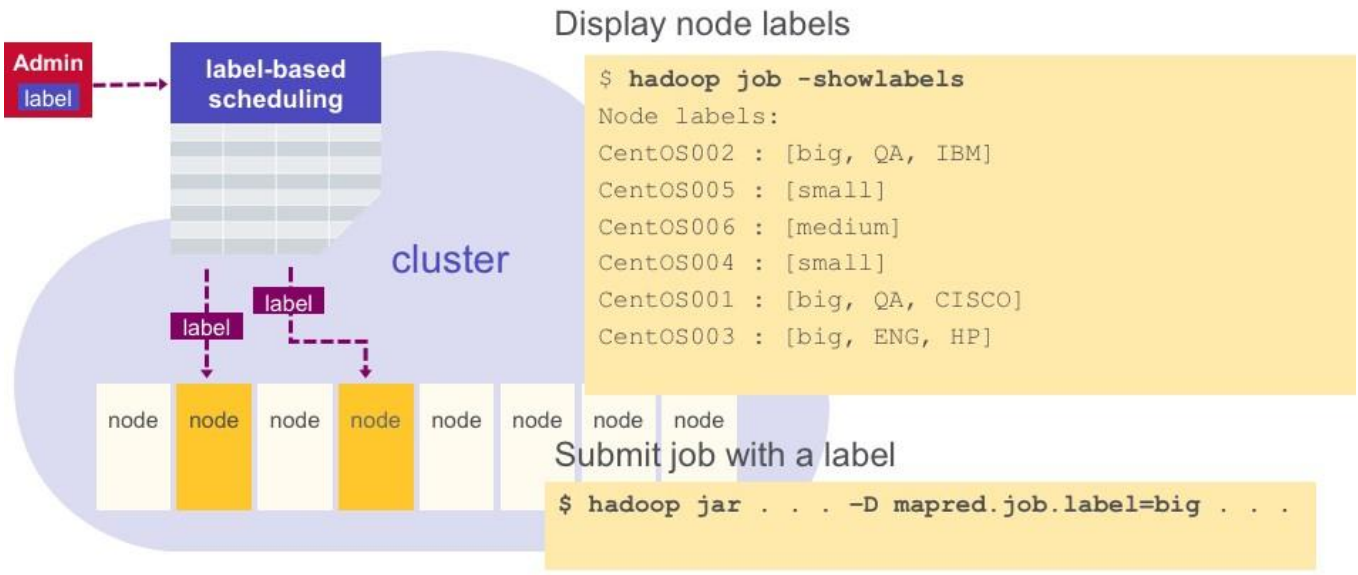


MapR has a feature called "label-based scheduling". This feature enables you to override the default scheduling algorithm and launch job tasks on specific nodes. This allows Admins and End Users more control over where jobs run on the cluster.

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.

Using Label-Based Scheduling



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.

Apache Commons Logging (JCL)

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 more information.

Logging Information

```
. . .
private static Log log = LogFactory.getLog(MyClass.class);
. . .
public void map( . . .) {
   if(debugcondition) {
      log.debug("debug log message");
      // OR
      System.out.println("debug output");
   else if (errorcondition) {
      log.error("error log message");
      // OR
      System.err.println("error output");
   }
. . .
}
```

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).

Viewing the History of a Job

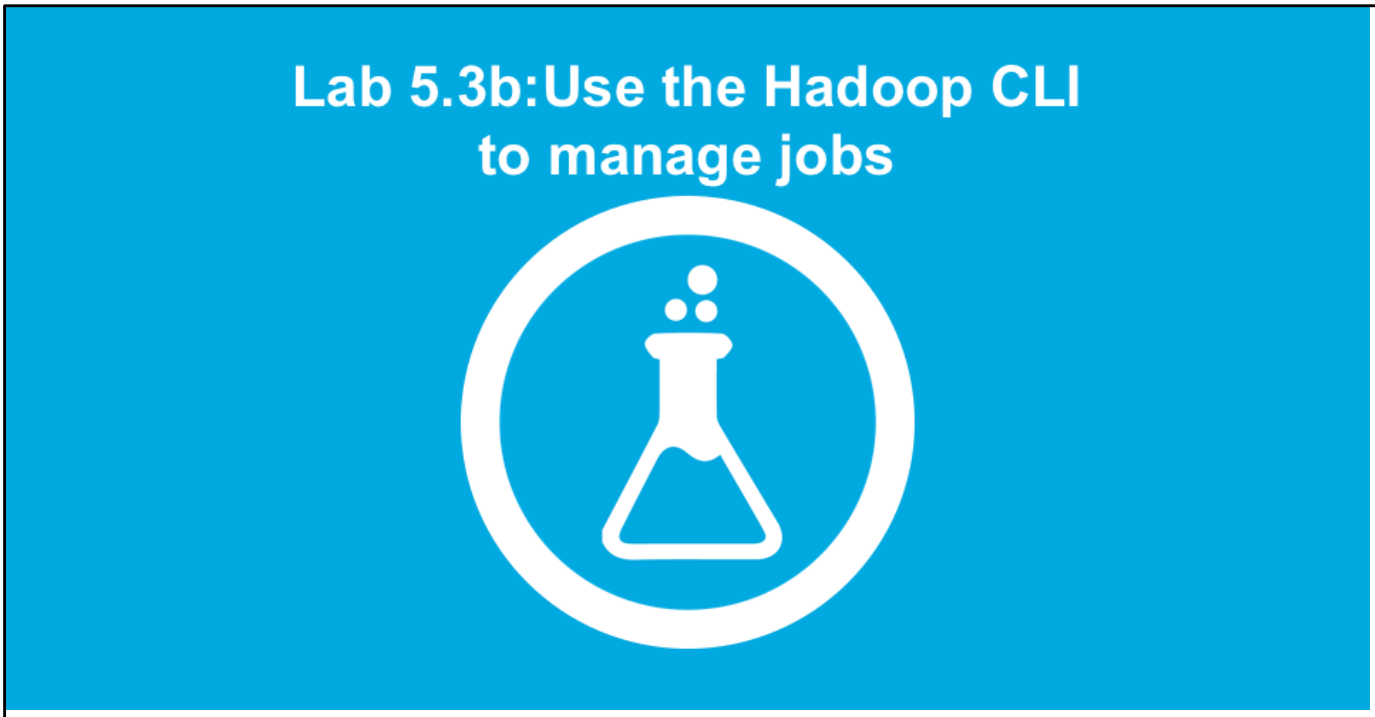
```
$ hadoop job -history TERASORT.OUT/
Hadoop job: job_201311211623_0015
. . .
Status: SUCCESS
. . .
|Job Counters                  |Aggregate execution time of mappers(ms)|0
|0         |28,978
. . .
Task Summary
Setup    1    1        0    0    23-Nov-2013 15:08:25   23-Nov-2013 15:08:26
(1sec)
. . .
Analysis
Time taken by best performing map task task_201311211623_0015_m_000001:
12sec
. . .
```

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.

Lab 5.3a: Use standard output, error, and logging

Refer to your lab guide to complete:

1. Lab 5.3a: Write MapReduce code to use standard output, error, and logging
2. Lab 5.3b: Use the Hadoop CLI to manage jobs



Refer to your lab guide to complete:

1. Lab 5.3a: Write MapReduce code to use standard output, error, and logging
2. Lab 5.3b: Use the Hadoop CLI to manage jobs

Learning Goals

Learning Goals

5.1 Work with counters

5.2 Use the MCS to monitor jobs

5.3 Manage and display jobs, history, and logs

5.4 Write unit tests for MapReduce programs

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

Summary of MRUnit

Testing harness for MapReduce based on `JUnit`

Developed at Cloudera (http://mrunit.apache.org)

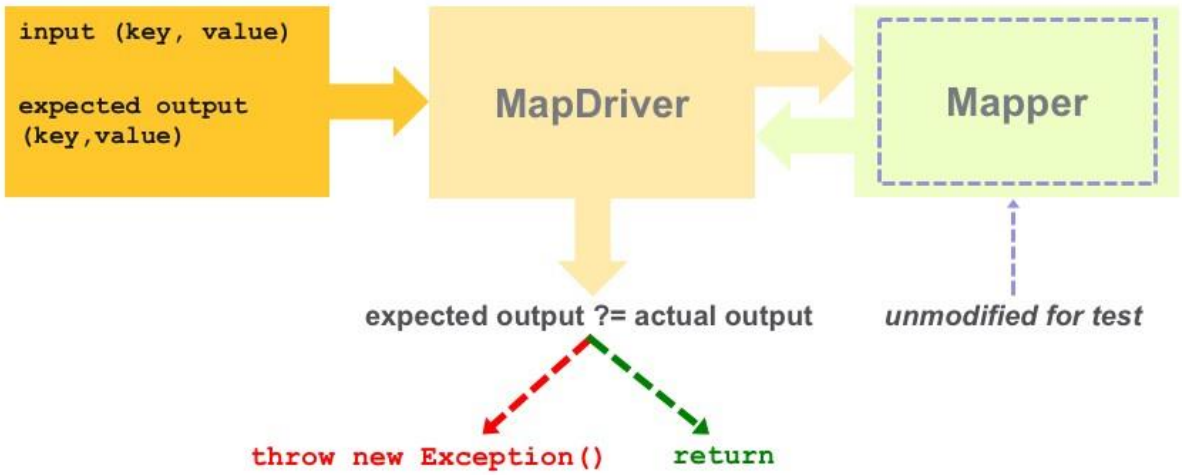
Uses `LocalJobRunner` to execute code

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 output and logging information for your debugging purposes if you like.

Conceptual View of `MRUnit` Map Test



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.

Example MRUnit Code: Map-only (1)

```
public class ReceiptsTest {
   private static MapDriver<LongWritable, Text, Text, Text>
mapDriver;

   @Before
   private static void setUp() {
      Receipts.ReceiptsMapper mapper = new
Receipts.ReceiptsMapper();
      mapDriver = MapDriver.newMapDriver(mapper);
   }
```

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.

Example MRUnit Code:Map-only (2)

```
  @Test
   private static void testMapper() throws IOException,
InterruptedException {
      final LongWritable key = new LongWritable(0);
      Text value = new Text("1901 588 525 63 588 525 63 ..........
.......... ..........");
      mapDriver
         .withInput(key, value)
         .withOutput(new Text("summary"), new Text ("1901_63"))
          .runTest();
      }
```

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”.

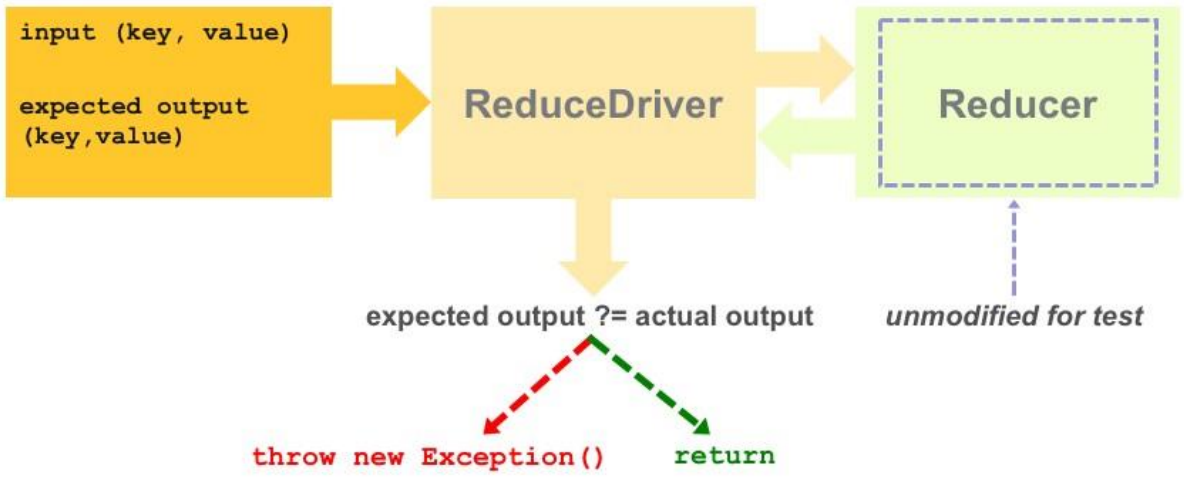
The runTest() method calls the MRUnit test harness to execute the job locally. This method will throw an exception if the test fails (for any reason).

Example MRUnit Code:map-only (3)

```
   public static void main(String[] args) {
      setUp();
      try { testMapper();}
      catch (Exception e) {System.err.println("exception: " +
e.toString());}
      return;
   }
}
```

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.

Conceptual View of MRUnit Reducer Test



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.

5.4: Use MRUnit to test a MapReduce Application

Complete lab 5.4: Use MRUnit to test a MapReduce Application.

Next Steps

Mange and Test Hadoop MapReduce Applications

Lesson 6: Managing Performance

Congratulations! You have completed Lesson Five. You should now know how Hadoop tracks counters for the file system, jobs, framework, and custom counters that you create. That the MCS can be used as a single pane of glass to manage your MapReduce jobs. How Hadoop commands can launch, manage, and monitor jobs. You should also now know that you can test your map and reduce functionality using MRUnit. In the next lesson you will learn strategies of enhancing performance.

Manage and Test Hadoop MapReduce Applications

Lesson 6: Managing Performance

Welcome to DEV 301 – Manage and Test Hadoop MapReduce Applications, Lesson 6: Managing Performance.

Learning Goals

Learning Goals

6.1 Review components of MapReduce performance

6.2 Enhance performance in MapReduce jobs

6.3 Describe MapR performance enhancements

Welcome to Lesson Six. In this lesson we will look at strategies of enhancing performance.

Learning Goals

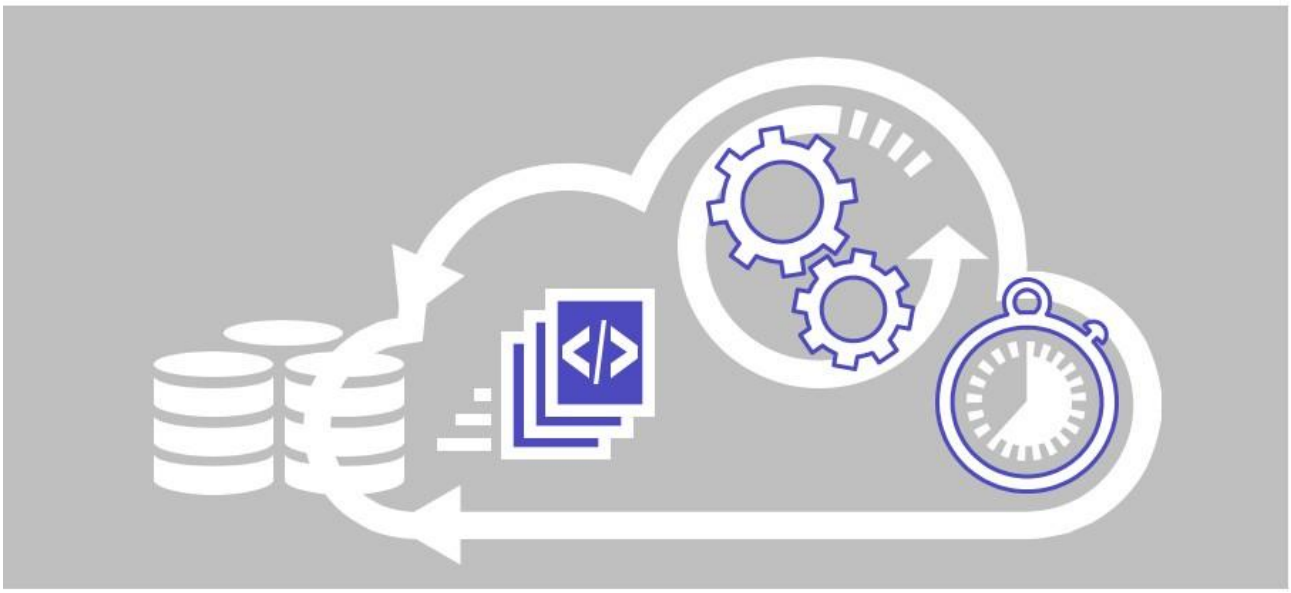
6.1 Review components of MapReduce performance

6.2 Enhance performance in MapReduce jobs

6.3 Describe MapR performance enhancements

Let’s begin by reviewing the components that contribute to MapReduce job performance.

Performance Tuning Tips



Let’s begin by discussing strategies for enhancing performance. When tuning performance, there are a few things to keep in mind.

Performance is related to either space or time. Space – how many bytes are read and written to memory, disk, and network. Time – how long does a task take, a job take, a phase of a task take, … etc.

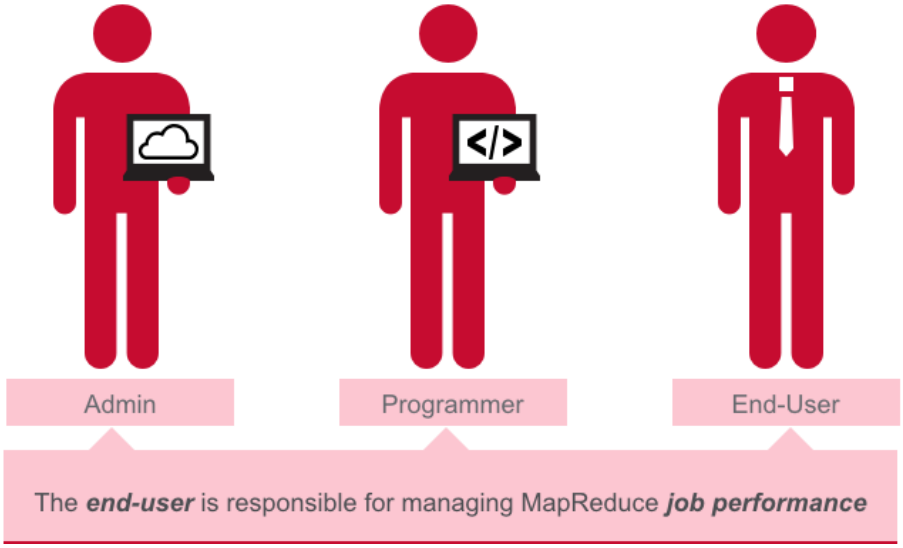
First and foremost, you must ensure that your code is bug free before you optimize it. By changing the nature of your code to improve performance, you may well introduce bugs, but also, you may introduce complexity which makes it more difficult to debug in the first place.

When tuning for performance, you should make the common case fast. Don't waste time on parts of your code that don't execute very often because the pay-off is small.

Performance tuning is very specific to an application and its workload. Making a change in configuration or the code may improve performance for one job but actually worsen performance for another.

You should create a set of scripts that automate your configuration and testing. In the best case, you can launch a series of tests and compare performance results without having to do it manually.

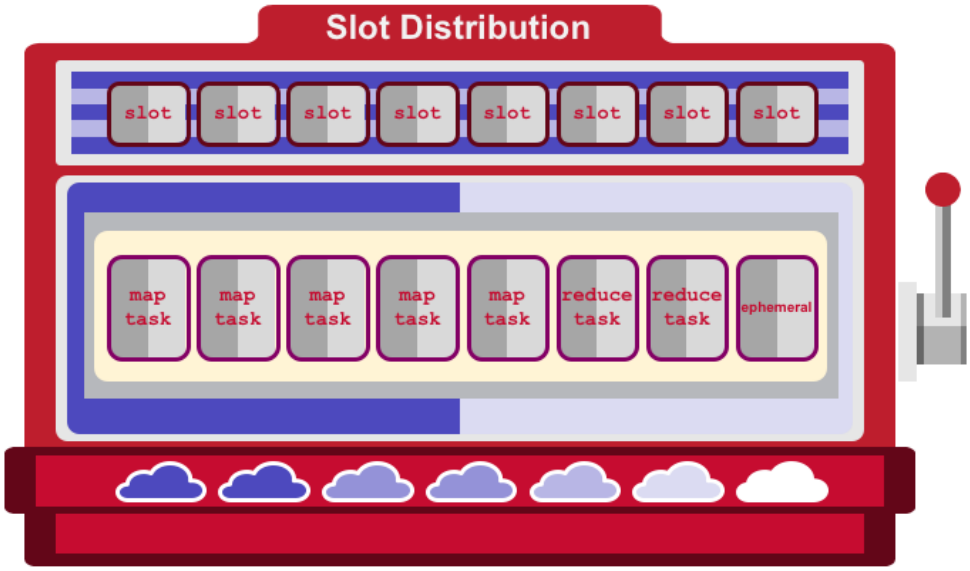
Breakdown of Runtime Performance



This lesson focuses on managing the performance of a single MapReduce job/application, not on the underlying cluster framework. There are several factors that go into the management and tuning of cluster performance that are beyond the scope and intent of this course.

Click on each to learn more.

Map and Reduce Slots



Slot configuration is calculated by CPU, memory, and disk. In this example, the term “slot” is used to represent a schedulable unit of task execution, and one slot can run only one task at a time. The maximum number of slots available on a given node is a function of the number of CPU cores by default (but your admin can override that default configuration). Some slots are reserved for map tasks and other slots are reserved for reduce tasks. Typically, reduce tasks consume more compute resources, so the total number of reduce slots should be lower than the total number of map slots.

In MapR, using the ExpressLane feature means that each task tracker will reserve one or more “ephemeral” slots for executing “small jobs.”

Knowledge Check

Knowledge Check

Slot configuration is calculated by:

A. CPU

B. Memory

C. Disk

D. All of the above

Learning Goals

Learning Goals

6.1 Review components of MapReduce performance

6.2 Enhance performance in MapReduce jobs

6.3 Describe MapR performance enhancements

In this lesson we will look at strategies of enhancing performance.

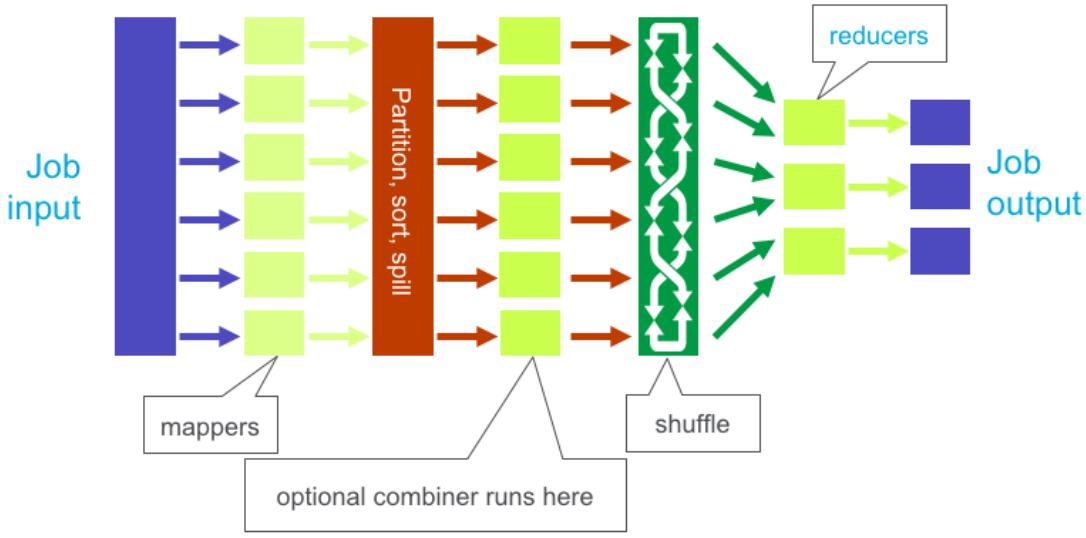
Performance Enhancements

Using a custom combiner
Compressing map results
Modifying number of reduce tasks
Using speculative execution
Reusing a JVM
Configuring sort properties
Configuring Java properties

Let's talk about what kind of things can be used to enhance performance. We will discuss some of the following enhancements:

- Using a custom combiner
- Compressing map results
- Modifying number of reduce tasks
- Using speculative execution
- Reusing a JVM
- Configuring sort properties
- Configuring Java properties

Custom Combiners



A custom combiner may be implemented and used to reduce the amount of network traffic during the shuffle phase. A combiner is called after all the mappers finish and before the results are shuffled and transmitted over the network. The primary rule to consider when implementing a combiner is that the output of the combiner must match the input of the reducer. If the reducer is itself commutative and associate, then you might be able to use the reducer as the combiner. If not both associative and commutative, then you need to write a custom combiner in addition to your map and reduce code.

Note that using a combiner does not always improve performance. Specifically, if your Hadoop job is more CPU bound than I/O bound. In other words, if it uses lots of computation over a relatively small set of data, then using a combiner may actually increase the amount of execution time with little savings on the network bandwidth.

As with all performance tuning, you'll need to vet out the performance impact of using a combiner with your specific application workload. The general recommendation is to use a combiner when performing aggregations if the number of keys is significantly less than the number of records. This suggests that there is a lot to do in the reduce phase that can be done in the map phase before the shuffle.

Note that the combiner is invoked on each mapper after the intermediate results have been partitioned and sorted. Also note that the framework may call the same combiner multiple times on the same mapper, as shown above. In general, the framework calls the combiner once for every buffer flush. We discuss buffer flushes later in this lesson.

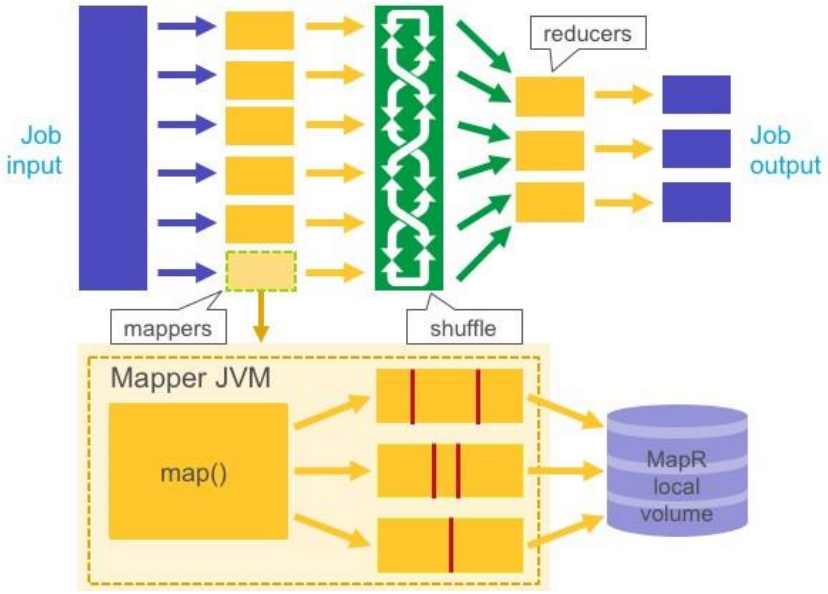
Example: Using a Combiner

```
public class UniversityDriver extends Configured implements Tool {

   public int run(String[] args) throws Exception {
      job.setCombinerClass(MyCombiner.class);
. . .
```

```
public class MyReducer  extends Reducer
<Text,IntWritable,Text,FloatWritable> {

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

   }
}
```

This code snippets shows how to define a custom combiner. The driver class must configure the job to use the combiner by invoking `setCombinerClass()` method of the `Configuration` object. The combiner class must implement the `reduce()` method. Again, recall the input to the combiner must match the output from the mapper, and the output of the combiner must match the input to the reducer.

Map Output Compression

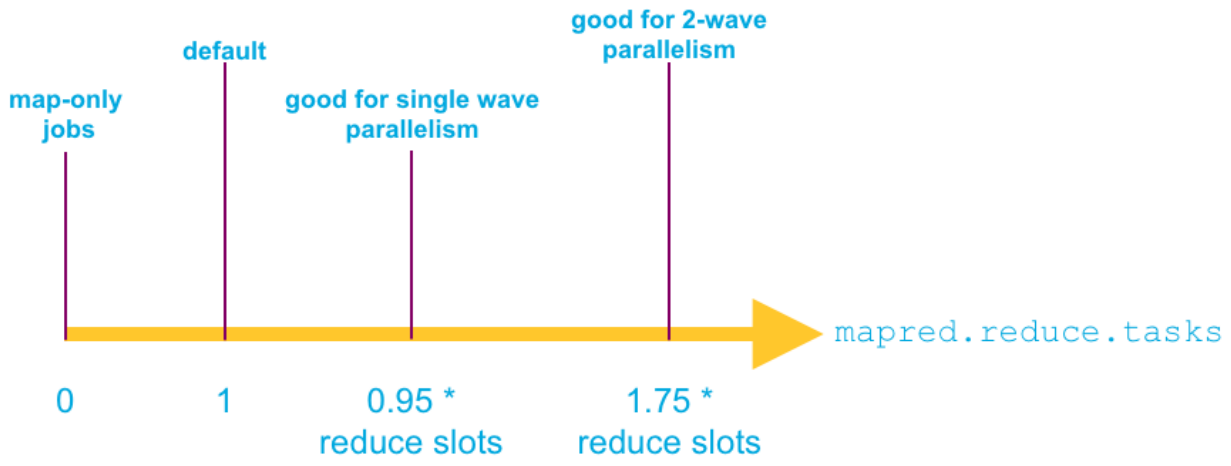


Enabling Map output compression reduces the disk and network I/O overhead at the expense of CPU cycles necessary for compression and decompression. In Hadoop, this data is written to the local disk of the mapper, not the distributed file system. MapR enhances this by writing to a local MapR volume on the mapper. We will discuss the MapR local volume in more detail later in this lesson.

The tunables for compression include turning it on or off, and the actual codec to use if enabled. The default compression algorithm used on MapR is LZ4. The ZLIB codec provides the highest level of compression but also requires the most CPU cycles.

In general, it is recommended to enable compression for intermediate results when using non- binary data.
The data that is compressed in this context is that which is spilled to storage during the sort phase on the mappers. ***Enable compression for non-binary data and configure LD_LIBRARY_PATH to enable native codecs***

Tuning Number of Reduce Tasks



You can request a specific number of reduce tasks by configuring the `mapred.reduce.tasks` parameter. This has the effect of partitioning the data into that number of partitions during the map phase and scheduling that same number of reduce tasks for each partition.

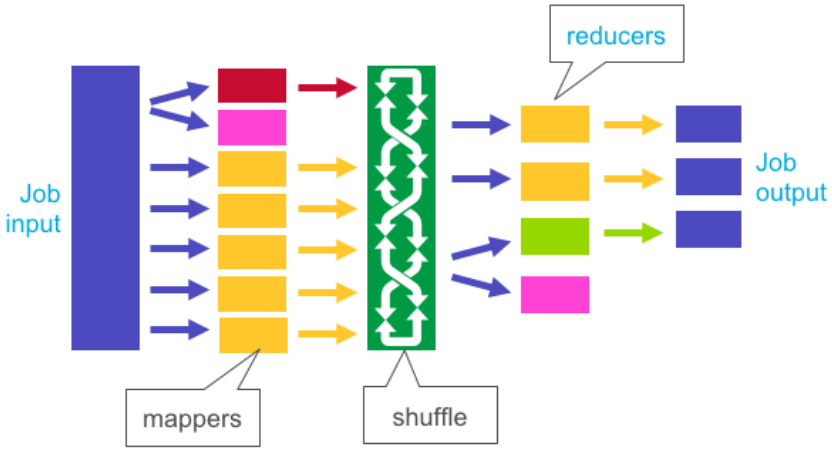
If your job is a map-only job, then you specify zero. By default, the number of reduce tasks is one.

The maximum number of reduce tasks per task tracker is a cluster-wide setting defined by the `mapred.max.reduce.tasks` parameter. The maximum number of reduce tasks that can execute at any given time across all jobs in a Hadoop cluster is given by (number of task trackers) times (`mapred.max.reduce.tasks`).

In general, you should configure `mapred.reduce.tasks` to be between 0.95 and 1.75 times the maximum number of concurrent reduce slots per cluster. At the low end, all the reduce tasks can begin at once (in a so-called single wave). At the high end, a second round of reducers can begin working if some nodes finish faster than others (achieving better parallelism in a so-called dual wave).

Depending on the logic of your reduce code, you may need to configure a specific number of reducers. For example, one output file per month requires 12 reducers, or map-only jobs require no reducer.

Speculative Execution



Speculative execution is a Hadoop enhancement in which the same task may be executed on multiple task trackers: the first and fastest result is used and all others discarded. Recall that the reduce phase does not start until all map tasks complete (including the slowest map task). Similarly, a job is not done until all the reducers have completed. If there is a particularly slow or overburdened task tracker in the cluster, tasks that run on it will cause the associated jobs to slow down. Speculative execution schedules tasks "around" these slow nodes to remove the effect of their slowdown.

While it may sound like a great thing to do in all cases, take caution.

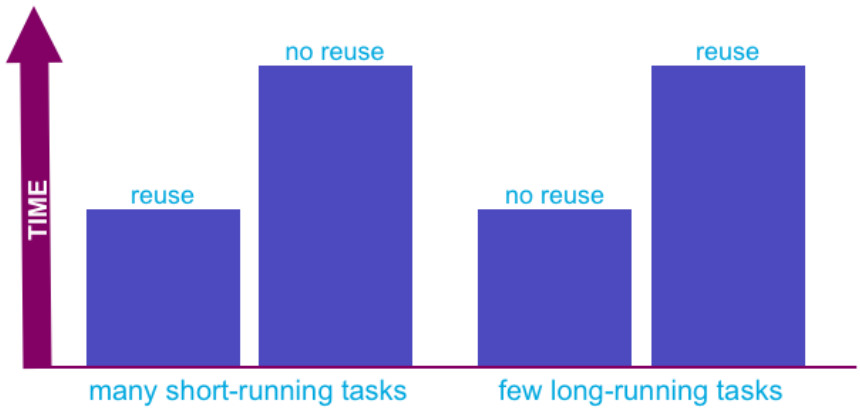
For one, the map and reduce tasks that execute speculatively must be idempotent and free of side effects. If, for example, your reducers write a single record to an external database, this is a side effect. Running multiple such reducers will produce multiple such records in the external database. If your applications can tolerate this, it's not an issue. If they cannot, you must disable speculative execution.

For two, speculative execution is a duplication of effort. The cost of launching multiple duplicated tasks must be weighed against any performance gain from getting the result from the fastest node.

In general, it is recommended to not use speculative execution.

Speculative execution occurs in both the map and reduce phases (though is separately configurable). In the diagram map phase, two tasks are launched for an identical input split. Assuming the "red" map task completes first, its output is processed and the "blue" map output is discarded. Similarly, in the diagram reduce phase, two tasks are launched for an identical partition. Assuming the ”yellow" reduce task completes first, its output is processed and the "blue" output is discarded.

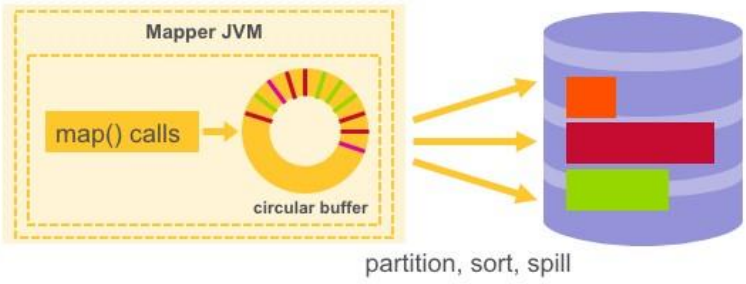
Reusing JVMs



Hadoop supports a configuration parameter called mapred.job.reuse.jvm.num.tasks which governs whether Map/Reduce JVM processes spawned once are re-used for running more than one task. This property can be found in mapred-site.xml and the default value of this parameter is one which means that the JVM is not re-used for running multiple tasks. Setting this value to -1 indicates that an unlimited number of tasks can be scheduled on a particular JVM instance. Enabling JVM reuse policy reduces the overhead of JVM startup and teardown. It also improves performance since the JVM spends less time interpreting Java byte code since some of the earlier tasks are expected to trigger JIT compilation of hot methods. JVM reuse is expected to specifically benefit in scenarios where there are large number of very short running tasks. By contrast, it is recommended to not reuse JVMs with small number of long-running tasks. First – the performance improvement will be negligible since task is long running and there are only a few tasks (JVMs) anyway. Second – long-running tasks are more likely to suffer from heap fragmentation which will actually degrade performance.

Every task executes executes in the context of a "child JVM." Don't confuse this child JVM with the JVM that is the task tracker itself. Reusing a JVM applies to the child JVM of either map tasks or reduce tasks, or both, depending on the configurations discussed.

Map-side Sort Properties



While Map tasks are running the generated intermediate output is stored into a circular buffer. This buffer is a chunk of reserved memory that is part of mapper JVM heap space. The default size of this buffer is 100 MB in Hadoop and 380 MB in MapR. This is governed by the value of `io.sort.mb` configuration parameter.

A part of this buffer is reserved for storing metadata for the spilled records. By default this is set to 0.05 (5%) of `io.sort.mb` in Apache Hadoop and 0.17 in MapR. This is governed by the `io.sort.record.percent` configuration parameter. Each metadata record is 16 bytes long.

Contents of this buffer are spilled to the disks as soon as a certain threshold of occupancy is reached by either the output data part of the buffer or the metadata part of the buffer. This threshold which is governed `io.sort.spill.percent` configuration parameter is set to 0.8 (80%) in Apache Hadoop and 0.99 in MapR by default. Spilling map output to the disk multiple times (before the final spill) can lead to additional overhead of reading and merging of the spilled records.

The default values of the properties affecting map-side spill behavior are only optimal for specific sized map input records. If there is sufficient Java heap memory available for map JVMs, the goal here should be to eliminate all intermediate spills and just spill the final output. If there is a limitation on available heap then one should try to minimize the number of spills by tuning `io.sort.record.percent` parameter values.

The map() function writes results to a circular memory buffer. When that buffer reaches a certain threshold (80% by default but configurable), the thread partitions the data according to the reducers they will be sent to, sorts the data by key, and runs the optional combiner method. Once these are complete, the data spills to files on disk. These files are merged into a single partitioned and sorted output file. The numbers of spills to merge is 10 by default but configurable. In HDFS, by default, the data is not compressed but that behavior is tunable.

Once the intermediate results are stored to local disk on the mappers, they are ready to be copied over to the reducers. Each time the memory buffer reaches the spill threshold, a new spill file is created. There may be multiple spill files when the map task is complete, and these need to be merged into a single partitioned and sorted file. The maximum number of files to merge and sort at once is defined by the `io.sort.factor` parameter which defaults to 10 in Apache Hadoop and 256 in MapR.

An easy way to detect if the Map phase is performing spills is to look at “Map output records” and “Spilled Records” counters. If the number of spilled records is greater than Map output records then spilling is occurring.

Limiting Memory Consumption

property	description
`mapred.map.child.java.opts`	JVM properties that apply to map tasks
`mapred.reduce.child.java.opts`	JVM properties that apply to reduce tasks
`mapred.child.java.opts`	JVM properties that apply to both map and reduce tasks
`mapred.child.ulimit`	Maximum size of all virtual memory consumed by a task and its children

This table shows the configuration parameters in Hadoop that control the JVM properties of a map or reduce task. You can either define one value (`mapred.child.java.opts`) that applies to all map and reduce tasks, or you can separately configure options for map and reduce tasks.

Note that the `mapred.child.ulimit` parameter defines the whole memory envelope for a JVM task. You must consider this value when you're modifying individual JVM memory properties for tasks. Note this parameter is particularly useful when doing streaming. In a MapReduce streaming task, the JVM is only there to pass data to the child process which run outside the scope of any JVM.

The JVM heap size defines the bulk of the working space for a map or reduce task. The heap contains all the non-primitive objects defined in your MapReduce code.

The default value of the heap size of a map or reduce task is 200MB, but you can configure that in two ways: setting the –Xmx<size> to the mapred.child.opts parameter or defining the HADOOP_HEAPSIZE parameter in your shell environment. It's better to use the job parameter rather than the environment variable because it allows you to more easily configure the value on a per-job basis.

In general, you should only tune the total memory and heap size if you experience out-of-memory errors in your MapReduce code. If you do tune this value, note that you need to consider other parameters as well, including the io.sort.mb property and the mapred.child.ulimit parameter. Specifically, the recommendation is that the ulimit parameter be more than twice the heap size, and the io.sort.mb parameter must be less than the heap size.

Knowledge Check

Knowledge Check

Some performance enhancements that can be used for MapReduce jobs are:

A. Using a custom Combiner (best if the number of keys is significantly less than the number of records)

B. Compressing Mapper results

C. Modifying number of reduce tasks

D. All of the above

Learning Goals

Learning Goals

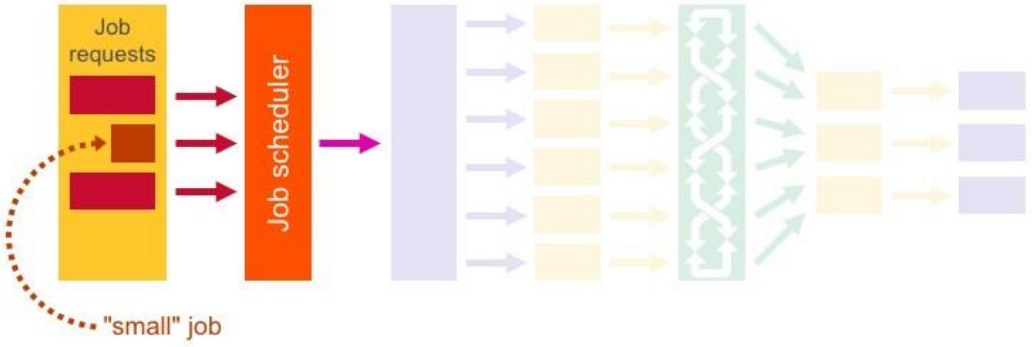
6.1 Review components of MapReduce performance

6.2 Enhance performance in MapReduce jobs

6.3 Describe MapR performance enhancements

In this section, we discuss how MapR has enhanced performance in a Hadoop cluster.

MapR ExpressLane



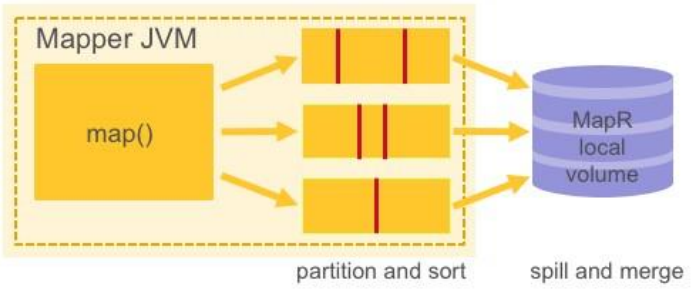
MapR has a feature called "ExpressLane" which allows small jobs to execute ahead of large jobs. As such, you can prevent large jobs from starving small jobs.

ExpressLane works by reserving one or more map or reduce slots on each task tracker for so- called "small jobs." A small job is configurable.
When a small job is submitted, if there are no available slots to execute it, the ExpressLane will schedule the job on "ephemeral slots."

If a small job executed on an ephemeral slot is found to violate the definition of "small," it is pre-empted (i.e. killed and rescheduled as a normal job).

The ExpressLane feature requires using the fair scheduler. Note that the values are configurable as appropriate for the resources and use cases on a given cluster. Small jobs can “jump ahead in the line” when the cluster is busy and does not have available slots if they satisfy the criteria.

Direct Shuffle



MapR distribution has a special feature called ”direct shuffle" which is based on the concept of a ”local" MapR volume. MapR distribution creates a volume that is local to a task tracker. This volume is called "local" because its contents are not replicated across the MapR cluster. In standard Apache Hadoop, intermediate map results are stored on the local disk of a task tracker (not on HDFS). The amount of map results that can be stored as such therefore depends on the size of the local boot disk of the task tracker. If this disk fills up, then the entire node cannot be used in the cluster.

A MapR distribution local volume, on the other hand, utilizes all the non-boot disk storage available to a task tracker to store intermediate map results. This volume will grow indefinitely (until all the storage on a task tracker fills up). If there is contention, the MapR-FS will delete replicated containers on the data node in favor of the local volume. These deleted replicas are then replicated to other data nodes in order to maintain the desired redundancy. The space required to store intermediate results will be provided as needed rather than having pre- allocated storage.

When map methods write their intermediate output in MapR-FS, they write them to the volume local to the node where the mapper is running.

Label-Based Scheduling (redux)

- Display node labels

```
$ hadoop job -showlabels
Node labels:
CentOS002 : [big, QA, IBM]
CentOS005 : [small]
CentOS006 : [medium]
CentOS004 : [small]
CentOS001 : [big, QA, CISCO]
CentOS003 : [big, ENG, HP]
```

- **Submit job with a label**

Q: what about data locality for map tasks?

```
$ hadoop jar my.jar -D mapred.job.label=big myin myout
```

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. The XML configuration file depicts how to request that a job be run on specific task trackers that match either the "big" or "good" 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.

Node Labels and Topology

```
$ hadoop job -showlabels
Node labels:
CentOS001 : [small]
CentOS002 : [small]
CentOS003 : [medium]
CentOS004 : [medium]
CentOS005 : [large]
CentOS006 : [large]
```

Hostname	Physical IP(s)	File...	Physical Topology
CentOS001	10.10.82.51 10.10.82.52	0 ago	/row1/rack1/CentOS001
CentOS002	10.10.82.53 10.10.82.54	0 ago	/row1/rack1/CentOS002
CentOS003	10.10.82.55 10.10.82.56	0 ago	/row1/rack1/CentOS003
CentOS004	10.10.82.57 10.10.82.58	0 ago	/row1/rack1/CentOS004
CentOS005	10.10.82.59 10.10.82.60	0 ago	/row1/rack1/CentOS005
CentOS006	10.10.82.61 10.10.82.62	0 ago	/row1/rack1/CentOS006

same label ↔ same topology

The answer is – you should configure your storage topology equivalent to your node labels. In this example, we have a node label called “large” that is associated with nodes centos5 and centos6. We should also configure a topology that contains these same two nodes.

Lab 6.3:
De-Tune a Job and Measure Performance Impact

Complete Lab 6.3: De-Tune a Job and Measure Performance Impact

Next Steps

Launch Jobs and Advanced Hadoop MapReduce Applications

Lesson 7: Working with Data

Congratulations! You have now completed Lesson Six. You should now have an understanding of the components of MapReduce performance, how to enhance MapReduce Performance as well as understand how MapR enhances performance. You should know to disable speculative execution, and that you can re-use JVMx for short running tasks. In the next lesson, you will learn how to work with different data sources.

Launch Jobs and Advanced Hadoop MapReduce Applications

Lesson 7: Working with Data

Welcome to DEV 302 – Launch Jobs and Advanced MapReduce Applications, Lesson 7: Working with Data.

Learning Goals

Learning Goals

7.1 Working with sequence files

7.2 Working with the distributed cache

7.3 Working with HBase

Welcome to Lesson Seven. In this lesson we discuss using sequence files for input and output in MapReduce jobs, work with distributed cache, and using HBase in MapReduce jobs.

Learning Goals

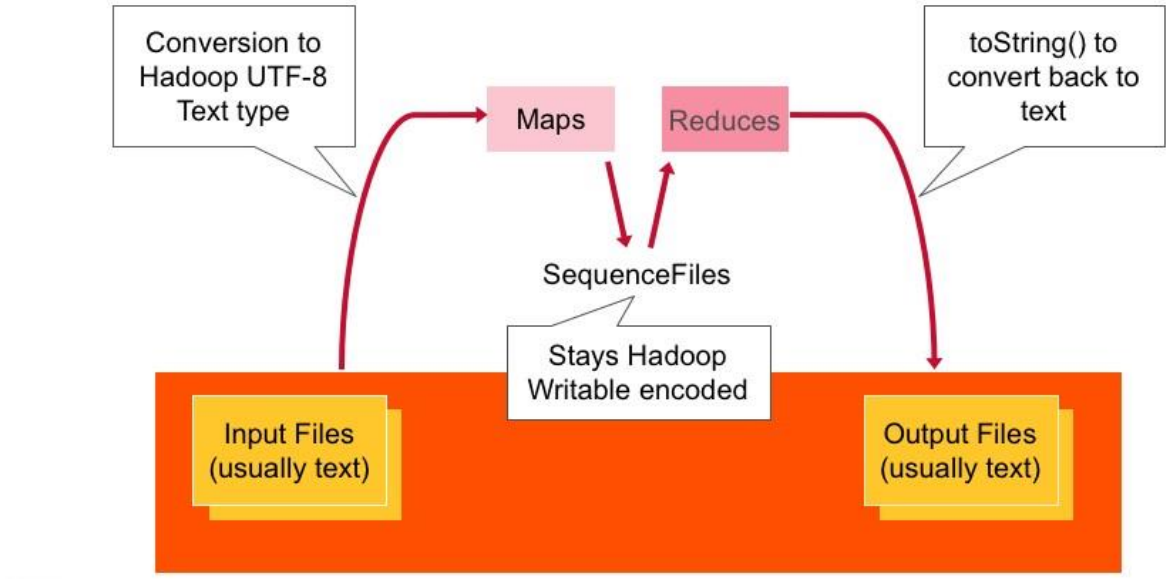
7.1 Working with sequence files

7.2 Working with the distributed cache

7.3 Working with HBase

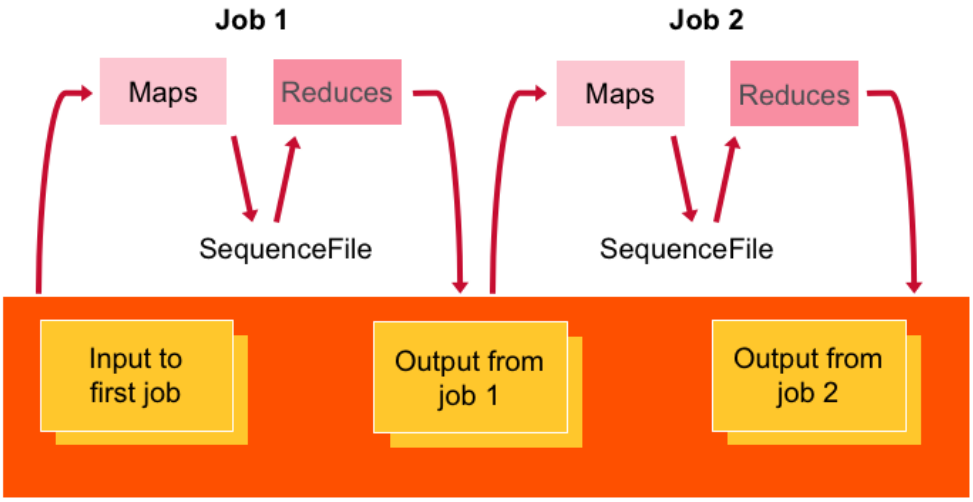
Let’s begin by discussing how to work with sequence files.

Generic MapReduce Data Flow



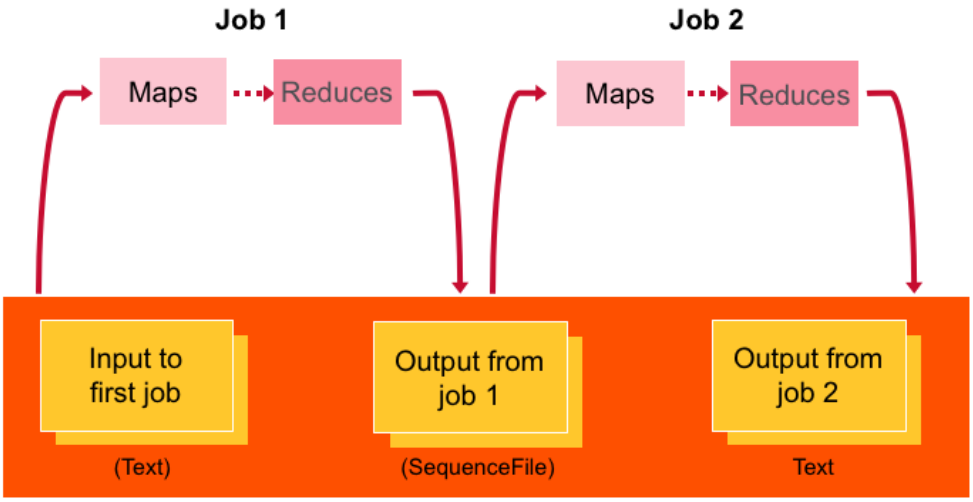
This graphic shows the typical flow of data through a MapReduce job. People use MapReduce most often to analyze text files. The Hadoop framework converts input textual data into a UTF- 8 Writable Text type and passes keys and Text values to the map method. The map method outputs intermediate data which is stored by the Hadoop framework as sequence files. Note the intermediate key-value pairs are stored as Writable types. Then the Hadoop framework shuffles the intermediate results for the reducers which get their partitions as Writable data types. Last, when the reducers write their output, it is usually written in text format.

Typical MapReduce Workflows



The MapReduce workflow shown here involves 2 MapReduce jobs. Running multiple jobs in series is very typical when using MapReduce. The restrictive nature of the data presented to mappers and reducers usually precludes writing a single "silver bullet" MapReduce job to get the work done. So the question is: why bother using text file formats if the "native" format for Hadoop is a sequence file?

Sequence Files with MapReduce



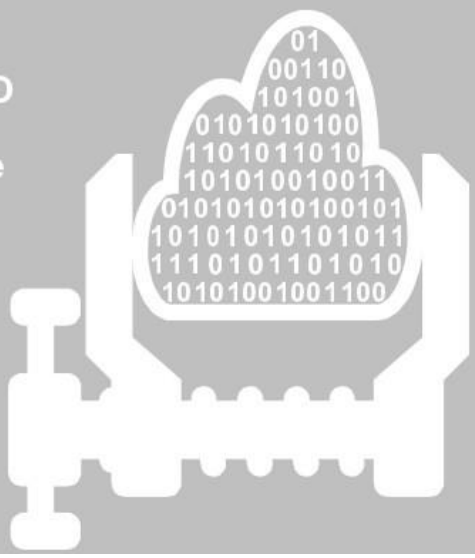
This flow shows using sequence files as intermediate file formats in a MapReduce job workflow. Note that the second job then reads its input as a sequence file and writes its output as usual (in text format). This slight variation can save space and time in complex workflows comprised of several mapreduce jobs.

File Compression

Use compression because it reduces I/O

MapR file system supports configurable compression

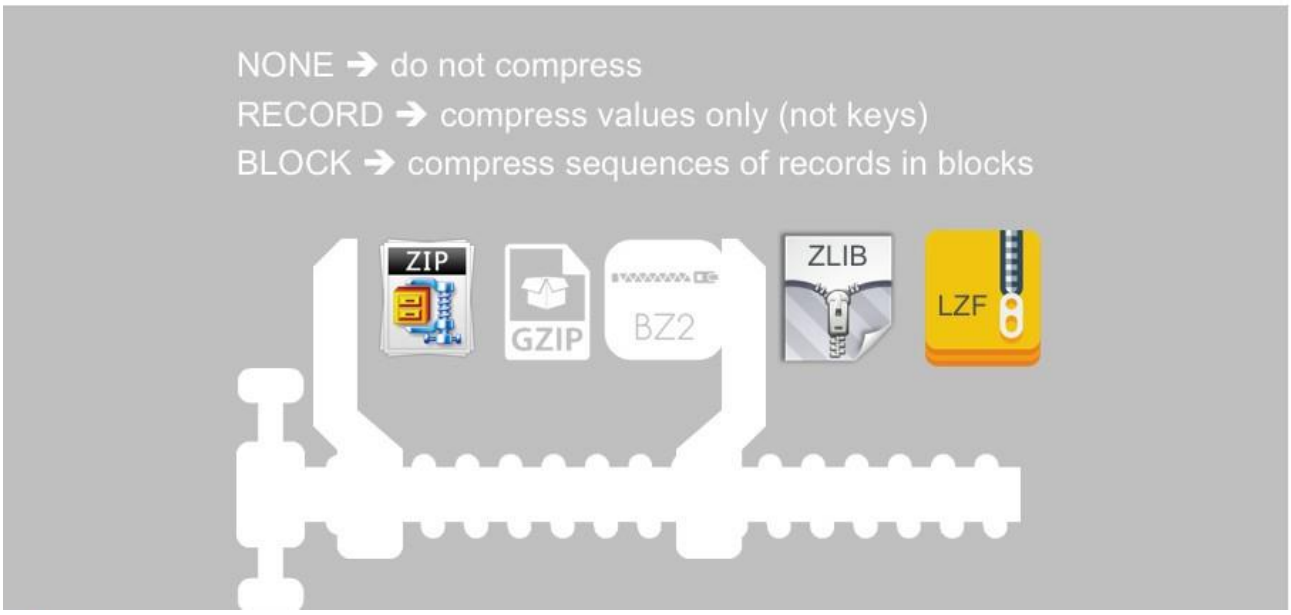
Sequence files support compression



Let’s discuss some of the aspects of using file compression on a MapR cluster. It's generally a good idea to compress files, especially large files, because it reduces the amount of I/O on the disks and networks during file transfers. MapR supports configurable compression at the volume level, and by default, all directories in that volume inherit that compression type. You can further configure compression on a directory-by-directory basis within a volume if the nature of a data within a volume lends itself to different compression types (or none at all).

MapR ships with native codecs for the built-in compression types. You can consume those native codecs by defining your LD_LIBRARY_PATH environment variable to point to the directory containing those native codecs. Last, MapR detects compression (based on either the filename extension like .gz or .zip or on the header information in a sequence file).

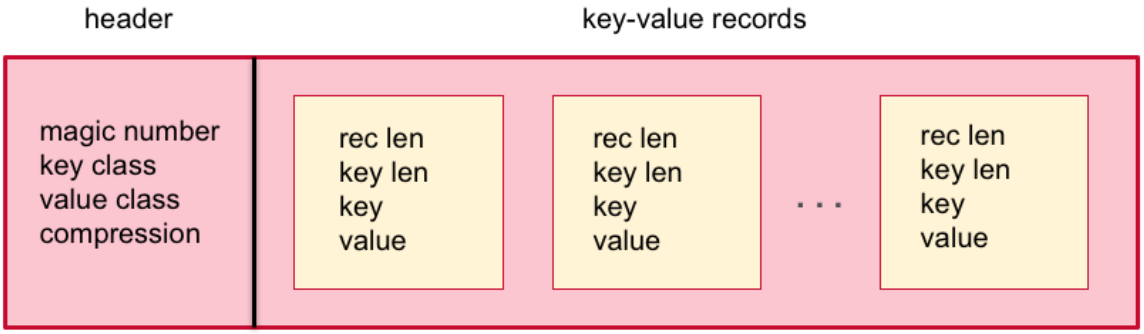
Compressing Sequence Files



When compressing sequence files, you can choose to compress at either the record or block level. When using compression, you can use any of the built-in compression codecs including LZO, LZF, GZIP, BZIP2, and ZLIB. In all cases, you define the type of compression (block or record) as well as the codec in the driver class.

NONE means do not compress
RECORD means compress values only (not keys) BLOCK means compress sequences of records in blocks

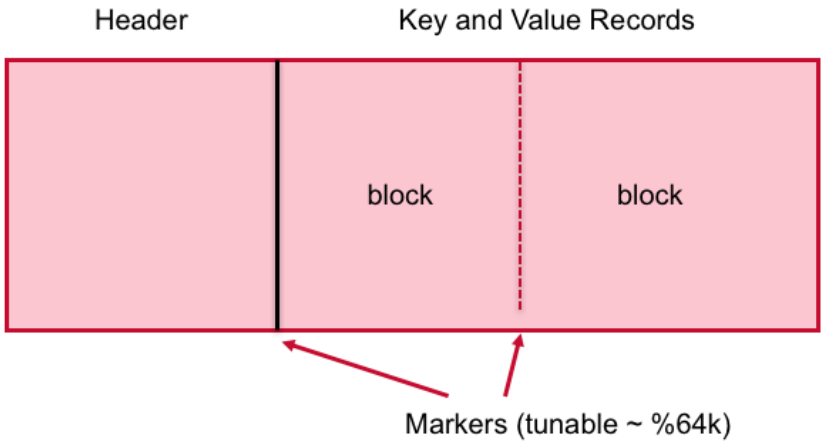
Internal Layout of a Sequence File



Let's discuss the internal layout of a sequence file. The header, stored in the front of a sequence file, contains metadata for the file. It stores a magic number which Hadoop can use to determine that it is a sequence file. It also stores the class types of the keys and values of the records stored in the file. Last, the header stores the compression used (if any) for the file.

The rest of the file is a series of tuples that contain the actual data for the sequence file. The tuples include the record length, key length, value of the key, and value of the value.

Sync Markers



The key-value records are bundled into blocks. The block delimiters are called "markers", and the size of a block is tunable.

Sync markers are used to mark a logical boundary of a record or a set of records. This is used to read records properly across block splits. In case of text files, record readers look for a line ending as '\n'. In case of sequence files there are no such record terminators, so a 'sync' marker is used instead to look for 'ends' of records so that they may be read back by Map/Reduce correctly.

MapReduce SequenceFile Syntax

```
job.setOutputFormatClass(SequenceFileOutputformat.class);

SequenceFileOutputFormat.setCompressOutput(job,true);

SequenceFileOutputFormat.setOutputCompressorClass(job,GzipCodec.class);

SequenceFileOutputFormat.setOutputCompressionType(job, CompressionType.BLOCK);

job.setInputFormatClass(SequenceFileInputFormat.class);
```

This is the code in the driver class that makes use of sequence files. In the driver, you define the following when using sequence files:

- The job output file format using the setOutputFormatClass() method
- Enable compression using the setCompressOutput() method
- Which codec to use (using the setOutputCompressorClass() method
- Which level of compression (using the setOutputCompressionType() method
- The job input file format using the setInputFormatClass() method


Knowledge Check

Knowledge Check

Which of the following are true of sequence files?

A. Intermediate data from the Mapper is stored as sequence files

B. Sequence files can be compressed

C. Using sequence files can save space and time in complex workflows comprising multiple MapReduce jobs

D. All of the above

E. 1 & 3

Learning Goals

Learning Goals

7.1 Working with sequence files

7.2 Working with the distributed cache

7.3 Working with HBase

In this section, we discuss how to use the distributed cache to distribute "side files" to mappers and reducers.

Populate the Distributed Cache Using the hadoop Command

- Pass plain text files to Mapper and Reducer classes

```
hadoop jar . . . --files file1,file2 . . .
```

- Pass Java jar files to Mapper and/or Reducer classes

```
hadoop jar . . . --libjars jar1,jar2 . . .
```

- Pass JAR/TAR or ZIP/GZIP archives to Mapper/Reducer classes

```
hadoop jar . . . --archives archive1,archive2 . . .
```

The distributed cache feature of Hadoop is a service that distributes files to mappers and reducers in a MapReduce job. There are several use cases for a distributed cache including distribution of jar files, dynamic information to run a task, or a complete data set to use in a map-side join.

The files you distribute may reside on either local file system or on the Hadoop file system.

Distributed cache supports various file types, depending on the use case (jar/tar, regular text, and compressed files). Further, you can distribute these files so that they are symbolically linked to the current working directory of the mapper or reducer task. This makes it easier to reference the files at run time.

You can use the hadoop command to pass files to the distributed cache when submitting a job from the command line. If you pass multiple files, you separate the file names with a "comma" separator.

Populate the Distributed Cache using the API

```
// driver class

DistributedCache.addCacheFile(new
Path("/user/user01/myfile").toUri(), getConf());

DistributedCache.addCacheArchive(new
Path("/user/user01/myarchive.zip").toUri(), getConf());
```

Instead of passing files to the distributed cache at the command line, you may also define files to use for the distributed cache from within the driver, as shown here.

In this example, we are adding a file called “myfile” and an archive called “myarchive.zip” to the distributed cache. You can then consume these distributed cache files in your mapper and reducer classes.

Describe How to Consume the Distributed Cache

```
// Mapper or Reducer class
. . .
Path[] localFiles, localArchives;

public void setup(Context context) throws IOException {
      Configuration conf = context.getConfiguration();
      localFiles = DistributedCache.getLocalCacheFiles(conf);
      localArchives = DistributedCache.getLocalArchives(conf);
   }

// consume cache in map or reduce method
```

From within the mapper or reducer class, you consume files in the distributed cache as shown here. In this case, we are exploiting the setup() method which gets called once before the map() or reduce() method is invoked. We pull the distributed cache configuration information out of the Hadoop configuration object. Once we have paths to the distributed cache files, we consume the data from within the map or reduce method.

Learning Goals

Learning Goals

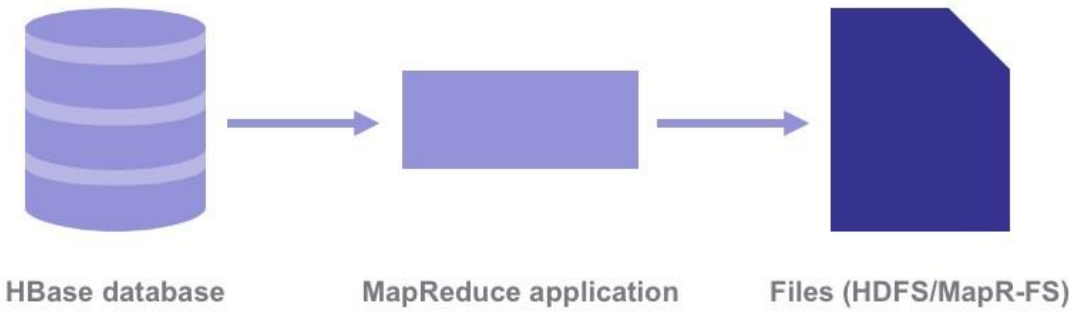
7.1 Working with sequence files

7.2 Working with the distributed cache

7.3 Working with HBase

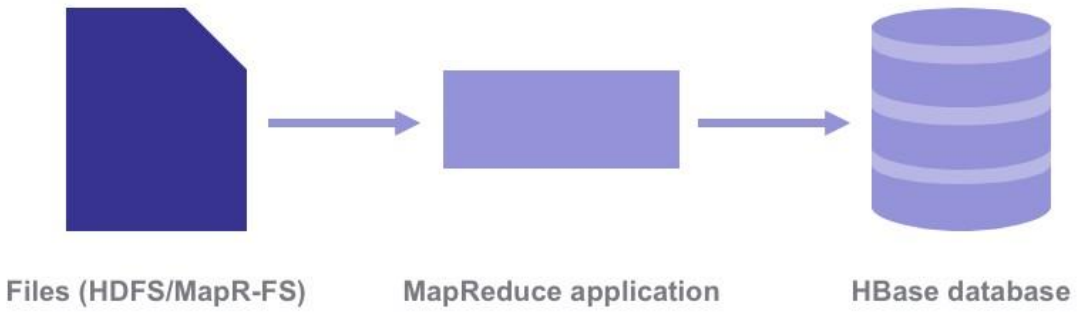
In this section, we discuss how to use HBase in a MapReduce job for data source and data sink (or both).

Using HBase as a Source



One way to leverage MapReduce with HBase is when you use the data in the Hbase database as source of your data flow. An example of doing so a frequently run well-defined batch query that you've implemented using MapReduce, and you wish to store and consume the results in files.

Using HBase as a Sink

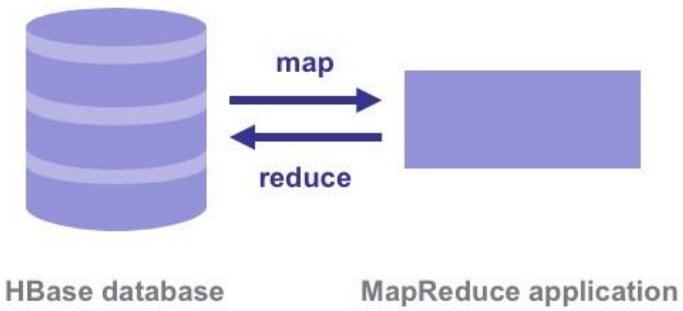


Another way to leverage MapReduce in Hbase is when you use the Hbase database as a sink for your data flow. The input for the MapReduce application can come from any variety of sources, including files. An example of using Hbase in a MapReduce application is when you use the importtsv utility to bulk load data.

When writing a lot of data to an HBase table from a MR job (e.g., with TableOutputFormat), and specifically where Puts are being emitted from the Mapper, skip the Reducer step. When a Reducer step is used, all of the output (Puts) from the Mapper will get spooled to disk, then sorted/shuffled to other Reducers that will most likely be off-node. It's far more efficient to just write directly to HBase.

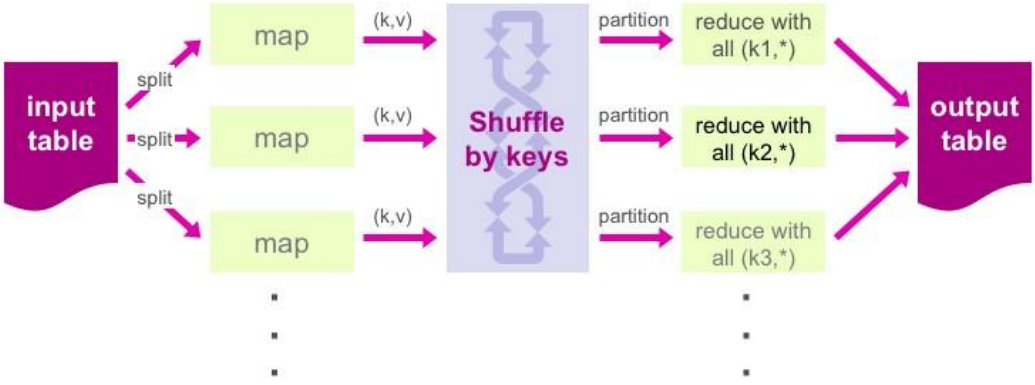
For summary jobs where HBase is used as a source and a sink, then writes will be coming from the Reducer step, for example summarize values then write out result. This is a different processing problem.

Using HBase as a Source and Sink



The last way to leverage MapReduce in Hbase is when you use the Hbase database as both a source and sink in your data flow. One example of this use case is when you calculate summaries across your Hbase data and then store those summaries back in the Hbase database. This is actually what you'll be doing in your lab.

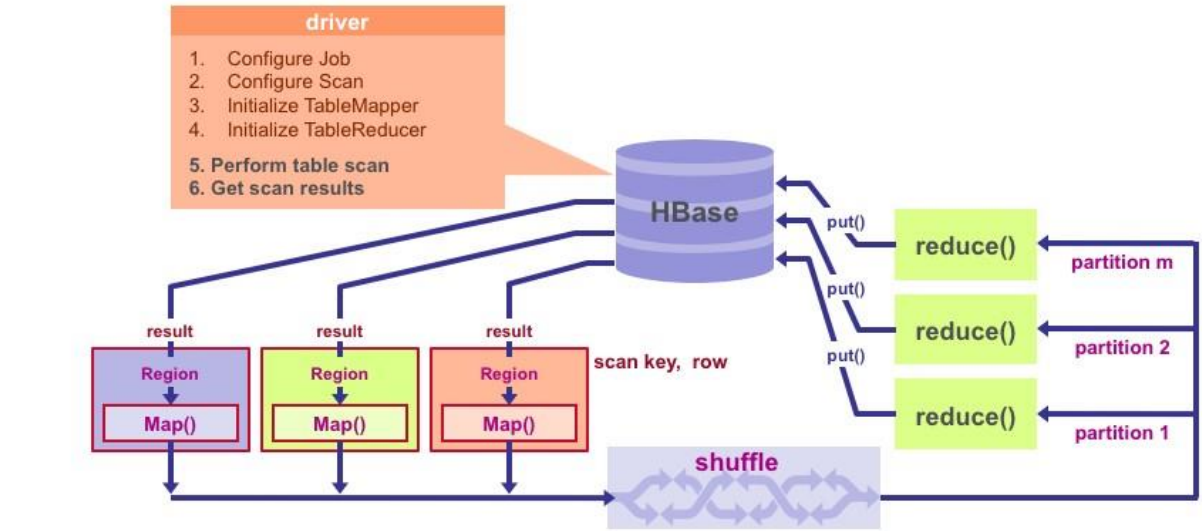
Data Flow in MapReduce with HBase



In this example, Hbase is used as both the source and sink for the map-reduce processing.

The data are split based on regions – all map tasks that process data from the same region are sent to that file server. Intermediate results from the mappers are shuffled into partitions such that all the intermediate results with the same key belong to the same partition. There is one partition per reducer. There may be more than one key in the same partition, so interpret “(kn,
*)” in this example means all keys from region “n”.

TableMapReduceUtil Model



The TableMapReduceUtil instantiates a ResultScanner before calling the map method and passes it the iterable results from the scan. The Text and LongWritable data types in that call define the KEYOUT and VALUEOUT object types for the map class.
One map task is launched for every region in the HBase table. In other words, the map tasks are partitioned such that each map task reads from a region independently. The JobTracker
tries to schedule map tasks as close to the regions as possible and take advantage of data locality.

Use a Flat-Wide Schema

```
create '/mapr/my.cluster.com/user/user01/trades_flat',
{NAME=>'price',VERSIONS=>1000000},
{NAME=>'vol',VERSIONS=>1000000},
{NAME=>'stats'}
```

rowkey	price:00	price:10	...	price:23	vol	stats:min	stats:max	stats:mean
AMZN_20131010								
GOOG_20131010								
CSCO_20131010								
...								

The slide above describes the Flat-Wide schema used in the code examples that follow. Note there are 3 column families (price, vol, and stats) in the table. The prices are written as cell versions in each hour from “00” (midnight) to “23” (11pm). Note that while this is not realistic for a real trading day (which starts at 9:30am and ends at 4:59pm), but the example is provided for instructional purposes.

Driver Example

```
public int run(String[] args) throws Exception {
      // set up the job
      Job job = Job.getInstance(getConf(), getClass().getSimpleName());
      job.setJarByClass(getClass());
     //Create and configure a Scan instance.
      Scan scan = new Scan();
      scan.setMaxVersions();
      scan.addFamily(Bytes.toBytes("price");
      // initialize the mapper and reducer
      TableMapReduceUtil.initTableMapperJob(INPUT_TABLE, scan,
              StockMapper.class, Text.class, LongWritable.class, job);
      TableMapReduceUtil.initTableReducerJob(
         OUTPUT_TABLE, StockReducer.class, job);
      return job.waitForCompletion(true) ? 0 : 1;
   }
```

First set up the job – set the jar file to the StockDriver class. In this case, we will not use the "usual" setMapperClass() and setReducerClass() methods. We will instead be defining those classes using the TableMapReduceUtil utility.

We instantiate a scan object, request all cell versions, and add a cell family to it.
We then initialize the map and reduce classes. Here note that we are passing the absolute path to the input and output tables in MapR-FS as represented by the package Text objects "INPUT_TABLE" and "OUTPUT_TABLE". They happen to be the same in this particular example (hbase = source and sink). You must specify absolute paths when using MapR-DB and relative paths when using Hbase.

The scan object is passed to the initTableMapperJob() method, but it is not passed to the mapper class. The TableMapReduceUtil instantiates a ResultScanner before calling the map method and passes it the iterable results from the scan. The Text and LongWritable data types in that call define the KEYOUT and VALUEOUT object types for the map class. There is one map task is launched for every region in the HBase table. In other words, the map tasks are partitioned such that each map task reads from a region independently. The JobTracker tries to schedule map tasks as close to the regions as possibly and take advantage of data locality.

Last we launch the job (waitForCompletion will launch the job if it's not already running).

TableMapper Example

```
public class StockMapper extends TableMapper<Text, LongWritable> {

  @Override
  protected void map(ImmutableBytesWritable key, Result row,
     Context context) {
        String keyString = Bytes.toString(key.get());
        final Text outKeyText = new Text(keyString);
        // for all columns in input row
        for (KeyValue col : row.list()) {
           byte[] cellValueBytes = col.getValue();
           long price = Bytes.toLong(cellValueBytes);
           context.write(outKeyText, new LongWritable(price));
        }
  }
}
```

Input to the overridden map method include the key and row from the scan result, along with the job context for this map job. Recall that we requested all cell versions to be part of the scan result, so we will need to iterate over the cell versions (from column price:hour in the snippet above) to emit all the results for this row key. Here we are storing the price as a long value. Converting this back to a price in $USD simply requires moving the decimal point appropriately.

TableReducer Example

```
public class StockReducer extends
     TableReducer<Text, LongWritable, ImmutableBytesWritable> {

   @Override
   protected void reduce(Text key, Iterable<LongWritable> prices,
         Context context) {
      long min=Long.MAX_VALUE;
      long temp = 0L;
      for (LongWritable price : prices) {
         temp=price.get();
         if(temp < min) min=temp;
      }
      Put put = new Put(Bytes.toBytes(key.toString()));
      put.add(OUTPUT_FAMILY, OUTPUT_COLUMN),
          Bytes.toBytes(Float.toString((float)min/100)));
      context.write(key, put);
   }
}
```

The TableReducer class takes 3 inputs – INKEY, INVALUE, and OUTVALUE which in our case are Text, LongWritable, and ImmutableBytesWritable respectively. Note that INKEY and INVALUE of the reducer class must *match* the OUTKEY and OUTVALUE of the mapper class. Also note that if the output of the reducer is written to HBase, then the data type must be some form of byte array.

What this particular reduce method does is iterate over the LongWritable prices to find the minimum value. Other statistics could be calculated in a similar fashion. The reduce method then constructs a Put object, adds the key and the minimum value into the stats:min column of the output table (defined in the job context).

Here note that the stored value in the input table is a long value. For ease of reading, we converted the minimum value to a float value to represent $dollars.cents.

Lab 7.3: Run a MapReduce Program Using HBase as a Source

Now turn to your lab guide and complete Lab 7.3.

Next Steps

Launch Jobs and Advanced Hadoop MapReduce Applications

Lesson 8: Launching Jobs

Congratulations! You have now completed Lesson Seven. You should now have an understanding of how to use sequence files for input and output in MapReduce jobs, work with distributed cache, and use HBase in MapReduce jobs. In the next lesson, you will learn about different ways of launching MapReduce jobs.

Launch Jobs and Advanced Hadoop MapReduce Applications

Lesson 8: Launching Jobs

Welcome to DEV 302 – Launch Jobs and Advanced MapReduce Applications, Lesson 8: Launching Jobs.

Learning Goals

Learning Goals

8.1 Implement programmatic job control in the driver

8.2 Use MapReduce chaining

8.3 Use Oozie to manage MapReduce workflows

Welcome to Lesson Eight. In this lesson, you will learn of different approaches for launching multiple MapReduce jobs, how to implement job control in the driver, how to use chaining and Oozie to manage MapReduce workflows.

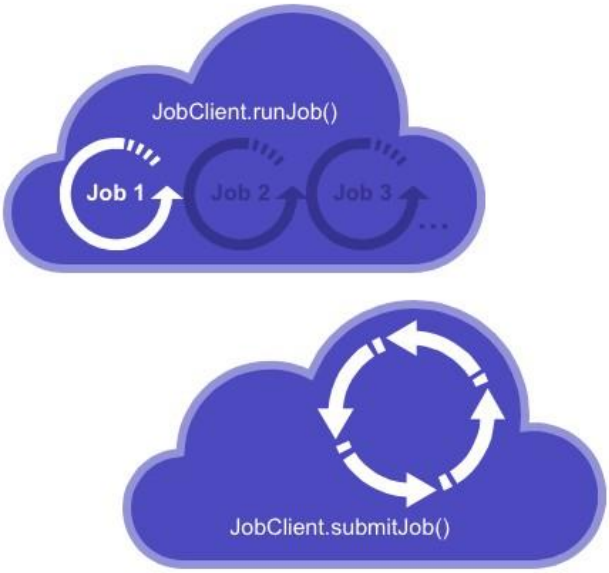
Learning Goals

8.1 Implement programmatic job control in the driver

8.2 Use MapReduce chaining

8.3 Use Oozie to manage MapReduce workflows

Review of the Job Client



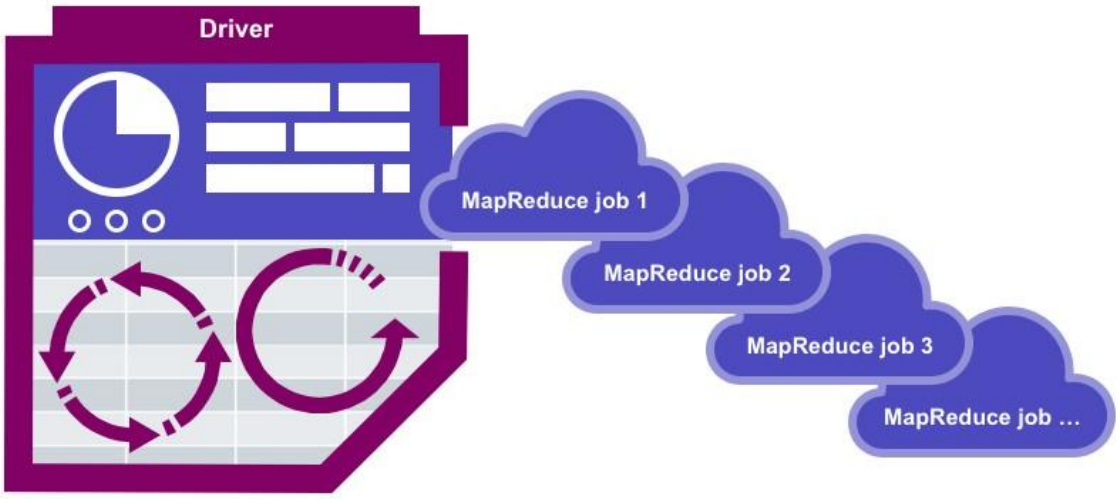
Let's review how MapReduce jobs are created and submitted in Hadoop. You've written several programs by now that launch a single job using a single job and configuration object.

Normally, you call the job client's runJob() method- which blocks
So the job client submits the job and returns only after the job has completed. This is synchronous – since it blocks your execution.

Asynchronous is also available
JobClient.submitJob() submits the job then returns control to the caller while the job executes. You code can poll the returned handle to the RunningJob to query status and make scheduling decisions.

Note that these calls use the mapred API – similar calls are available in the mapreduce API as well.

Controlling Multiple Jobs in the Driver



One way to manage multiple jobs in MapReduce is from within the driver. It is a straightforward extension of what you've already been doing in the driver to manage a single job. There is a variation of this theme using a class called "JobControl" which we won't be discussing in this course. For a small number of map and reduce classes, this method is perfectly fine. But as the number of your map and reduce classes increases (and/or as you need to incorporate other non-mapreduce functionality into your pipeline), this method becomes less and less usable. Oozie is the tool of choice to manage complex workflows, as we discuss later in this lesson.

Submitting Two Jobs

```
public class TwoJobs {
   public static class Map extends MapReduceBase implements Mapper<...> {
     public void map() { ... }
   }
   public static class Reduce extends MapReduceBase implements Reducer<...> {
     public void reduce() { ... }
   }
   public static class Map2 extends MapReduceBase implements Mapper<...> {
     public void map() { ... }
   }
   public static class Reduce2 extends MapReduceBase implements Reducer<...> {
     public void reduce() { ... }
   }
```

The code above shows using the mapred package API to create two map classes (Map and Map2) and two reducer classes (Reduce and Reduce2). Map and Reduce correspond to the first job, and Map2 and Reduce2 correspond to the second job.

Submitting Two Jobs (2)

```
public static void main(String[] args) throws Exception {
    JobConf conf1 = new JobConf(TwoJobs.class);
    conf1.setMapperClass(Map.class);
    conf1.setReducerClass(Reduce.class);
    JobClient.runJob(conf1);
    JobConf conf2 = new JobConf(TwoJobs.class);
    conf2.setMapperClass(Map2.class);
    conf2.setReducerClass(Reduce2.class);
    JobClient.runJob(conf2);
}
```

This is the main() method for the driver (which could equivalently be implemented using the ToolRunner). Notice we did not recycle the configuration between jobs because that is not supported in the mapred API. In this particular case, we are launching the first job synronously (using JobClient.runJob() from the mapred package).

Learning Goals

Learning Goals

8.1 Implement programmatic job control in the driver

8.2 Use MapReduce chaining

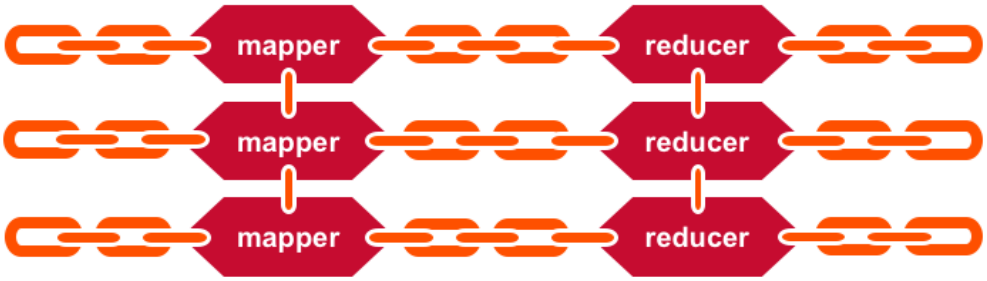
8.3 Use Oozie to manage MapReduce workflows

In this section, we discuss using MapReduce chaining.

Job Chaining

- `ChainMapper` and `ChainReducer` "chain" successive mappers or successive reducers

Data transfer is in memory, rather than disk

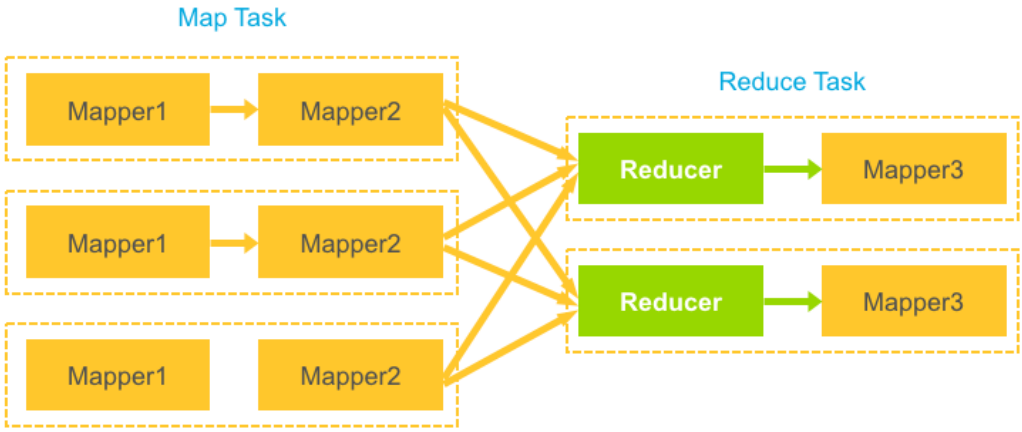


Yet another solution to managing multiple MapReduce jobs is called job chaining. The primary advantage of this solution over all the other solutions is that data from mappers are passed to the next mappers or reducers in memory rather than on disk. This can be a huge performance gain for programs that lend themselves to this model. However, there is no ecosystem interface except MapReduce.

In general, Oozie is the better choice.

- Oozie can be modifed relatively easily
- Oozie also allows job monitoring by a cluster administrator

MapReduce Chaining



Pattern: [MAP+ REDUCE MAP]*

MapReduce job chaining allows you to launch multiple mapper and reducer classes within the same task. Output from a given Mapper class can either be sent as input to the next Mapper class or as input to a Reducer class. Similarly, output from a reducer can be sent as input to a Mapper class.

All output is transferred in memory, except for the "normal" shuffle from the map phase to the reduce phase.

Note that you must use the mapred package API for job chaining as mapreduce does not support chaining.

The flow of data and execution when using MapReduce job chaining pattern is:

1. One or more mappers followed by
2. Shuffle phase followed by
3. Exactly one reducer followed by
4. Zero or more mappers

Note that you can run multiple mappers in the map phase before the shuffle/sort. Similarly, you can run multiple mappers on in the reduce phase after the singular Reducer is called.

Job Chaining Pattern

```
...
 JobConf conf = new JobConf(true);
...
 JobConf mapAConf = new JobConf(false);
 ChainMapper.addMapper(conf, AMap.class, LongWritable.class, Text.class,
Text.class, Text.class, true, mapAConf);

 JobConf mapBConf = new JobConf(false);
ChainMapper.addMapper(conf, BMap.class, Text.class, Text.class,
   LongWritable.class, Text.class, false, mapBConf);

  JobConf reduceConf = new JobConf(false);
ChainReducer.setReducer(conf, Reduce.class, LongWritable.class, Text.class,
Text.class, IntWritable.class, true, reduceConf);
 …
 JobClient.runJob(conf);
```

The code snippet above shows an implementation with ChainMapper and ChainReducer. In this example, we are constructing a chain of 2 mappers (Amap.class and Bmap.class) followed by a single reducer (Reduce.class).

Note that there is a single JobConf object (called "conf") that manages the entire job. There are also individual JobConf objects for the chain mapper job and chain reducer job.


Knowledge Check

Knowledge Check

Which of the following are true of MapReduce chaining

A. Can be used to manage multiple MapReduce jobs

B. Data from mappers are passed onto next mappers or reducers in memory

C. You must use the Mapred package for chaining

D. All of the above

E. 2 & 3 only

Learning Goals

Learning Goals

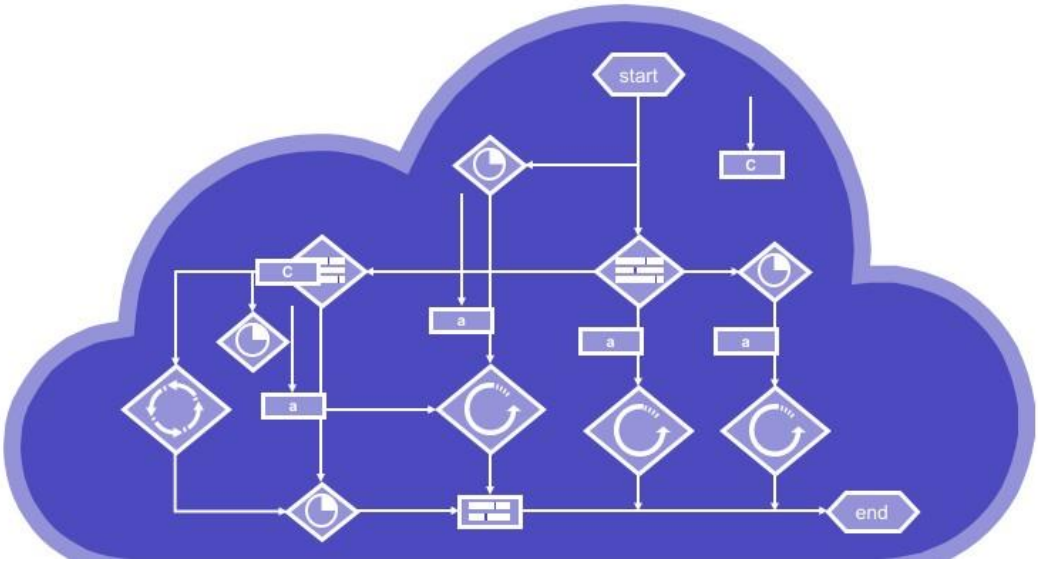
8.1 Implement programmatic job control in the driver

8.2 Use MapReduce chaining

8.3 Use Oozie to manage MapReduce workflows

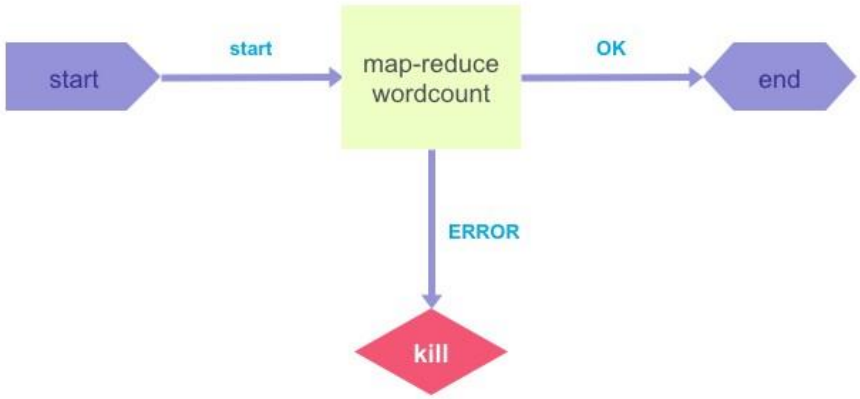
In this section, we discuss Oozie to manage Dadoop workflows

Overview of Oozie



Oozie is a client-server workflow engine for MapReduce and the Hadoop ecosystem for tasks that run on the same cluster. Workflows can be expressed as directed acyclic graphs that contain control flow and action nodes. A control flow node marks either the beginning or end of a workflow, and the action nodes are the intermediate nodes in the DAG.

WordCount Workflow DAG



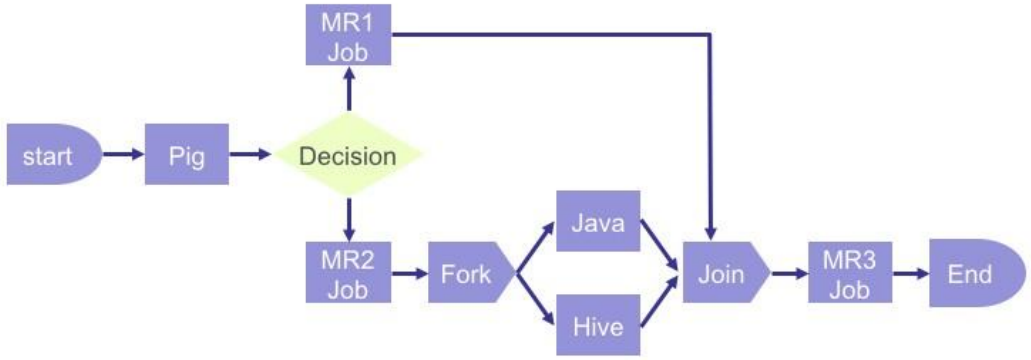
Let’s discuss the workflow for the simple wordcount MapReduce job. The "start", "end", and "kill" nodes are called control flow nodes and the "map-reduce wordcount" node is called an action node. In this directed acyclic graph, the job will either terminate due to error or successful return from the wordcount application. Obviously, oozie is intended to manage much more complex workflows than this.

Oozie Workflow XML

```
<workflow-app name='wordcount-wf' xmlns="uri:oozie:workflow:0.1">
    <start to='wordcount'/>
    <action name='wordcount'>
        <map-reduce>
            <job-tracker>${jobTracker}</job-tracker>
            <name-node>${nameNode}</name-node>
            <configuration>
                <property>
                    <name>mapred.mapper.class</name>
                    <value>org.myorg.WordCount.Map</value>
                </property>
                <property>
                    <name>mapred.reducer.class</name>
                    <value>org.myorg.WordCount.Reduce</value>
                </property>
                <property>
                    <name>mapred.input.dir</name>
                    <value>${inputDir}</value>
                </property>
                <property>
                    <name>mapred.output.dir</name>
                    <value>${outputDir}</value>
                </property>
            </configuration>
        </map-reduce>
        <ok to='end'/>
        <error to='end'/>
    </action>
    <kill name='kill'>
        <message>Something went wrong: ${wf:errorCode('wordcount')}</message>
    </kill/>
```

The same wordcount MapReduce job we just saw in the workflow DAG is ultimately defined in an XML file. Creating and modifying oozie workflows is performed by editing the corresponding XML file associated with the workflow. This makes oozie very easy to use.

More Complex Workflow Example



A more complex workflow DAG is provided here. Note that Oozie can manage non-MapReduce jobs like Pig, Hive, and standalone Java programs. Also note that fork and join operations are supported which allow for parallel processing in an Oozie workflow.

Knowledge Check

Knowledge Check

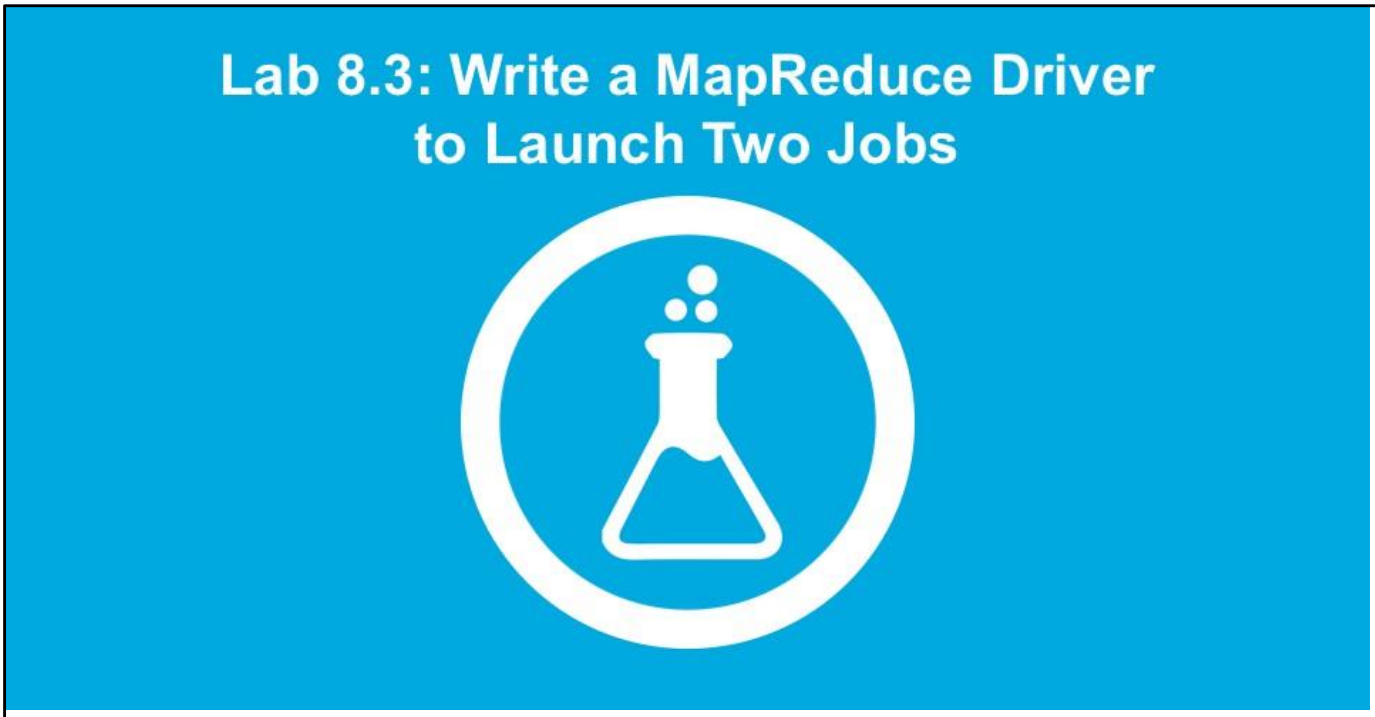
Apache Oozie is a client-server workflow engine for MapReduce and the ecosystem. Which of the following are true of Oozie?

A. Oozie workflows can be modified by editing the corresponding XML file

B. Workflows are represented as cyclic graphs

C. Oozie workflows handle linear flow

D. All of the above



Now turn to your lab guide and complete Lab 8.3.

Next Steps

Launch Jobs and Advanced Hadoop MapReduce Applications

Lesson 9: Streaming MapReduce

Congratulations! You have now completed Lesson Eight. You should now have an understanding of launching multiple jobs, chaining, and using Oozie to manage MapReduce jobs. In the next lesson, you will learn how to use non-Java languages for MapReduce applications.

Welcome to DEV 302 – Launch Jobs and Advanced MapReduce Applications,
Lesson 9: Streaming MapReduce Using non-Java Programs.

Learning Goals

Learning Goals

9.1 Overview of Streaming: using non-Java programming

9.2 Define the programming contract for mappers and reducers

9.3 Monitoring and debugging MapReduce streaming jobs

Welcome to Lesson Nine. In this lesson, we introduce the concept of using non-java programs or streaming for MapReduce jobs.

Learning Goals

9.1 Overview of Streaming: using non-Java programming

9.2 Define the programming contract for mappers and reducers

9.3 Monitoring and debugging MapReduce streaming jobs

MapReduce Streaming

MapReduce streaming enables using other languages

Streaming ***!= pipes*** (C++ equivalent)

Streaming ***may*** introduce some performance penalty

Streaming ***may*** improve performance

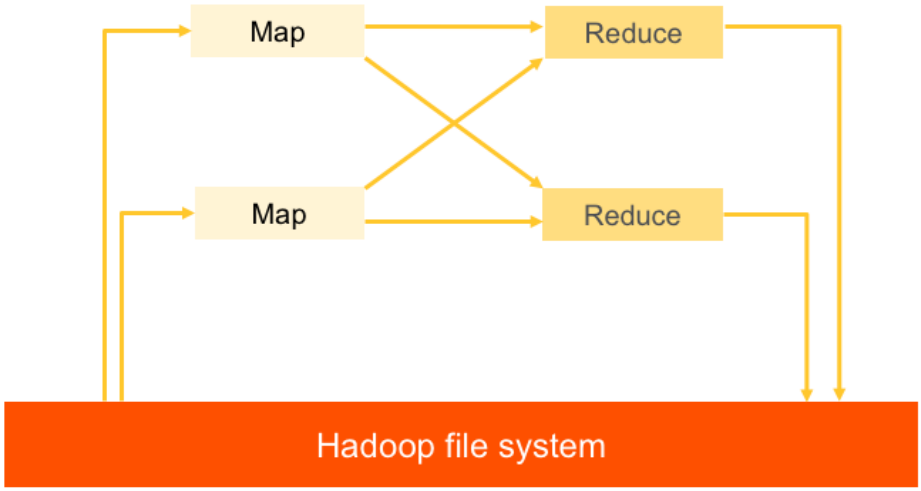
The MapReduce streaming feature allows programmers to use languages other than Java such as Perl or Python to write their MapReduce programs. You can use streaming for either rapid prototyping using sed/awk, or for full-blown MapReduce deployments. Note that the streaming feature does not include C++ programs – these are supported through a similar feature called pipes.

Streaming ***may*** introduce some performance penalty Framework still creates JVMs for tasks
Scripted programs may run more slowly

Streaming ***may*** improve performance such as
Code doing map and reduce functions may perform better than Java

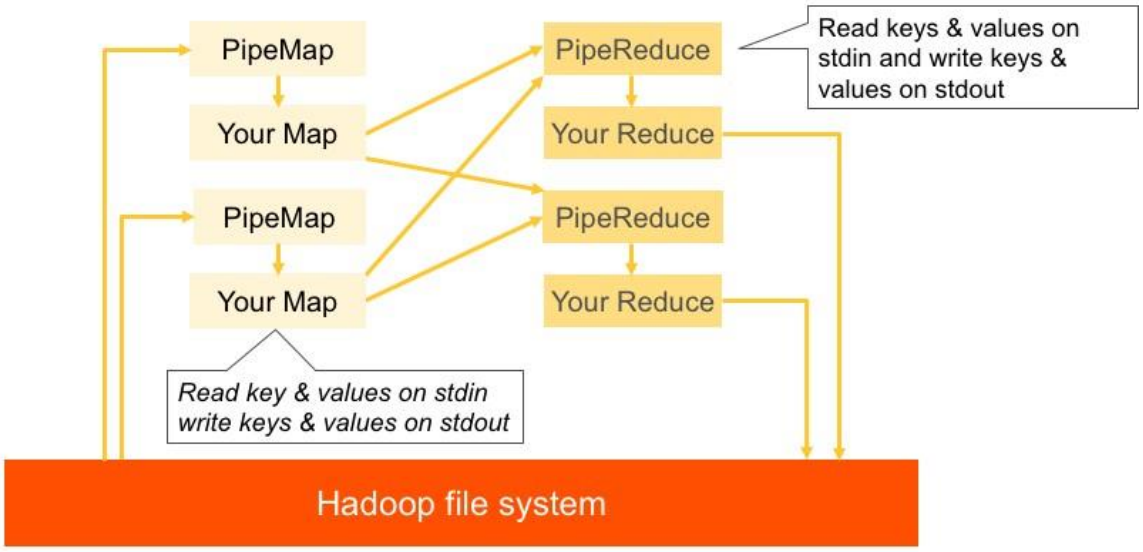
Performance is not why you should choose streaming. It should be to support your development team's favorite languages.

Review the Typical MapReduce Job



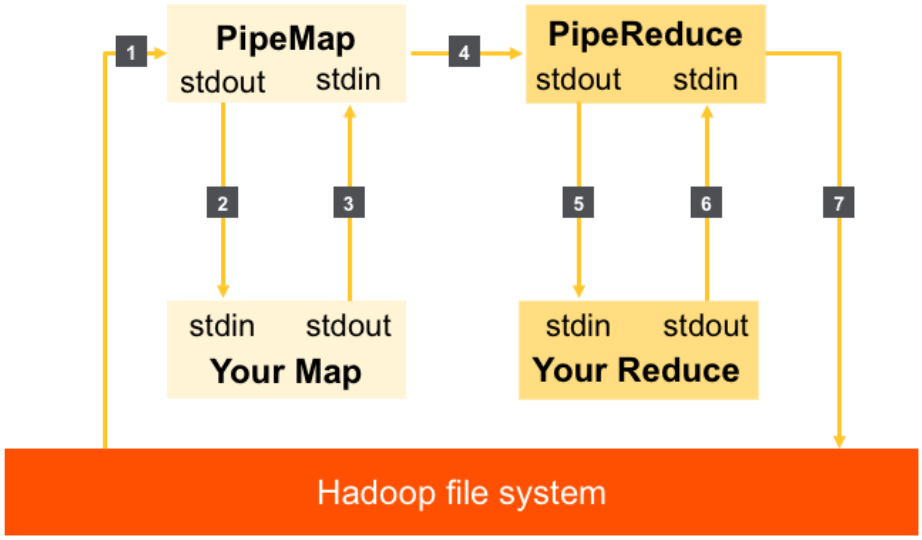
This is just review of what you already know. This is a typical MapReduce job with two Mappers and two reducers. Input to the Mappers comes from the distributed file system. Output from the mappers is sent via the framework to the reducers, which write their output back to the distributed file system.

MapReduce Streaming



This shows how the MapReduce framework uses the streaming feature. There is a mapper called PipeMap that runs inside a JVM which calls your map program. Similarly, there is a reducer called PipeReduce that runs inside a JVM which calls your reduce program. All key- value pairs are sent and received through the standard input and output channels of your map and reduce programs. The reference to a “pipe” in the names of these 2 classes comes from the fact that we are connecting standard input and standard output streams from one process to another.

Data Flow in a Streaming Job

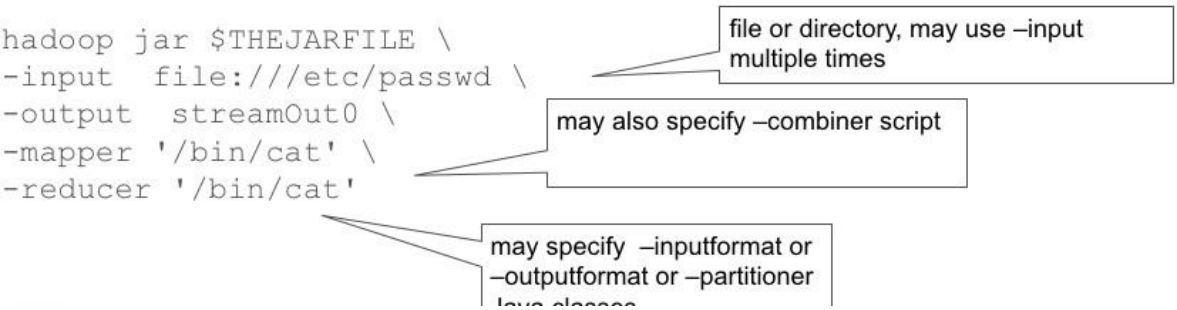


Here is a more detailed description of the data flow in a MapReduce streaming job. The PipeMap task processes input from your input files/directories and passes them to your script as standard input. Your map function processes key-value pairs one record at a time in an input split (just as in a normal MapReduce job). You write your output to standard output which is wired into the standard input of the PipeMap task. The PipeMap task then processes intermediate results from your map function, and the Hadoop framework sorts and shuffles the data to the reducers.

The same data flow mechanism occurs now on the reduce side. PipeReduce sends these intermediate results to its standard out which is wired to the standard input of your reduce script. After your reduce script processes a record from standard input, it may write to its standard output (which is wired to the PipeReduce standard input. The PipeReduce program will then collect all the output and write the output directory.

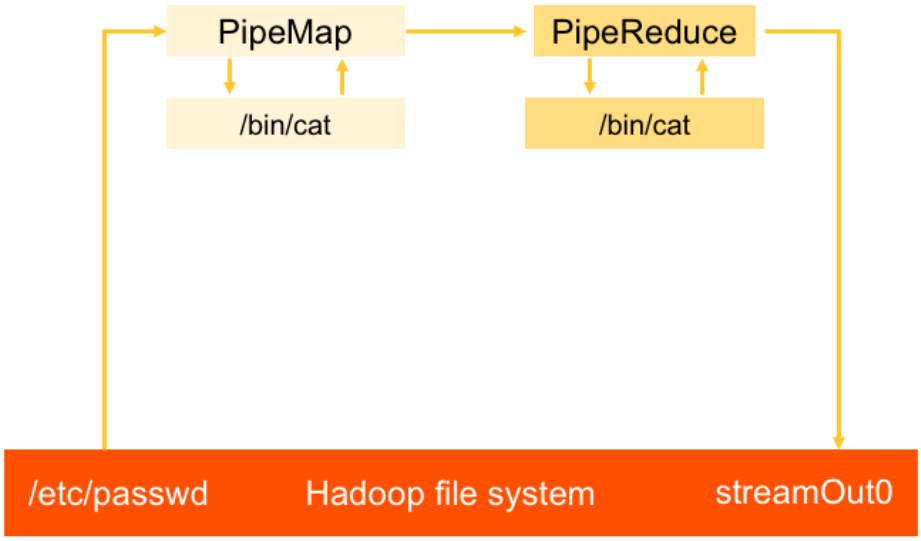
Linux Commands in Streaming (1)

```
# Read input from local filesystem
# Write output to home dir on cluster
•  #  eg. /mapr/my.cluster.com/user/...

export CONTRIB=/opt/mapr/hadoop/hadoop*/contrib/streaming
export STREAMINGJAR=hadoop-*-streaming.jar
export THEJARFILE=$CONTRIB/$STREAMINGJAR

hadoop jar $THEJARFILE \
-input  file:///etc/passwd \
-output  streamOut0 \
-mapper '/bin/cat' \
-reducer '/bin/cat'
```



The streaming invocation above shows how you can quickly test the MapReduce streaming framework. The script defines some environment variables just to make it easier to read within the screen shot (i.e. $THEJARFILE). In this example, we are simply using the /bin/cat command as a mapper to cat the local /etc/passwd file to the reducer (also the /bin/cat command). The output from this job is being sent to the "./streamOut0" directory. Think of the
/bin/cat command as the streaming analog of the identity mapper and identity reducer since it performs no transformations on the data.

Linux Commands in Streaming (2)



This is how the Hadoop framework passes input and output through the MapReduce streaming job launched in the previous slide. The PipeMap reads the /etc/passwd file from the local file system and passes it to the /bin/cat command which doesn't do any transformation of the data. It simply outputs what it gets as input. The framework then performs the same shuffle and sort on the mapper output which is sent to the PipeReduce JVM. The PipeReduce passes the intermediate results to the /bin/cat command which also doesn't perform any transformations or reductions and simply outputs the input to the specified output directory. The results in that output directory are identical to "standard" MapReduce jobs, so it contains:

- _SUCCESS file
- _logs directory
- part-xxxxx files

Linux Commands in Streaming (3)

more /etc/passwd

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
...
```

more /mapr/my.cluster.com/user/user01/streamOut0/part-00000

```
avahi-autoipd:x:103:110:Avahi autoip daemon,,,:/var/lib/avahi-autoipd:/bin/false
avahi:x:104:111:Avahi mDNS daemon,,,:/var/run/avahi-daemon:/bin/false
backup:x:34:34:backup:/var/backups:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
couchdb:x:105:113:CouchDB Administrator,,,:/var/lib/couchdb:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
games:x:5:60:games:/usr/games:/bin/sh
...
```

The slide above depicts the input file contents (local /etc/passwd file) as well as the output file from the streaming MapReduce job. Note that while the contents of both the input and output file is identical, the order of the output is changed – more specifically – sorted based on the first string in each record of the file. Also note the name of the output file is part-00000. This indicates that the streaming API uses the mapred package.

Knowledge Check

Knowledge Check

What advantages does MapReduce streaming provide?

A. You can use streaming for rapid prototyping using sed/awk.

B. It enables you to use languages other than Java (Perl, Python) to write MapReduce programs

C. It definitely enhances performance

D. All of the above

E. 1 & 2

Learning Goals

Learning Goals

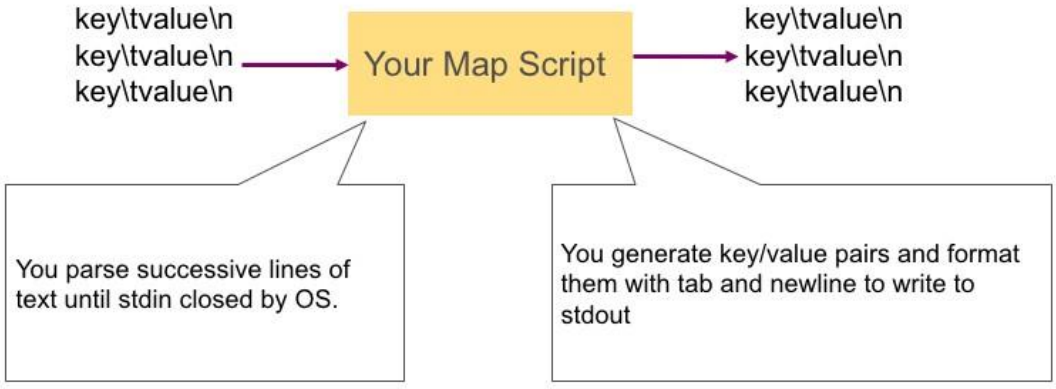
9.1 Overview of Streaming: using non-Java programming

9.2 Define the programming contract for mappers and reducers

9.3 Monitoring and debugging MapReduce streaming jobs

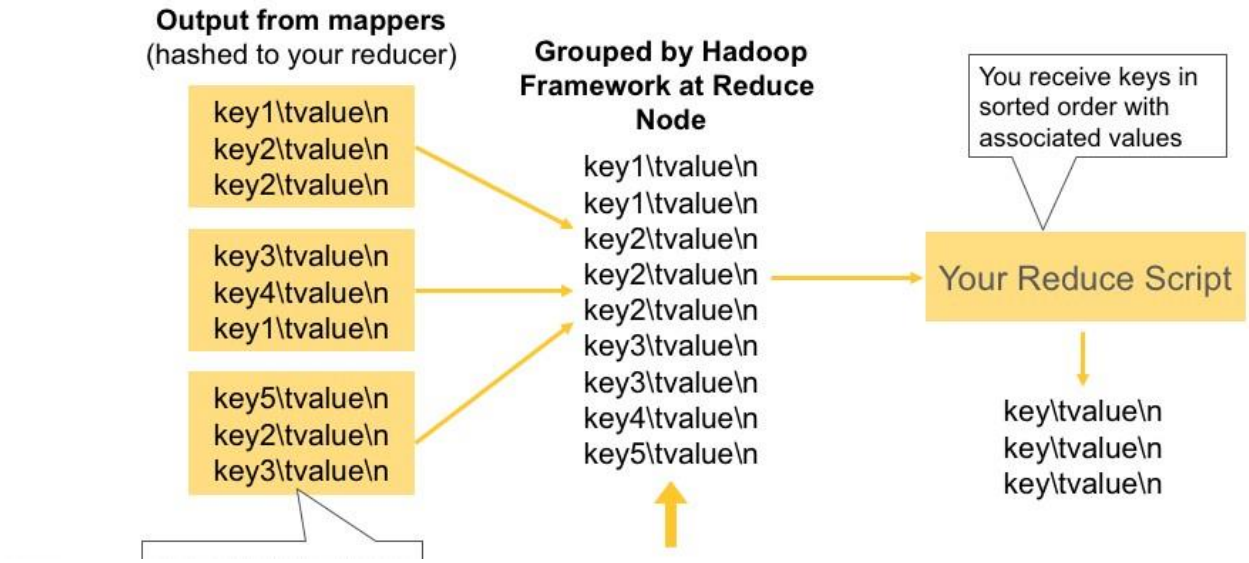
In this section, we define the programming contract for mappers and reducers in MapReduce streaming.

Data Flow Through Mapper



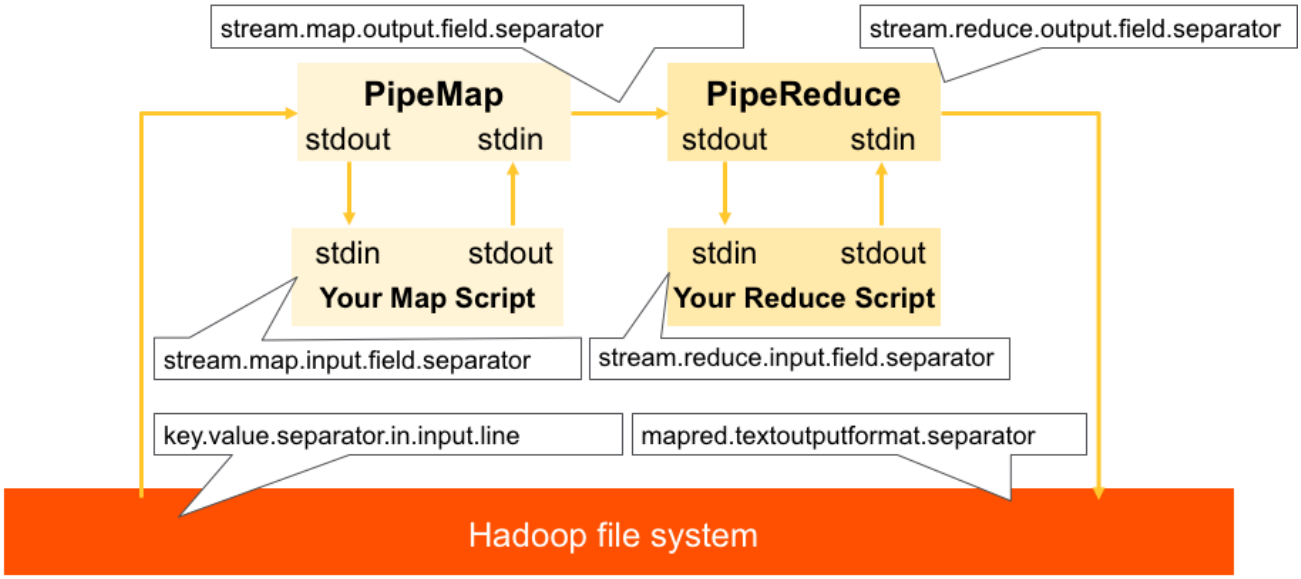
This is the data flow through the mapper in a MapReduce streaming job. Note that the mapper will continue to get records (key-value pairs) on the standard input stream until the OS closes the stream (end-of-file). Your mapper code will generate key-value pairs and emit them in the format (key \t value \n).

Data Flow Through Reducer



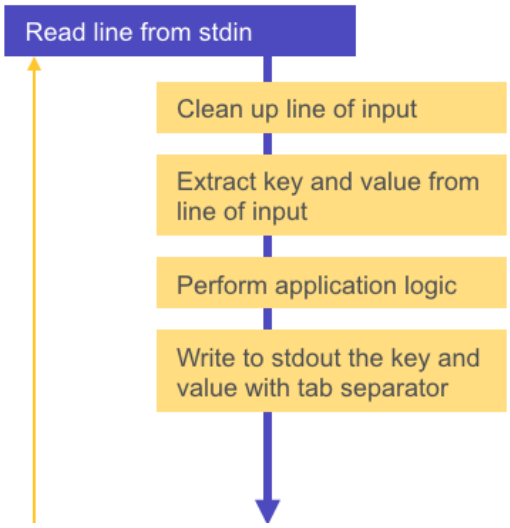
When there are multiple mappers (e.g. multiple input splits), then the intermediate results are hashed based on the key and partitioned. Your reduce program will received keys in sorted order along with their associated values, one value at a time. This is a different model than in the Java reducer paradigm in which the reducer method is called with a single key and the associated list of values at once. Your reducer code will need to account for this by checking whether the key has changed.

Configure Field Separators



You can configure the key-value separator (default is \t) using the key.value.separator.in.input.line parameter. Configure the input and output field separator in the map using stream.map.input.field.separator and stream.map.output.field.separator parameters (default is any white space). Last, you can configure the key-value separator in the reducer output files by configuring the mapred.texoutputformat.separator (default is \t).

Mapper Data Processing



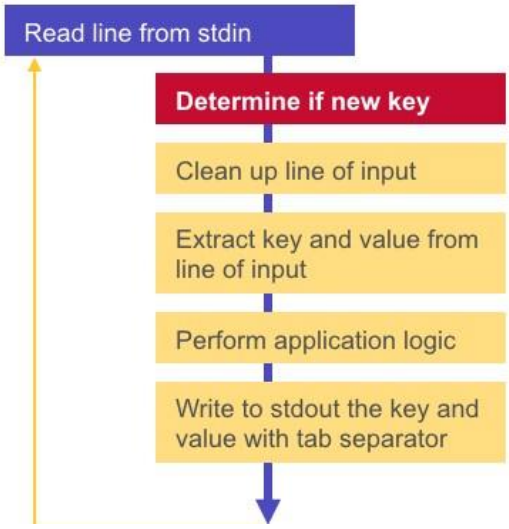
This shows how a MapReduce streaming mapper function processes data. It loops reading standard input, cleans up the line, extracts the key and value, performs any operations per your program logic, and then writes key and value (separated by \t) to standard output.

Perl Example: Mapper

```
#!/usr/bin/env perl
while (<>) {      # read stdin
  chomp;    # remove last char if newline from the implicit variable $_
  my ( $key,$value ) = split(/\t/,$_);     # extract key and value
  print "key:".$key."\t"."value:".$value."\n";     # write to stdout
}
```

This Perl code is a sort of "identity" mapper in the sense that it doesn't perform any transformations or filtering of the input data. It loops through standard input until the operating system closes the stream (end of file). For each record, it removes the last character (\n), and defines key and value variables as output to the split method (split on \t). It then just prints "key:" followed by the key and "value:" followed by the value to standard out. Note the key and value strings are separated by the tab character, and the key-value pair is terminated by a newline character. This constitutes the intermediate results of the streaming job.

Reducer Data Processing



Let’s discuss how a MapReduce streaming reducer function processes data. It loops reading standard input, and starts by determining if it's looking at a new key. Then it cleans up the line, extracts the key and value, performs any operations per your program logic, and then writes key and value (separated by \t) to standard output.

Reducer Data Processing

```
#!/usr/bin/env python
import sys

currentKey = None
groupTotal = 0

for line in sys.stdin:
  line = line.strip()  # remove leading and trailing whitespace
  if not line:    # ingnore empty lines
     continue

  keyFromLine, valueFromLine = line.split("\t", 1)

  if currentKey == keyFromLine:  # key hasn't changed
        groupTotal += int(valueFromLine)
  else:   # we just got new key
     if currentKey:  # if not first line of processing
        print( "%s\t%d" % (currentKey, groupTotal) )
     groupTotal = int(valueFromLine)  # total is just our current valueFromLine
     currentKey = keyFromLine  # set the new key as our currentKey

# The last key must be processed
if currentKey == keyFromLine:
  print( "%s\t%d" % (currentKey, groupTotal) )
```

Unlike a Java MapReduce program for which the reducer is sent an iterable list of values for a given key, a streaming mapreduce program is sent all the key-value pairs one at a time.
Assuming you're performing a set of operations in your reducer on all the values of a given key, you will need to identify when the key changes in the standard input.

In the reduce script above, we use the "currentKey" and "keyFromLine" variables to make this distinction. This particular python reduce script writes the key and total number of records associated with that key to standard out. As we know, the PipeReduce class will capture this and write it to the output directory we specify when we launch this streaming m/r job.

Python MapReduce Job

```
export CONTRIB=/opt/mapr/hadoop/hadoop*/contrib/streaming
export STREAMINGJAR=hadoop-*-streaming.jar
export THEJARFILE=$CONTRIB/$STREAMINGJAR
rm -rf /mapr/my.cluster.com/user/user01/pyNoOpMapRed 2>>/dev/null
hadoop jar $THEJARFILE \
   -D mapred.reduce.tasks=2 \
   -mapper "python noOpMap.py" \
   -file noOpMap.py  \
   -reducer "python noOpReduce.py" \
   -file noOpReduce.py  \
   -input file:///$PWD/goodInput.txt   \
   -output "pyNoOpMapRed"
```

The commands here show how to prepare for and launch a MapReduce streaming job in python. Note that you can pass the “usual” generic hadoop job options when submitting a streaming mapreduce job to the framework. In this case, we are requesting that two reduce tasks get launched in this job.

Launching Perl Streaming Code

```
#!/usr/bin/env bash
export CONTRIB=/opt/mapr/hadoop/hadoop*/contrib/streaming
export STREAMINGJAR=hadoop-*-streaming.jar
export THEJARFILE=$CONTRIB/$STREAMINGJAR

hadoop jar $THEJARFILE \
  -mapper 'perl map.pl' \
  -reducer 'perl reduce.pl' \
  -file map.pl \
  -file reduce.pl \
  -input  file:///etc/passwd \
  -output  perlOut
```

This script shows how to launch a perl mapreduce streaming job using a bash shell script.

Learning Goals

Learning Goals

9.1 Overview of Streaming: using non-Java programming

9.2 Define the programming contract for mappers and reducers

9.3 Monitoring and debugging MapReduce streaming jobs

In this section, we discuss monitoring and debugging MapReduce streaming jobs.

General Debugging Tips

Launch map or reduce script ***"standalone"***
with stdin

Test with ***bad data***

Run a ***map-only*** job to test mapper

Run ***reduce-only*** with an "identity mapper" that you create

Use ***counters and status***

Here are some general debugging tips when writing streaming MapReduce code. Note there isn't a lot for you to do – unlike with Java MapReduce code.

Ensure that your mapper and reducer script can run on it's own by feeding it input on standard in. As with any program, you should test with bad data (i.e. is not formatted according to what your map and reduce scripts expect). Test the map and reduce functions in the hadoop framework by using "identity" mapper and reducers accordingly. Last, you can update counters and status from within your mapper and reducer scripts – as explained in the next slide.

Perl Mapper Updating a Counter

Perl

```
. . .
print STDERR "reporter:counter: MyGroup,NumberOfRecords,100\n";
```

Python

```
. . .
sys.stderr.write("reporter:counter: Mygroup, NumberOfRecords,100\n"
```

To use and update counters and status from within your map and reduce scripts you can send from streaming code. The Hadoop framework listens for output on the standard error stream of your map and reduce scripts and looks for strings beginning with the string "reporter:counter" and "reporter:status". Each update to the reporter must appear on its own line.

This is effectively the extent of debugging when programming in streaming. The string has the format:
reporter:counter: group, counter, amount

The call to standard error is given above for both Perl and Python.

Perl Reducer Updating Status

```
#!/usr/bin/env perl
my $numRecords = 0;
while (<>) {       # read from stdin
  my ($key,$value) = split(/\t/);      # split input on newline
  $numRecords += 1;                    # count num of records
}
print "$numRecords\n";               # after all stdin processed
                                     # write the number of records
print STDERR "reporter:status:A Reducer Exiting\n";
```

When a job progress is updated, the mapper or reducer reads an input record or writes an output record, it is considered progress. If the job calls the Reporter methods, it is considered progress. The default of 600 seconds represents the amount of time a task will wait on progress before timing out.

To generate use the string: reporter:status:message

The string is passed to Reporter.setStatus() by the Hadoop Framework. Status messages are "transient". They appear in a "status" column for running processes. As status changes, the field is overwritten. This is typically used for a very long running process so that you can check the status column as the job runs.

Lab 9.3: Implement a MapReduce Streaming Application

Now turn to your lab guide and complete Lab 9.3.

Congratulations!

You've completed the course