pdf

MODULE 1 – INTRODUCTION TO MAPREDUCE AND YARN

OVERVIEW

Course sections

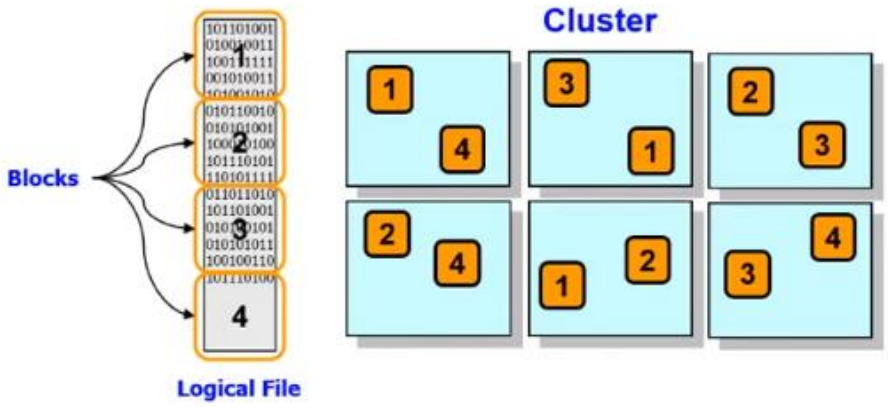
1. MapReduce processing based on MR1
   - 1a. Introduction
   - 1b. MapReduce phases
   - 1c. Wordcount example
   - 1d. Miscellaneous details

   Lab Exercise 1
2. Issues with / Limitations of Hadoop v1 & MapReduce v1
   - 2a. Overview
   - 2b. YARN features
3. The Architecture of YARN
   - 3a. High-level architecture
   - 3b. Running an application

   Lab Exercise 2

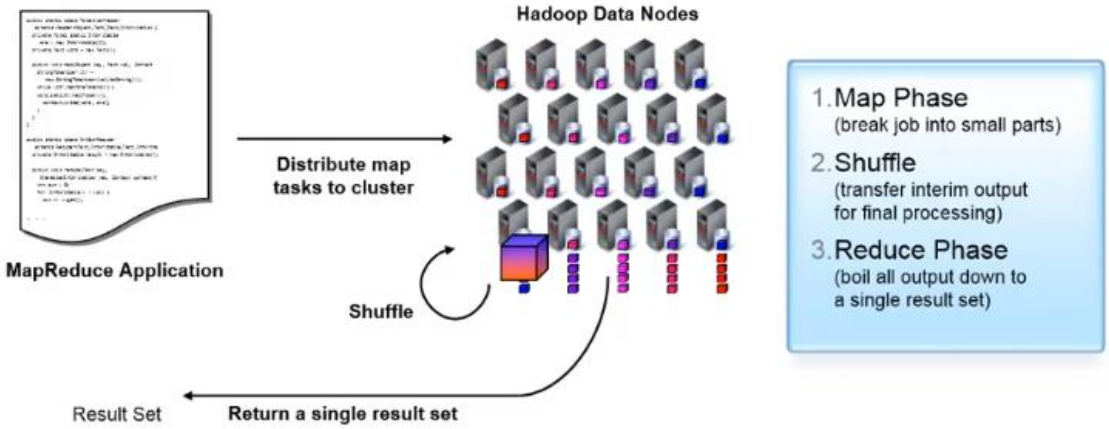
MapReduce – the Distributed File System

- Driving principals
  - Data is stored across the entire cluster
  - Programs are brought to the data, not the data to the program
- Data is stored across the entire cluster (the DFS)
  - The entire cluster participates in the file system
  - Blocks of a single file are distributed across the cluster
  - A given block is typically replicated as well for resiliency



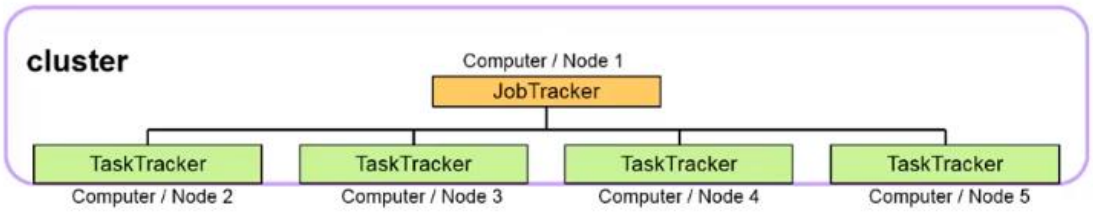
MapReduce v1 explained

- **Hadoop computational model**
  - Data stored in a distributed file system spanning many inexpensive computers
  - Bring function to the data
  - Distribute application to the compute resources where the data is stored
- **Scalable to thousands of nodes and petabytes of data**



MapReduce v1 engine

- **Master / Slave architecture**
  - Single master (JobTracker) controls job execution on multiple slaves (TaskTrackers).
- **JobTracker**
  - Accepts MapReduce jobs submitted by clients
  - Pushes *map* and *reduce* tasks out to TaskTracker nodes
  - Keeps the work as physically close to data as possible
  - Monitors tasks and TaskTracker status
- **TaskTracker**
  - Runs map and reduce tasks
  - Reports status to JobTracker
  - Manages storage and transmission of intermediate output



The MapReduce programming model

- "Map" step
  - Input is split into pieces (HDFS blocks or "splits")
  - Worker nodes process the individual pieces in parallel (under global control of a Job Tracker)
  - Each worker node stores its result in its local file system where a reducer is able to access it
- "Reduce" step
  - Data is aggregated ("reduced" from the map steps) by worker nodes (under control of the Job Tracker)
  - Multiple reduce tasks parallelize the aggregation
  - Output is stored in HDFS (and thus replicated)

The MapReduce execution environments

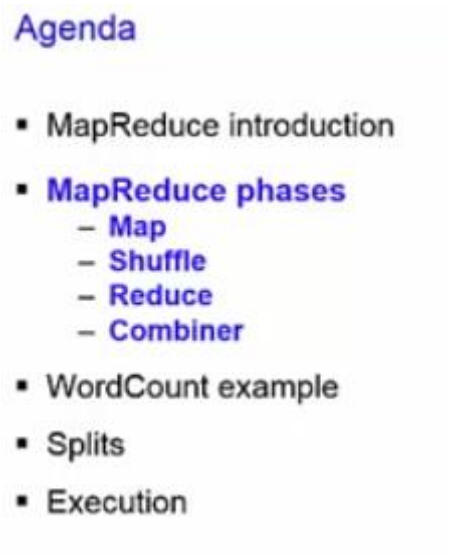
- APIs vs. Execution Environment
  - APIs are implemented by applications and are largely independent of execution environment
  - Execution Environment defines how MapReduce jobs are executed
- MapReduce APIs
  - org.apache.mapred:
    - Old API, largely superseded some classes still used in new API
    - Not changed with YARN
  - org.apache.mapreduce:
    - New API, more flexibility, widely used
    - Applications may have to be recompiled to use YARN (not binary compatible)
- Execution Environments
  - Classic JobTracker/TaskTracker from Hadoop v1
  - YARN (MapReduce v2): Flexible execution environment to run MapReduce and much more
    - No single JobTracker, instead ApplicationMaster jobs for every application

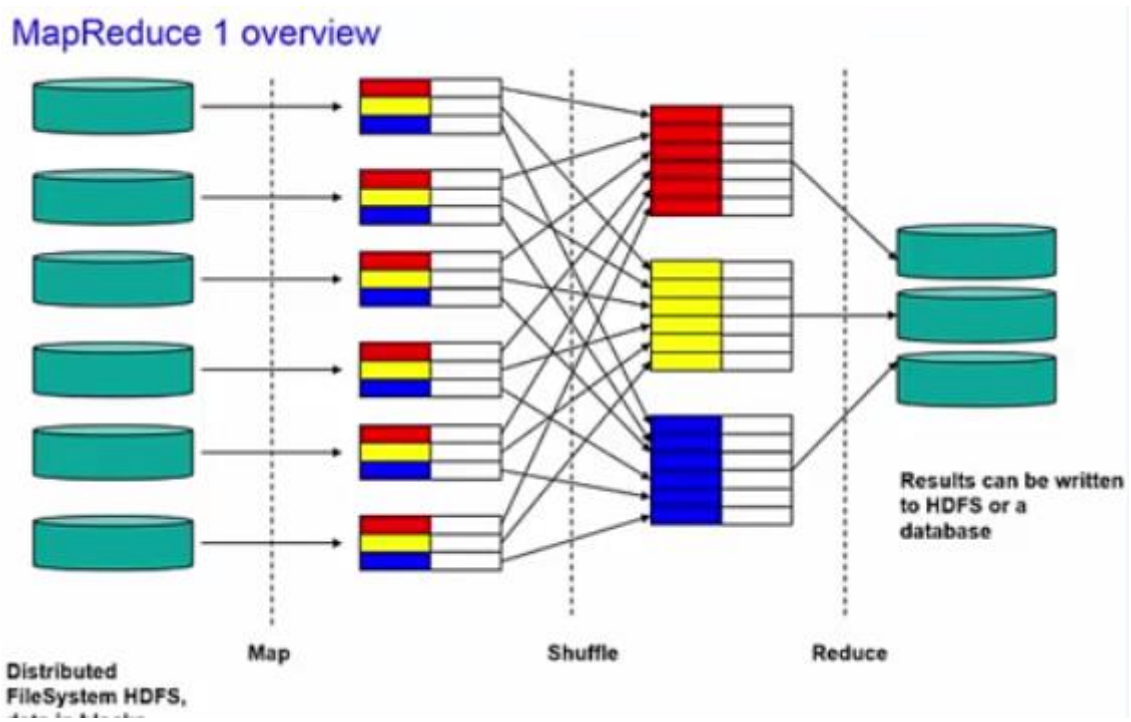
In this course, you will study MapReduce and YARN over a total of three major sections, eight videos, and two hands-on labs. The objectives of the course are to provide you with the ability to: Describe the MapReduce model v1 — this is the "classic" version that comes with Hadoop 1. List the limitations of both Hadoop 1 and MapReduce 1 Review the Java code required to handle the Mapper class, the Reducer class, and the program driver needed to access MapReduce Describe the YARN model, including the features of YARN and how a YARN program is run, and finally,
Compare "YARN / Hadoop 2 / MR2" versus "Hadoop 1 with MapReduce1".
In this course there are three main sections and a total of eight videos. There are hands-on labs after the 4th and the 8th video. We will first look at "classic" MapReduce processing, based on MapReduce v1. Then there will be a discussion of the issues with and limitations of that approach. And, finally, we will study the architecture

or YARN — Yet Another Resource Negotiator — and how YARN/MapReduce v2 relates to classic
MapReduce v1. We start with an Introduction to MapReduce
processing based on MapReduce v1. This is the agenda for the first section of
this course. It deals with classic MapReduce, now known as MapReduce v1 (MR1).
The driving principal of MapReduce is a simple one: spread your data out across a large cluster of machines and then — rather than bringing the data to your programs as you do in a traditional
programming environment — you write your program in a specific way that allows the program logic
to be moved to the data. Thus, the entire cluster is brought to bear in both reading
the data as well as processing the data. A distributed file system — the Hadoop Distributed File System, aka HDFS — is at the heart of MapReduce. It is responsible for spreading
data across the cluster, by making the entire cluster look like one giant file system. When
a file is written to the cluster, blocks of the file are spread out and replicated across
the whole cluster. In the diagram we can see that every block of the file is replicated
to three different nodes, or machines, in a cluster of machines.
Adding more nodes to the cluster instantly adds capacity to the file system and, as we'll see on the next slide, automatically increases the available processing power and parallelism.
There are two aspects of Hadoop that are important to understand:
MapReduce is a software framework originally introduced by Google to support distributed computing on large data sets held on a clusters of computers.
The Hadoop Distributed File System (HDFS) is where Hadoop stores its data. This file
system spans all the nodes in a cluster. Effectively, HDFS links together the data that resides on many local nodes, making the data part of one big file system.
Furthermore, HDFS assumes nodes will fail, so it replicates a given chunk of data across multiple nodes to achieve reliability. The degree of replication can be customized by
the Hadoop administrator or programmer. However, the default is to replicate every chunk of data across 3 nodes: 2 on the same rack, and 1 on a different rack.
The key to understanding Hadoop lies in the MapReduce programming model. This is essentially
a representation of a divide and conquer processing model, where the files input is split into many small pieces (in the map step), and the Hadoop nodes process these pieces in parallel.
Once these pieces are processed, the results are distilled (in the reduce step) down to
a single set of results. If one TaskTracker is very slow, it can delay
the entire MapReduce job -- especially towards the end of a job, where everything can end up waiting for the slowest task. With speculative-execution enabled, however, a single task can be competitatively
executed on multiple slave nodes in parallel. For jobs scheduling, by default Hadoop uses a FIFO approach(First in, First Out) with five optional scheduling priorities to schedule
jobs from a work queue. Other scheduling algorithms are available as add-ins: Fair Scheduler, Capacity Scheduler, and others.. The JobTracker master node breaks the job
up into Map and Reduce steps, finds where the blocks or splits of the input file or
files are located, and then assigns work to appropriate worker nodes for Map and Reduce processing. In the "Map" step, each worker node participating
in the Mapper process takes its designed block of input, reads it, and then performs the Map processing on it. It produces an output file for each of the Reducers that participate in the follow up step. In the "Reduce" step, the answers to all the
sub-problems generated by the Map nodes are sent to a set of Reduce nodes where they are combined or aggregated in some way to produce the final output — the results of the problem MapReduce was originally asked to solve. The slide shows the execution environment
and describes the Application Programming Interfaces (APIs) that are used. We will see

that a similar, and somewhat compatible, API environment is used for both MapReduce 1 and for YARN/MapReduce 2. This is the end of this video. In the next
video we will look at the processing phases in the MapReduce process.

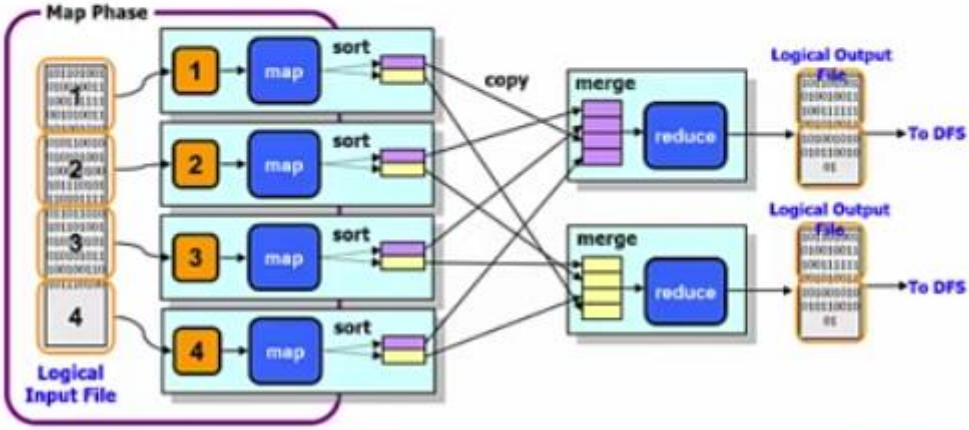
MARPREDUCE PHASES





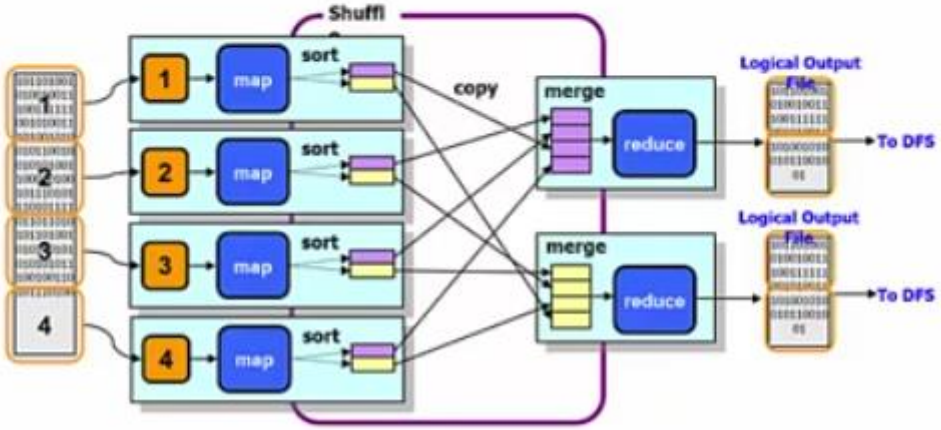
MapReduce – Map Phase

- Mappers
  - Small program (typically), distributed across the cluster, local to data
  - Handed a *portion* of the input data (called a split)
  - Each mapper parses, filters, or transforms its input
  - Produces grouped **<key,value>** pairs



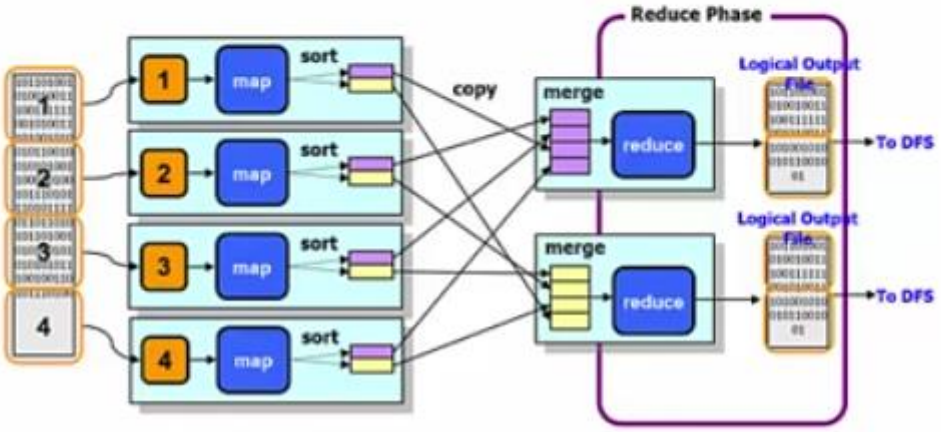
MapReduce – Shuffle Phase

- The output of each mapper is locally grouped together by **key**
- One node is chosen to process data for each unique **key**
- All of this movement (shuffle) of data is transparently orchestrated by MapReduce



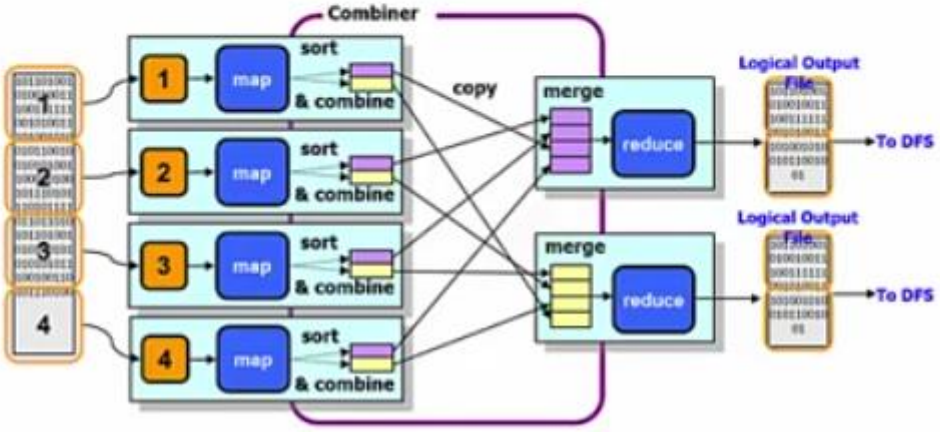
MapReduce – Reduce Phase

- Reducers
  - Small programs (typically) that aggregate all of the `values` for the `key` that they are responsible for
  - Each reducer writes output to its own file



MapReduce – Combiner (Optional)

- The data that will go to *each* reduce node is sorted and merged before going to the reduce node – pre-doing some of the work of the receiving reduce node in order to minimize network traffic between map and reduce nodes



In this video we will look at the phases of MapReduce.
The two major phases are Map and Reduce. But we will see that there are two other, minor phases. Here we see listed the four phases: Map,Shuffle,Reduce,
and Combiner The Combiner phase is optional, and fits just
before the Shuffle phase. But we will look at the four in the listed order for your better understanding. This slide provides an overview of the complete
MapReduce v1 process. File blocks (stored on different DataNodes)
in HDFS are read and processed by Map tasks running on the DataNodes where the blocks of data are stored. The output of the Map tasks are shuffled,

then sent to the Reducer tasks (one output file from each Mapper task to each of the Reducer tasks) — the files exchanged here are not replicated and are temporarily stored local to the Mapper node. The Reducers produces the output and that output is stored in HDFS, with one file for each Reducer, and replicated.

On a previous slide you saw something to the effect that "if you write your programs in a special way" the programs can be brought to the data. This special way is called MapReduce, and involves breaking your program down into two discrete parts: Map and Reduce.

Let's look at the Map Phase first. A mapper is typically a relatively small program with a simple task: it is responsible for reading a portion of the input data — one block of one file — interpreting, filtering or transforming the data as necessary and then finally producing a stream of <key, value> pairs. What these keys and values are doesn't really matter for this discussion, but just keep in mind that these <key, value> pairs don't have to be simple values. They can be as large and as complex as you need for the job at hand. As shown in the diagram, the MapReduce environment automatically takes care of your small "map" program (the blue boxes on the left) and pushing that program out to every machine that has a block of the file that you will be processing. This means that the bigger the file, the bigger the cluster, more mappers get involved in processing the data! That's a pretty powerful idea.

This next phase is called Shuffle and is orchestrated behind the scenes by the underlying MapReduce logic embedded in Hadoop. The idea here is that all of the data that is being emitted from the mappers is first locally grouped by the <key> that your program chose, and then for each unique key, a node is chosen to process all of the values from all of the mappers for that key. For example, let say you used U.S. state (e.g. "MA", "AK", "NY", "CA", etc.) as the key of your data, then one machine would be chosen to send all of the California data to, one all of the New York data, and so on. Each machine would be responsible for processing the data for its selected state or states. In the picture above, Hadoop has decided that only two sets of keys (yellow and purple) are needed, but keep in mind that there may be many, many records with the same key (or set of keys) coming from a given mapper. Reducers are the last major part of the picture. Again, reducers are small programs (well, typically small) that are responsible for sorting and/or, in particular, aggregating or reducing over the values associated with the key (or keys) assigned to the reducer node. Just like with mappers, when a larger number of reducers are used, the more parallelism that can be provided.

Once each reducer has completed whatever it is assigned to do – maybe, for example, add up the total sales for the state (or states) it was assigned – it, in turn, emits key/value pairs that get written to HDFS disk and, in turn, be used as the input to, perhaps, another MapReduce job. In an extremely over-simplified nutshell, that is MapReduce. At the same time as the sort is done during the Shuffle work on the Mapper node, an optional Combiner function may be applied. For each key set, all key/values with that key set will be sent to the same Reducer node — that is the purpose of the Shuffle phase. Rather than sending multiple key/value pairs with the same key value to the Reducer node, the values are combined into one key/value pair. This is only possible where the reduce function is additive (that is, does not lose information when combined). Since only one key/value pair is sent, the file transferred from Mapper node to Reducer node is smaller and network traffic is minimalized.

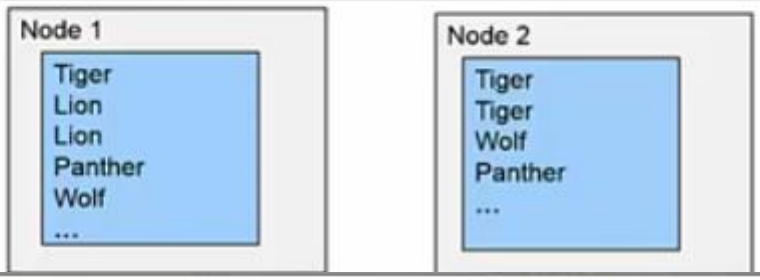
This is the end of this video. In the next video we will look at an example program, wordcount.

WORDCUNT PROGRAM EXAMPLE

WordCount example

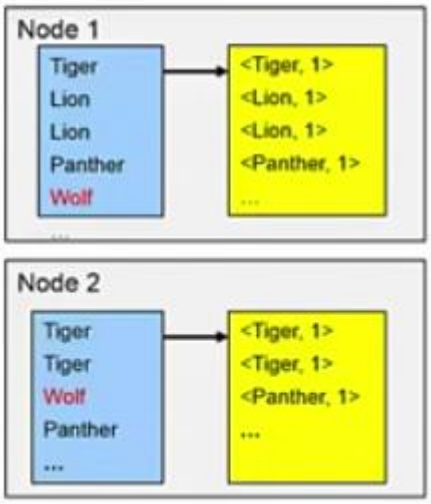
- In this example we have a list of animal names
  - MapReduce can automatically split files on line breaks
  - Our file has been split into two blocks on two nodes
- We want to count how often each big cat is mentioned.
  In SQL that would be:

```
SELECT COUNT(NAME) FROM animcls
   WHERE name IN ("Tiger", "Lion", …)
   GROUP BY name;
```



Map Task

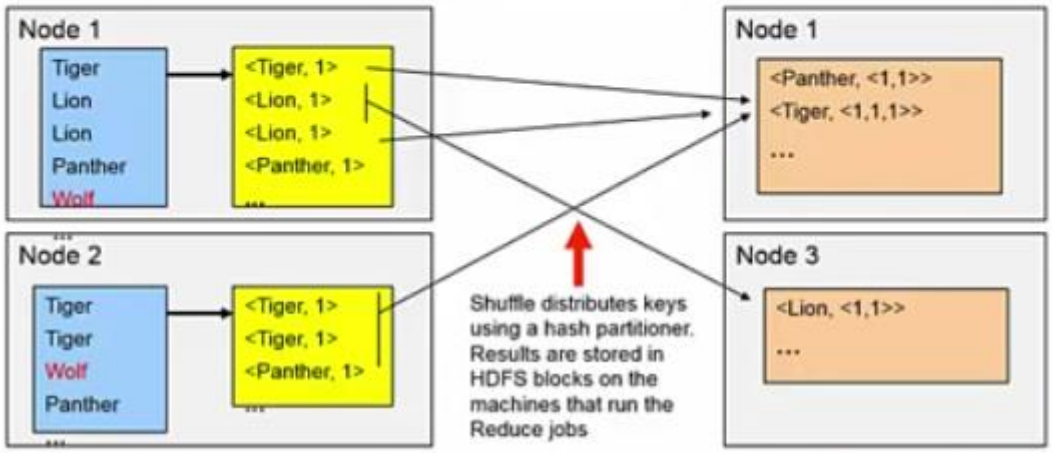
- We have two requirements for our Map task
  - Filter out the non big-cat rows
  - Prepare count by transforming to **<*Text*(name), *Integer*(1)>**



The Map Tasks are executed locally on each split

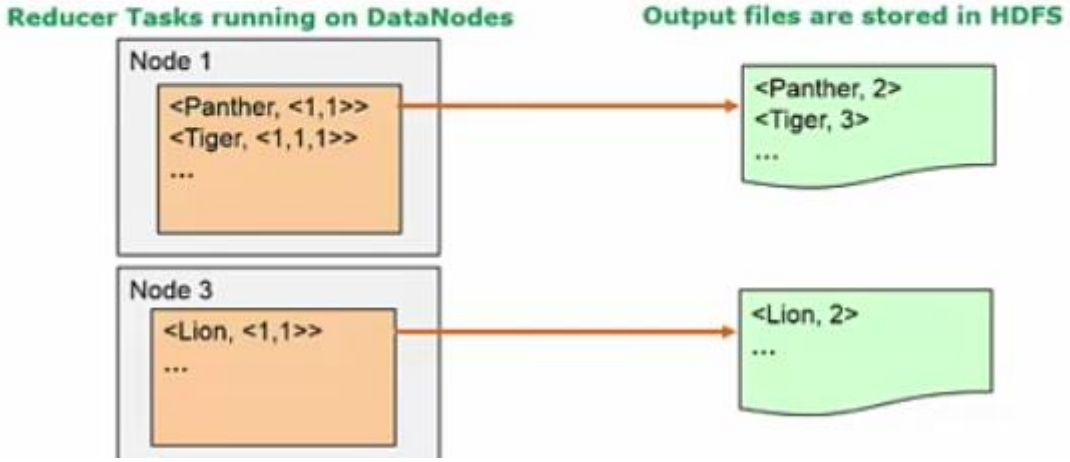
Shuffle

- Shuffle moves all values of one key to the same target node
- Distributed by a Partitioner Class (normally hash distribution)
- Reduce Tasks can run on any node -- here on Nodes 1 and 3
  - The number of Map and Reduce tasks do not need to be identical
  - Differences are handled by the hash partitioner



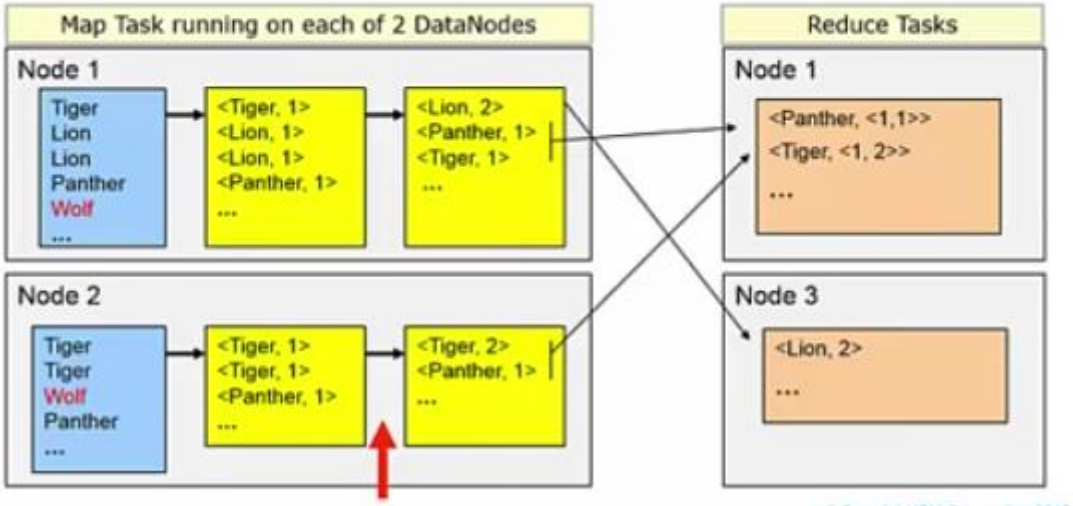
Reduce

- The reduce task computes aggregated values for each key
  - Normally the output is written to the DFS
  - Default is one output part-file per Reduce task
  - Reduce tasks aggregate all values of a specific key — our case, the count of the particular animal type



Optional: Combiner

- For performance, a pre-aggregate in the Map task can be helpful
- Reduces the amount of data sent over the network
  - Also reduces Merge effort, since data is premerged in Map
  - Done in the Map task, before Shuffle



Source code for WordCount.java

```
1. package org.myorg;
2.
3. import java.io.IOException;
4. import java.util.*;
5.
6. import org.apache.hadoop.fs.Path;
7. import org.apache.hadoop.conf.*;
8. import org.apache.hadoop.io.*;
9. import org.apache.hadoop.mapred.*;
10. import org.apache.hadoop.util.*;
11.
12. public class WordCount {
13.
14.   public static class Map extends MapReduceBase implements Mapper<LongWritable, Text,
          Text, IntWritable> {
15.     private final static IntWritable one = new IntWritable(1);
16.     private Text word = new Text();
17.
18.     public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable>
              output,Reporter reporter) throws IOException {
19.       String line = value.toString();
20.       StringTokenizer tokenizer = new StringTokenizer(line);
21.       while (tokenizer.hasMoreTokens()) {
22.           word.set(tokenizer.nextToken());
23.           output.collect(word, one);
24.       }
25.     }
26.   }
27.
```

Source code for WordCount.java (2 of 3)

```
28.   public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable,
          Text, IntWritable> {
29.     public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text,
          IntWritable> output, Reporter reporter) throws IOException {
30.       int sum = 0;
31.       while (values.hasNext()) {
32.         sum += values.next().get();
33.       }
34.       output.collect(key, new IntWritable(sum));
35.     }
36.   }
37.
```

Source code for WordCount.java (3 of 3)

```
38. public static void main(String[] args) throws Exception {
39.     JobConf conf = new JobConf(WordCount.class);
40.     conf.setJobName("wordcount");
41.
42.     conf.setOutputKeyClass(Text.class);
43.     conf.setOutputValueClass(IntWritable.class);
44.
45.     conf.setMapperClass(Map.class);
46.     conf.setCombinerClass(Reduce.class);
47.     conf.setReducerClass(Reduce.class);
48.
49.     conf.setInputFormat(TextInputFormat.class);
50.     conf.setOutputFormat(TextOutputFormat.class);
51.
52.     FileInputFormat.setInputPaths(conf, new Path(args[0]));
53.     FileOutputFormat.setOutputPath(conf, new Path(args[1]));
54.
55.     JobClient.runJob(conf);
57.   }
58. }
```

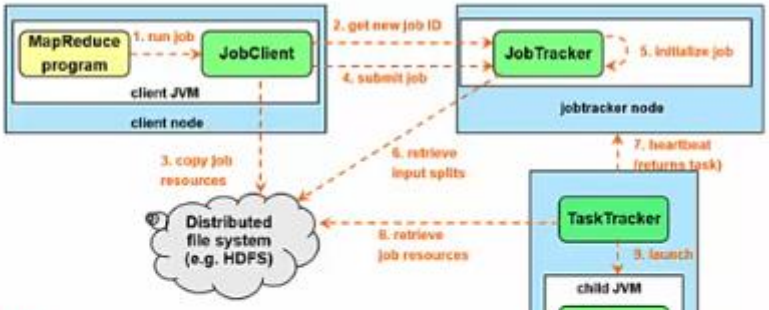
We will now look at the standard wordcount example program that comes as a test program with MapReduce. Over the next set of slides in this video
we will look at the wordcount program under MapReduce v1. Here, in this video, we will show wordcount under MR v1 – later we will discuss how the running job differs under YARN (including the similarities). We also look at the Java code shortly for
the Java version of the wordcount program to foster a brief discussion of the major parts of the program from a code perspective, thereby illustrating Map, Combiner, and Reduce
phases. We have a file, that for simplicity here,
has two blocks (or “splits) of data, that lists animal names.
There is one animal per line in the files. We want to count the number of mentions of each particular animal, but we are interested only in members of the cat family.
Since the blocks are held on different nodes, software running on the individual nodes process
the blocks separately. If we were using SQL — which we are not
— the SQL would be as shown. Slide 4 The Map step shown on this slide does the following processing: Each Map node reads its own “split” (or
block) of data The information required — in this case,
the names of animals — is extracted from each record — in this case, one line = one record Data is filtered — we keep only the names
of cat family animals (“Wolf” is eliminated) Key-value pairs are created — n this case,
key = animal & value = 1 The key-value pairs are accumulated into locally
stored files on the individual nodes where the Map task is being executed
Shuffle distributes the key-value pairs to the nodes where the Reducer task will run.
Each Mapper task produces one file for each Reducer task. A hash function running on the Mapper node determines which Reducer task will receive any particular key-value pair.
All key-value pairs with a particular key will be sent to the same Reducer task.
Reduce tasks can run on any node, either different from the set of nodes where the Map task run
or on the same DataNodes. Here we show that Node 1 is used for one Reduce task, but a new node, Node 3, is used for a second Reduce node.
There is no relation between the number of Map tasks — which is, generally, one node for each block of the file(s) begin read — and the number of Reduce tasks.
Commonly the number of Reduce tasks is smaller than the number of Map tasks.
Note that there are two Reducer tasks running — these are running on two separate nodes, Nodes 1 and 3. The Reduce node takes the answers to all the

sub-problems and combines them in some way to get the output — the answer to the problem
we were originally trying to solve. In this case, the Reduce step shown on this
slide does the following processing: Each Reduce node processes the data sent to
it from the various Map nodes This data has been previously sorted (and
possibly partially merged by Combiner processing) The Reduce node aggregates the data — in
the case of wordcount, it sums up the counts received for each particular word — each
animal in this case One file is produced for each Reduce task
and that will is written to HDFS where the output blocks are automatically replicated
The Combiner step is optional. When it is used, it runs on a Mapper node
and preprocess the intermediate data files on the individual Mapper node which will be
sent to Reduce tasks. It pre-merges and pre-aggregates the data in the files that will be transmitted
to the Reduce tasks. The Combiner thus reduces the amount of data
that will be sent to the Reducer tasks — and that speeds up the processing as smaller files
need to be transmitted from the Mapper nodes to the Reducer nodes.
This is a slightly simplified version of WordCount.java for MapReduce 1. The full program is slightly
larger, and there are some recommended differences for compiling for MapReduce 2 with Hadoop
2. Code from the Hadoop classes is brought in
with the import statements at the top of this page of code. Like an iceberg, most of the
actual code executed at runtime is hidden from the programmer — it runs deep down
inside Hadoop itself. Our interest here is the Mapper class, Map.
This class reads the file — we will see the file name on the driver class that two
slides later is passed as arg[0], a string.
Each input record, or line, from the file is read as a string and the string is tokenized,
that is, broken into words separated by space(s). Note the following shortcomings of the standard
code: No lowercasing is done, thus The and the are
treated as separate words that will be counted separately
Any adjacent punctuation is appended to the word, thus “the and the will be counted
separately, and any word followed by punctuation, e.g., cow, is counted separately from cow
(the same word without trailing punctuation) We will see these shortcomings in the output.
But, note that this is the standard wordcount program and we are less interested in the
actual results but only in the process at this stage.
The program, wordcount is to Hadoop Java programs functions as the “Hello, world!” program
does to the C language. It is generally the first program that people experience when
coming to the new technology. The Reducer class, Reduce, is shown here.
The key-value pairs arrive at this class already sorted, courtesy of the core Hadoop classes
that we do not see.
Thus adjacent records will have the same key. While the key does not change, the values
are aggregated — in this case, summed with: sum += ... Slide
10 The driver routine, embedded in main, does
the following work: Sets the JobName for runtime
Sets the Mapper class to Map Sets the Reducer class to Reduce
Sets the Combiner class to Reduce Sets the input file to arg[0]
Sets the output directory to arg[1] Note that the Combiner runs on the Map task
and use the same code as the Reducer task. The names of the output files will be generated

inside the Hadoop code. This is the end of this video. In the next video we will look at a number of miscellaneous details that are applied in the MapReduce process.

MISCELLANEOS

How does Hadoop run MapReduce jobs?



- **Client:** submits MapReduce jobs
- **JobTracker:** coordinates the job run, breaks down the job to map and reduce tasks for each node to work on the cluster
- **TaskTracker:** execute the Map and Reduce task functions

Classes

- There are three main Java classes provided in Hadoop to read data in MapReduce:
  - `InputSplitter` dividing a file into splits
    - Splits are normally the block size but depends on number of requested Map tasks, whether any compression allows splitting, etc.
  - `RecordReader` takes a split and reads the files into records
    - For example, one record per line (**LineRecordReader**)
    - But note that a record can be spit across splits
  - `InputFormat` takes each record and transforms it into a <key, value> pair that is then passed to the Map task
- Lots of additional helper classes may be required to handle compression, etc.
  - For example, IBM BigInsights provides additional compression handlers for LZO compression, etc.

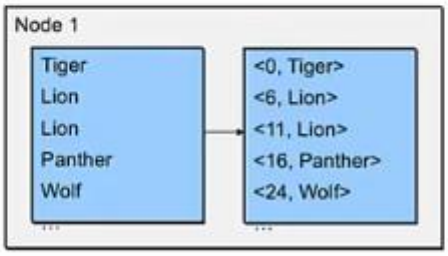
RecordReader

- Most of the time a Split will *not* happen at a block end
- Files are read into Records by the RecordReader class
  - Normally the RecordReader will start and stop at the split points.
- `LineRecordReader` will read over the end of the split till the line end.
  - HDFS will send the missing piece of the last record over the network
- Likewise LineRecordReader for Block2 will disregard the first incomplete line

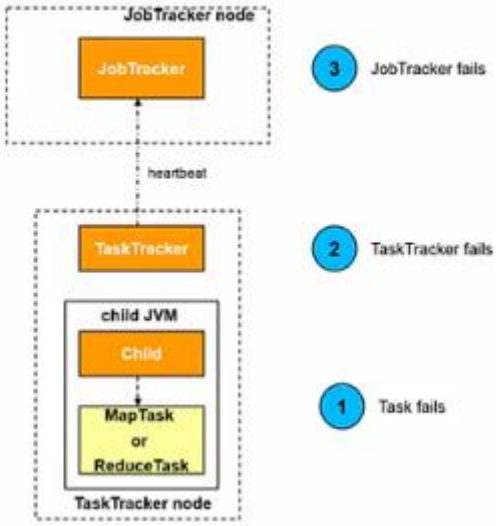
In this example RecordReader1 will not stop at "Pan" but will read on until the end of the line. Likewise RecordReader2 will ignore the first line

InputFormat

- MapReduce Tasks read files by defining an InputFormat class
  - Map tasks expect <key, value> pairs
- To read line-delimited textfiles Hadoop provides the **TextInputFormat** class
  - It returns one key, value pair per line in the text
  - The value is the content of the line
  - The key is the character offset to the new line character (end of line)



Fault tolerance



In this video we will cover a number of miscellaneous details:

How does Hadoop run a MapReduce job? What do the various [hidden] Java classes

do for us that we don’t see in our code A discussion of “splits”
RecordReader and split records Fault tolerance
The process of running a MapReduce job on Hadoop consists of 10 major steps.
1. The first step is the MapReduce program you've written tells the Job Client to run
a MapReduce job. 2. This sends a message to the JobTracker
which produces a unique ID for the job. 3. The Job Client copies job resources, such
as a jar file containing Java code you have written to implement the map or the reduce
task, to the shared file system, usually HDFS. 4. Once the resources are in HDFS, the Job
Client can tell the JobTracker to start the job.
5. The JobTracker does its own initialization for the job. It calculates how to split the
data so that it can send each "split" to a different mapper process to maximize throughput.
6. It retrieves these "input splits" from the distributed file system, not the data
itself. 7. The TaskTrackers are continually sending
heartbeat messages to the JobTracker. Now that the JobTracker has work for them, it
will return a map task or a reduce task as a response to the heartbeat.
8. The TaskTrackers need to obtain the code to execute, so they get it from the shared
file system. 9. Then they can launch a Java Virtual Machine
with a child process running in it, and 10. This child process runs your map code
or your reduce code. The results of the map operation are stored
on local disk without replication for a given TaskTracker node (and, thus, not in HDFS).
The results of the reduce operation are stored to HDFS with standard replication.
The InputSplitter, RecordSplitter, and InputFormat classes are found inside the Hadoop code.
InputSplitter divides a file into splits RecordReader takes a split and reads the files
as a sequence of records InputFormat takes each record and transforms
it into <key, value> pair(s) that is then passed to the Map task
Other helper classes are needed to support Java MapReduce programs. Some of these are
provided from inside the Hadoop code itself, but Hadoop distribution vendors and end-user
programmers can provide other classes that either override or supplement the standard
code. Thus IBM provides the LZO compressions algorithm
to supplement standard compression codecs (e.g., codecs for bzip2, etc).
Files are stored as “splits” or blocks, with a default size of 128 MB, in most versions
of Hadoop these days. Each split or block is processed on the actual
datanode, where it is stored, by a Mapper task running on that node.
Generally, because the file splits or blocks are a fixed size, most of the time a record
(or line) will end up broken across blocks. In the example illustrated, the record with
the word “Panther” is broken across blocks, “Pan” is in one block, and the continuation,
“ther” followed by newline, or record terminator, in the next block.
The LineRecordReader code inside Hadoop is designed to correct this situation so that
the complete word, “Panther,” is processed accurately on just one of these nodes.
The InputFormat class describes the input-specification for a Map-Reduce job. The Map-Reduce framework
relies on the InputFormat of the job to: Validate the input-specification of the job.
Split-up the input file(s) into logical InputSplits, each of which is then assigned to an individual
Mapper. Provide the RecordReader implementation to
be used to glean input records from the logical InputSplit for processing by the Mapper.
The default behavior of file-based InputFormat classes, typically sub-classes of FileInputFormat,
is to split the input into logical InputSplits based on the total size, in bytes, of the
input files. However, the FileSystem blocksize of the input files is treated as an upper

bound for input splits. A lower bound on the split size can be set by a configuration parameter
for Hadoop / MapReduce. Clearly, logical splits based on input-size
are insufficient for many applications since record boundaries must be respected. In such cases, the application has to also implement a RecordReader class that has the responsibility to respect record-boundaries and present a record-oriented view of the logical InputSplit to the processing. Slide 7
Now that you know what a successful MapReduce job looks like, let us see what happens when
something goes wrong. Failures can happen at the lowest level, the
task level (1), or at the TaskTracker level (2), or the JobTracker level (3).
The primary way that Hadoop achieves fault tolerance is through restarting tasks. Individual task nodes (TaskTrackers, plural) are in constant communication with the head node of the system,
called the JobTracker (singular). If a TaskTracker fails to communicate with the JobTracker for
a period of time (by default, 1 minute), the JobTracker will assume that the TaskTracker in question has crashed. The JobTracker knows which map and reduce tasks were assigned to
each TaskTracker. If the job is still in the mapping phase,
then other TaskTrackers will be asked to re-execute all map tasks previously run by the failed TaskTracker. If the job is in the reducing phase, then other TaskTrackers will re-execute all reduce tasks that were in progress on the failed TaskTracker.
Reduce tasks, once completed, have been written their output to HDFS. Thus, if a TaskTracker
has already completed two out of three reduce tasks assigned to it, only the third task must be executed elsewhere. Map tasks are slightly more complicated: even if a node has completed ten map tasks, the reducers may not have all copied their inputs from the output of those map tasks. If a node has crashed, then its mapper outputs are inaccessible.
So any already-completed map tasks must be re-executed to make their results available to the rest of the reducing machines. All of this is handled automatically by the Hadoop platform. This fault tolerance underscores the need
for program execution to be side-effect free. If Mappers and Reducers had individual identities
and communicated with one another or the outside world, then restarting a task would require
the other nodes to communicate with the new instances of the map and reduce tasks, and the re-executed tasks would need to reestablish their intermediate state. This process is notoriously complicated and error-prone in the general case.
MapReduce simplifies this problem drastically by eliminating task identities or the ability for task partitions to communicate with one another. An individual task sees only its own direct inputs and knows only its own outputs, to make this failure and restart process clean
and dependable. You should now do Hands-On Lab #1.
You will need to have downloaded the IBM BigInsights QuickStart VMware image v4.
In the lab, you will start the various services and run a simple MapReduce job to give you a chance to see what you have learned, in action.
After the Hands-On Lab, continue to section #2 of this course, where we will look at issues with and limitations of Hadoop v1 and MapReduce v1.

MODULE 2 – LIMITATION OF HADOOP V1 VS MAPREDUCE V1

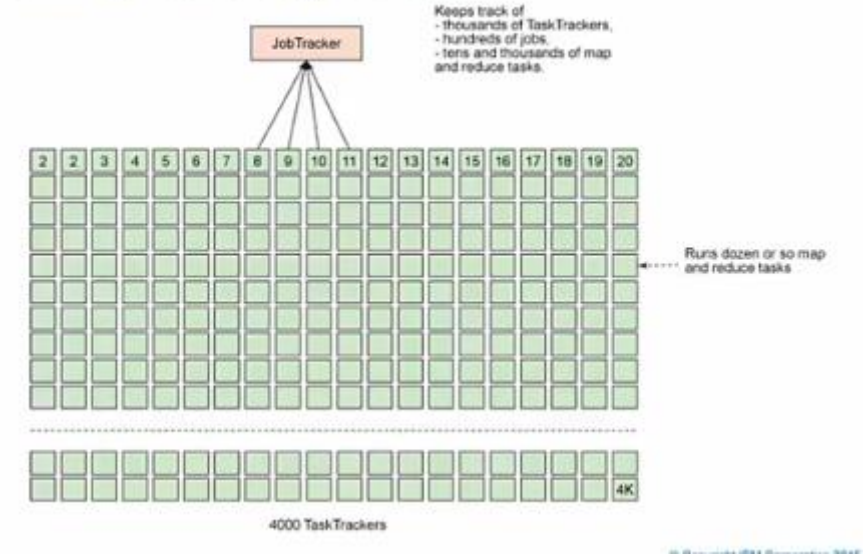
Issues with the original MapReduce paradigm

- Centralized handling of job control flow
- Tight coupling of a specific programming model with the resource management infrastructure
- Hadoop is now being used for all kinds of tasks beyond its original design

Limitations of classic MapReduce (MRv1)

- The most serious limitations of classic MapReduce are:
  - Scalability
  - Resource utilization
  - Support of workloads different from MapReduce
- In the MapReduce framework, the job execution is controlled by two types of processes:
  - A single master process called *JobTracker*, which coordinates all jobs running on the cluster and assigns map and reduce tasks to run on the TaskTrackers
  - A number of subordinate processes called *TaskTrackers*, which run assigned tasks and periodically report the progress to the JobTracker

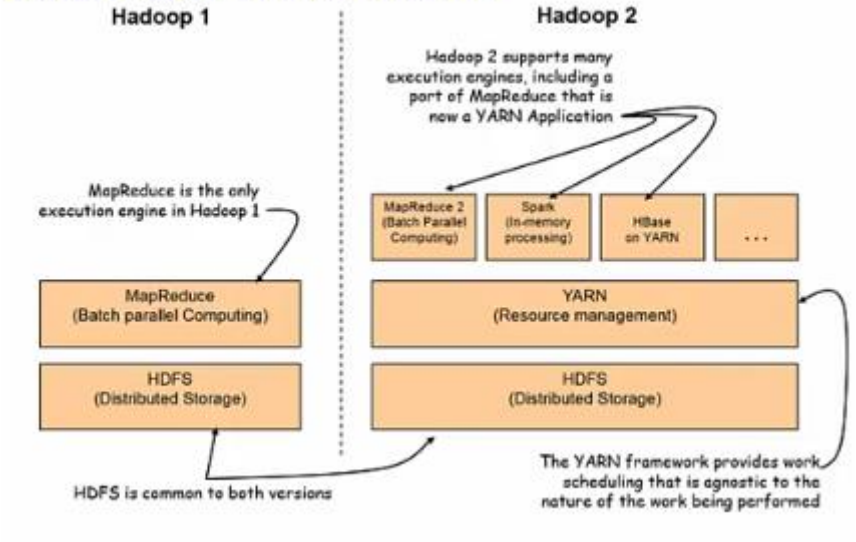
Scalability in MRv1: Busy JobTracker



YARN overhauls MRv1

- MapReduce has undergone a complete overhaul with YARN, splitting up the two major functionalities of JobTracker (resource management and job scheduling/monitoring) into separate daemons
- **ResourceManager (*RM*)**
  - The global ResourceManager and per-node slave, the NodeManager (*NM*), form the data-computation framework
  - The ResourceManager is the ultimate authority that arbitrates resources among all the applications in the system
- **ApplicationMaster (*AM*)**
  - The per-application ApplicationMaster is, in effect, a framework specific library and is tasked with negotiating resources from the ResourceManager and working with the NodeManager(s) to execute and monitor the tasks
  - An application is either a single job in the classical sense of Map-Reduce jobs or a directed acyclic graph (DAG) of jobs

Hadoop 1 and 2 architectures compared



The original Hadoop and MapReduce (v1) had limitations, and a number of issues have surfaced over time. We will look at what these were, in preparation for looking at the differences and changes introduced with Hadoop 2 and MapReduce v2.

The first issue to note is that in the original paradigm: there is only one JobTracker. Thus job flow control is centralized, limiting performance at scaled-up operations

Since the time when Hadoop and MapReduce were first made available, Hadoop has been called upon for a range of tasks that go beyond the original design.

We will study these and other issues in more detail in the upcoming slides.

This topic is well discussed in an article on an article on IBM DeveloperWorks:

Introduction to YARN http://www.ibm.com/developerworks/library/bd-yarn-intro

The most serious limitations of classic MapReduce are:

Scalability Resourceutilization

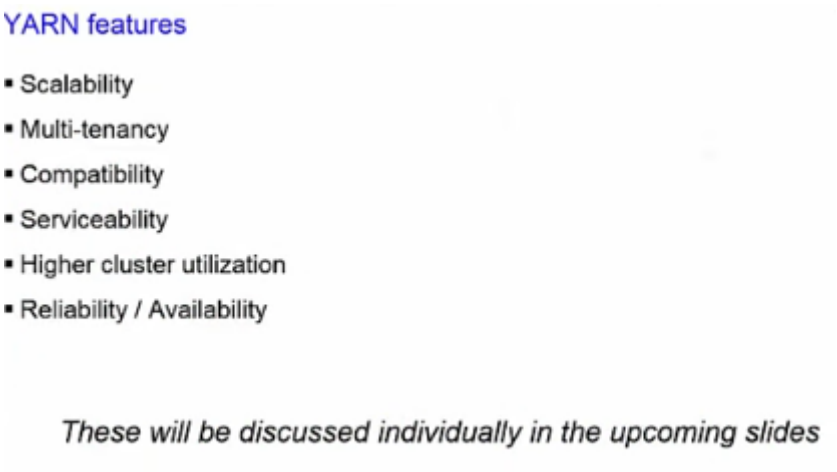
Support of workloads different from MapReduce Let's go into these is some detail

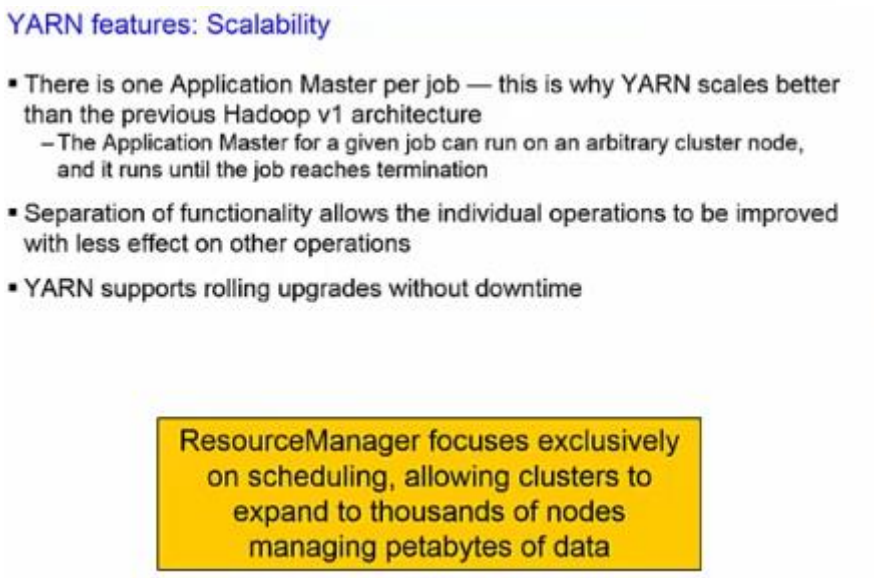
In Hadoop MapReduce v1, the JobTracker is charged with two distinct responsibilities:

Management of computational resources in the cluster, which involves maintaining the list of live nodes, the list of available and occupied map and reduce slots, and allocating the available slots to appropriate jobs and tasks according to selected scheduling policy, and

Coordination of all tasks running on a cluster, which involves instructing TaskTrackers to start map and reduce tasks, monitoring the execution of the tasks, restarting failed tasks, speculatively running slow tasks, calculating total values of job counters, and more
The large number of responsibilities given to a single process caused significant scalability issues, especially on a larger cluster where the JobTracker had to constantly keep track of thousands of TaskTrackers, hundreds of jobs, and tens of thousands of map and reduce tasks. The image on the slide illustrates the issue.
On the other hand, the TaskTrackers — running on the individual worker, data nodes — have
an easy time, usually running only a dozen or so map or reduce tasks each that were assigned
to them by the hard- working JobTracker. The fundamental idea behind YARN / MRv2 is to split up the two major functionalities of the JobTracker, resource management and job scheduling/monitoring, into separate daemons. The idea is to have a global ResourceManager
(RM) and per-application Application Masterx (AM). An application is either a single job in the classical sense of Map-Reduce jobs or a directed acyclic graph (aka DAG) of jobs.
The ResourceManager and per-node slave, the NodeManager (NM), form the data-computation
framework. The ResourceManager is the ultimate authority that arbitrates resources among all the applications in the system. A per-application ApplicationMaster is, in effect, a framework-specific library and is tasked with negotiating resources from the ResourceManager and working with the NodeManager(s) to execute and monitor the tasks.
The ResourceManager itself has two main components: Scheduler and ApplicationsManager.
The Scheduler is responsible for allocating resources to the various running applications subject to familiar constraints of capacities, queues etc. The Scheduler is a pure scheduler in the sense that it performs neither monitoring nor tracking of status for the application. Also, it offers no guarantees about restarting failed tasks either due to application failure or hardware failure. The Scheduler performs its scheduling function based on the resource requirements of the applications; it does so based on the abstract notion of a resource Container which incorporates elements such as memory, cpu, disk, network etc. In the first (current) version, only memory is supported. The Scheduler itself has a pluggable policy-plug-in,
which is responsible for partitioning the cluster resources among the various queues, applications etc. The current Map-Reduce schedulers such as the CapacityScheduler and the FairScheduler
are examples of the plug-in. For instance, the CapacityScheduler supports hierarchical queues to allow for more predictable sharing of cluster resources
The ApplicationsManager is responsible for accepting job-submissions, negotiating the first container for executing the application-specific ApplicationMaster and it provides the service
for restarting the ApplicationMaster container on failure.
The NodeManager is the per-machine framework agent that is responsible for containers, monitoring their resource usage (cpu, memory, disk, network), and reporting the same to the ResourceManager/Scheduler. The per-application ApplicationMaster has
the responsibility of negotiating appropriate resource containers from the Scheduler, tracking
their status and monitoring for progress. MRV2 maintains API compatibility with previous stable releases (that is, Hadoop v1). This means that all Map-Reduce jobs should still run unchanged on top of MRv2 with just a recompile. In Hadoop v1, MapReduce is the only execution

engine. In Hadoop v2, the YARN framework provides
work scheduling that is indifferent to the nature of the work being performed. It manages resources at a high level. The execution engine is now separate.
Hadoop 2 supports a number of execution engines, including a port of MapReduce that is now
a YARN Application, but also other execution engines, e.g.,
• Spark, an in-memory processing engine • HBase on YARN, and
• others, as we will see shortly That ends this video. The next video will
look at YARN features that solve the main problems seen with Hadoop v1 and MapReduce v1.





YARN features: Multi-Tenancy

- YARN allows multiple access engines (either open-source or proprietary) to use Hadoop as the common standard for batch, interactive, and real-time engines that can simultaneously access the same data sets
- YARN uses a shared pool of nodes for all jobs
- YARN allows the allocation of Hadoop clusters of fixed size from the shared pool

Multi-tenant data processing improves an enterprise's return on its Hadoop investment

YARN features: Compatibility

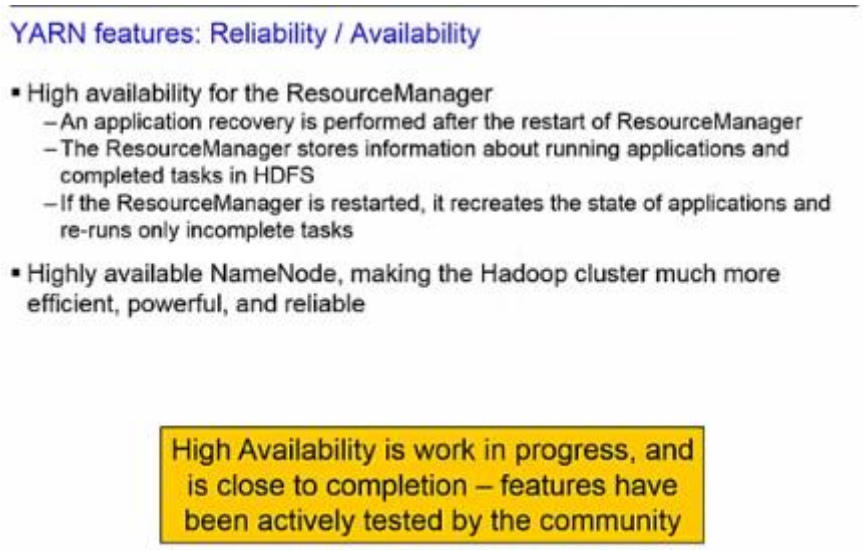
- To the end user (a developer, not an administrator), the changes are almost invisible
- Possible to run unmodified MapReduce jobs using the same MapReduce API and CLI
  - May require a recompile

There is no reason *not* to migrate from MRv1 to YARN

YARN features: Higher cluster utilization

- Higher cluster utilization, whereby resources not used by one framework can be consumed by another
- The NodeManager is a more generic and efficient version of the TaskTracker.
  - Instead of having a fixed number of map and reduce slots, the NodeManager has a number of dynamically created resource containers
  - The size of a container depends upon the amount of resources assigned to it, such as memory, CPU, disk, and network IO

YARN's dynamic allocation of cluster resources improves utilization over the more static MapReduce rules used in early versions of Hadoop (v1)



In this video, we will look at YARN features that show why YARN is integral to the future of Hadoop. The YARN features that we will discuss on the upcoming slides are: Scalability Multi-tenancy Compatibility Serviceability Higher cluster utilization Reliability / Availability YARN lifts the scalability ceiling in Hadoop by splitting the roles of the JobTracker in Hadoop v1 into two processes: a ResourceManager that controls access to the clusters resources (memory, CPU, etc.), and an ApplicationManager (one per job) that controls task execution. Because of this, YARN can run on larger clusters than can MapReduce v1. MapReduce v1 hits scalability bottlenecks in the region of 4,000 nodes and 40,000 tasks, stemming from the fact that the JobTacker has to manage both jobs and tasks. YARN overcomes these limitations by virtue of its separation of ResourceManager from a per-job ApplicationMaster. This architecture is designed to scale up to 10,000 nodes and 100,000 tasks. In contrast to the JobTracker approach, each instance of an application has a dedicated ApplicationMaster, which runs for the duration of the application. This architectural model is actually closer to the original Google MapReduce paper, which describes how a master process is started to coordinate map and reduce tasks running on a set of workers. Multi-tenancy Multi-tenancy generally refers to a set of features that enable multiple business users and processes to share a common set of resources, such as an Apache Hadoop cluster, via policy, rather than physical separation, yet without negatively impacting service level agreements (SLA), violating security requirements, or even revealing the existence of each other party or job. What YARN does is essentially de-couple Hadoop workload management from resource management. This means that multiple applications can share a common infrastructure pool. While this idea is not new to many of us, it is new to Hadoop. Earlier versions of Hadoop consolidated both workload and resource management functions into a single JobTracker. This approach resulted in limitations for consumers hoping to run multiple applications on the same cluster infrastructure. To borrow from object-oriented programming

terminology, multi-tenancy is an “over-loaded” term. It means different things to different people depending on their orientation and context. To say a solution is multi-tenant is not helpful unless we are specific about the meaning. Some interpretations of multi-tenancy in Big Data environments are: Support for multiple concurrent Hadoop jobs Support for multiple lines of business on a shared infrastructure Support for multiple application workloads of different types (Hadoop and non-Hadoop) Provisions for security isolation between tenants Contract-oriented service-level guarantees for tenants Support for multiple versions of applications and application frameworks concurrently Organizations that are sophisticated in their view of multi-tenancy will need all of these capabilities and more. YARN promises to address some of these requirements, and does so in large measure. But, you will find in future releases of Hadoop that there will be other approaches that will be used to address other forms of and approaches to multi-tenancy. While an important technology, the world is not suffering from a shortage of resource managers. Some Hadoop providers (including IBM) support YARN, while others, for instance, support Apache Mesos. Compatibility

To ease the transition from Hadoop v1 to YARN, a major goal of YARN and the MapReduce framework implementation on top of YARN was to ensure that existing MapReduce applications that were programmed and compiled against previous MapReduce APIs (we’ll call these MRv1 applications) can continue to run with little or no modification on YARN (we can refer to these as MRv2 applications).

For the vast majority of users who use the Hadoop v1 MapReduce APIs, MapReduce on YARN ensures full binary compatibility. Thus existing applications can run on YARN directly without recompilation. You can use .jar files from your existing application that code against mapred APIs, and use bin/hadoop to submit them directly to YARN.

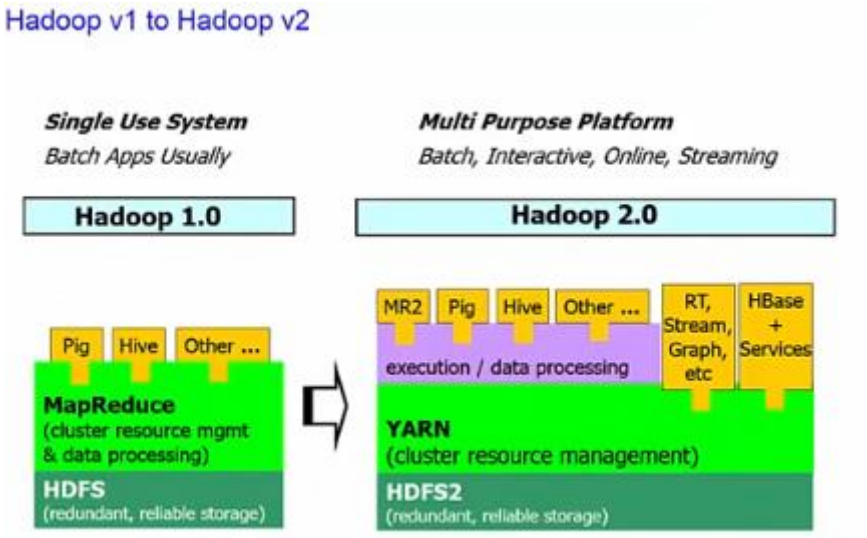
Unfortunately, however, it was difficult to ensure full binary compatibility to the existing applications that compiled against MRv1’s org.apache.hadoop.mapreduce APIs. These APIs have gone through many changes. For example, several classes stopped being abstract classes and changed to interfaces. Therefore, the YARN community compromised by only supporting source compatibility for org.apache.hadoop.mapreduce APIs. Existing applications that use MapReduce APIs are source- compatible and can run on YARN either with no changes, with simple recompilation against MRv2 .jar files that are shipped with Hadoop 2, or with minor updates.

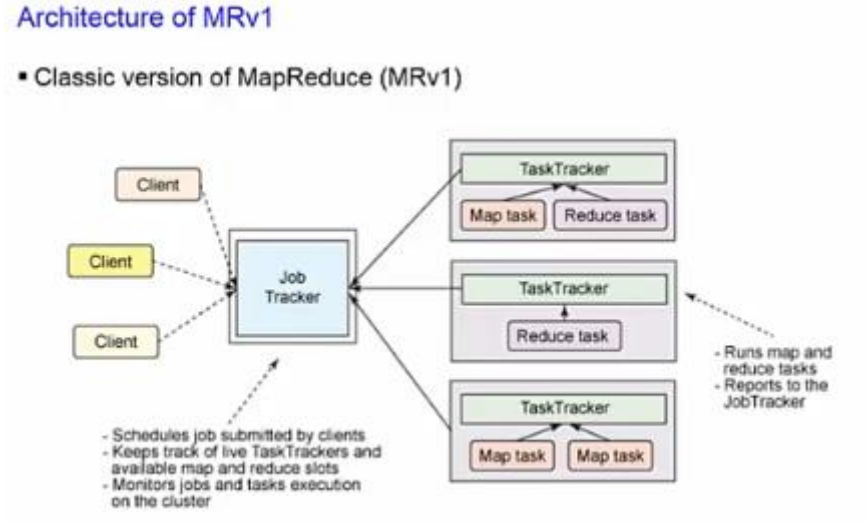
High cluster utilization The NodeManager is a more generic and efficient version of the TaskTracker found in Hadoop v1. Instead of having a fixed number of map and reduce slots, the NodeManager has a number of dynamically created resource containers.

The size of a container depends upon the amount of resources assigned to it, such as memory, CPU, disk, and network IO. Currently, only memory and CPU are supported (YARN-3). It is expected that cgroups might be used to control disk and network IO in the future. The number of containers on a node is a reflection of the configuration parameters used and the total amount of node resources (such as total CPUs and total memory) — outside the resources dedicated to the slave daemons and the OS

itself. Reliability / Availability
High availability is important, but is also work in progress. A number of features are being incorporated into each new version of Hadoop.
Importantly, a highly available NameNode — making the Hadoop cluster much more efficient, powerful,
and reliable — is the most important issue being worked on.
In summary, YARN has a number of major features. These are:
Multi-tenancy Cluster utilization
Scalability Compatibility
And the features are detailed on the slide. This ends this video and this section of the course. In the next two videos we will look at the
high-level architecture of YARN and how to run a job under YARN.

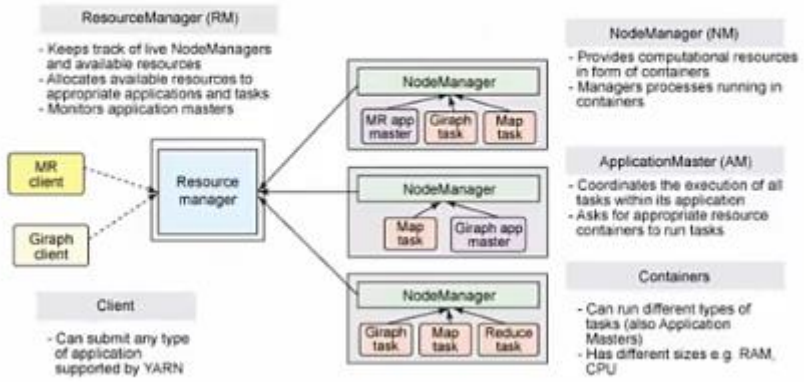
MODULE 3 – THE ARCHITECTURE OF YARN





YARN architecture

- High level architecture of YARN



Terminology changes from MRv1 to YARN

YARN terminology	Instead of MRv1 terminology
ResourceManager	Cluster Manager
ApplicationMaster (but dedicated and short-lived)	JobTracker
NodeManager	TaskTracker
Distributed Application	*One particular* MapReduce job
Container	Slot

YARN in BigInsights

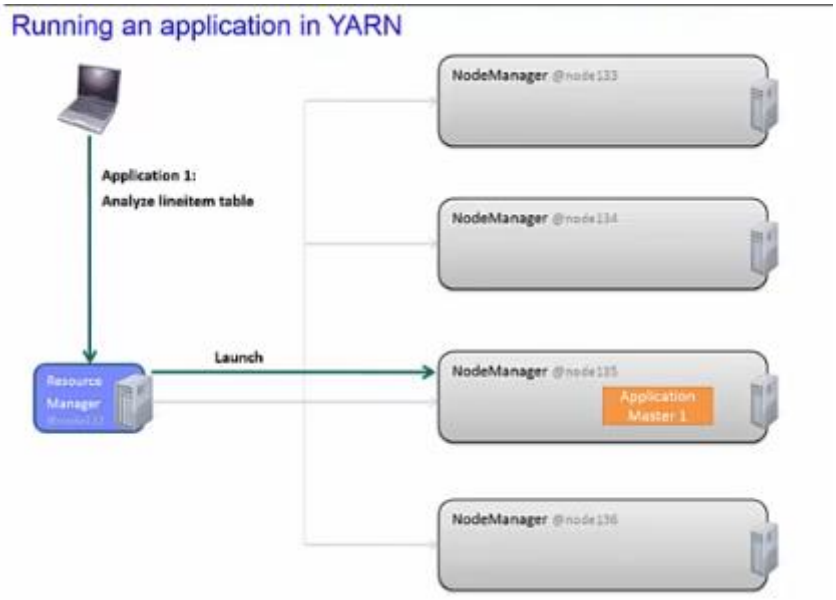
- Acronym for "Yet Another Resource Negotiator"
- New resource manager included in Hadoop 2.x and later
- De-couples Hadoop workload & resource management
- Introduces a general purpose application container
- Hadoop 2.2.0 includes the first GA version of YARN
- Most Hadoop vendors support YARN including IBM

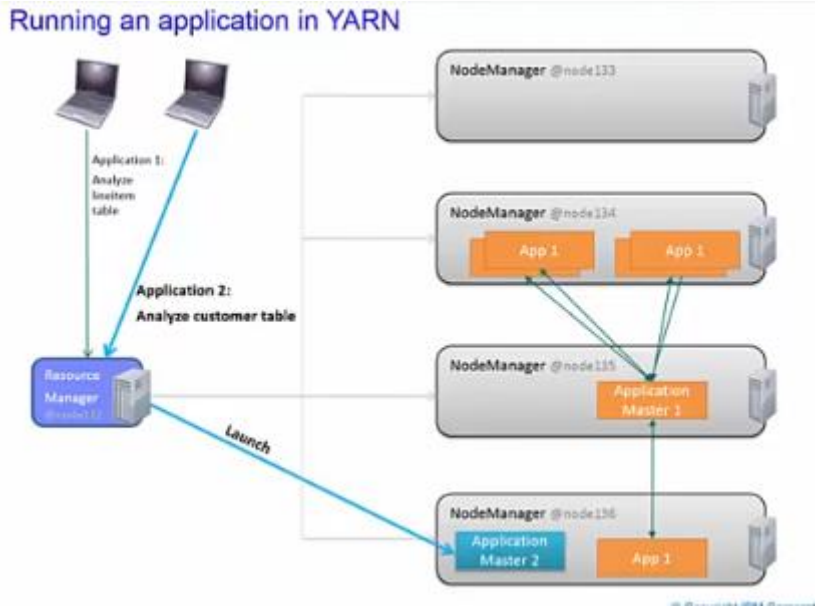
In the last section of this course, with two videos, we will look at the architecture of YARN. The most notable change from Hadoop v1 to Hadoop v2 is the separation of cluster & resource management from the execution & data processing environment. This allows for a new variety of application types to run, including, of course, MapReduce v2.

Note here, in the right side of the diagram, that execution is separated from cluster resource management. YARN provides the cluster resource management, allowing for a wide variety of

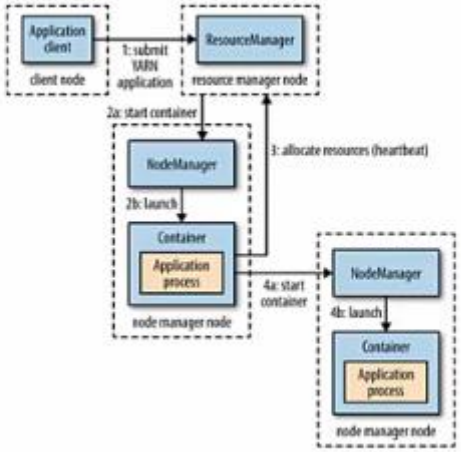
execution engines. We can see the effect most prominently with
the overall job control. In MapReduce v1 there is just one JobTracker
that is responsible for allocation of resources and task assignment to data nodes (as TaskTrackers).
The TaskTrackers provide ongoing monitoring (through a “heartbeat” sent back to the
JobTracter) as each job is run. The TaskTrackers constantly report back to the JobTracker on
the status of each running task. In the YARN architecture, a global ResourceManager
runs as a master daemon — usually on a dedicated machine — that arbitrates the available
cluster resources among various competing applications. The ResourceManager tracks how
many live nodes and resources are available in the cluster and then coordinates what applications
submitted by users should get these resources and when.
The ResourceManager is the single process that has this information so it can make its
allocation (or rather, scheduling) decisions in a shared, secure, and multi-tenant manner
(for instance, according to application priority, queue capacity, access control lists [ACLs],
data locality, etc.). When a user submits an application, an instance
of a lightweight process called the ApplicationMaster is started to coordinate the execution of
all tasks within the application. This includes monitoring tasks, restarting failed tasks,
speculatively running slow tasks, and calculating total values of application counters. These
responsibilities were previously assigned to the single JobTracker in the cluster for
all jobs. The ApplicationMaster and tasks that belong to its application run in resource
containers controlled by the NodeManagers. The NodeManager is a more generic and efficient
version of the TaskTracker. Instead of having a fixed number of map and
reduce slots, the NodeManager has a number of dynamically created resource containers.
The size of a container depends upon the amount of resources it contains and controls, such
as memory, CPU, disk, and network IO. Interestingly, the ApplicationMaster can run
any type of task inside a container. For example, the MapReduce ApplicationMaster requests a
container to launch a map or a reduce task, while the Giraph ApplicationMaster requests
a container to run a Giraph task. You can also implement a custom ApplicationMaster
that runs specific tasks and, in this way, invent a shiny new distributed application
framework that changes the big data world. I encourage you to read about Apache Twill,
which aims to make it easy to write distributed applications sitting on top of YARN.
In YARN, MapReduce is simply degraded to a role of just one possible distributed application
— but still a very popular and useful one —and is now called MRv2. MRv2 is simply
the re-implementation of the classical MapReduce engine, now called MRv1, to run on top of
YARN. Here are some terminology changes.
The MR v1 terminology is on the right, the new, YARN, terminology on the left. You have
the parallels: Cluster Manager becomes ResourceManager in
YARN The singular, but permanent, JobTracker of
MR1 become the ApplicationMaster in YARN (but it is dedicated to one job and thus
is short-lived) The TaskTracker is replaced by the NodeManager
in Hadoop v2 One particular MapReduce job in MR1 is now
a Distributed Application A Slot, which had to be either a Map slot
or a Reduce slot is replaced by a generic Container that can be used for many purposes.
Containers are the new way of describing a lot of virtual processing approaches these
days. YARN is a key component in the Open Data Platform
Initiative and hence in IBM BigInsights v4 BigInsights version 4 is currently based on
Hadoop 2.6.0. Other revisions are currently being tested for

the OPT initiative. Expect to hear about Hadoop 2.7, etc.
At a high level, we have with YARN a generic resource management environment that can be used as the basis for many different execution and processing engines. Some are illustrated here. We have now finished this video.
In the next, and final, video, you will look at various aspects of running a YARN application.

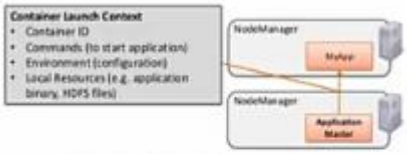




How YARN runs an application



Container Java command line



- Container JVM command (generally behind the scenes)
  - Launched by "yarn" user with /bin/bash

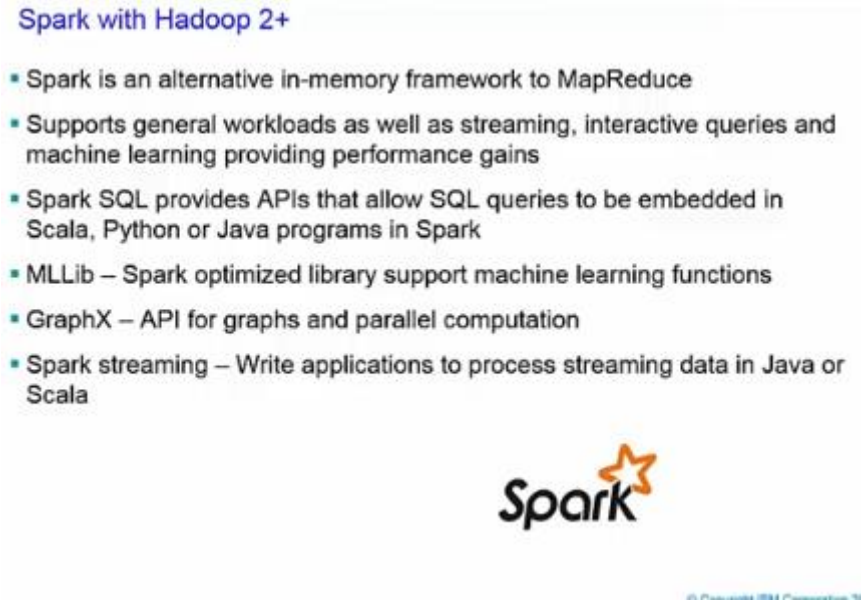
```
yarn  1251527 1199943  0 14:38 ?  00:00:00 /bin/bash -c
    /opt/ibm/biginsights/jdk/bin/java -Djava.net …
```

  - If you count "java" process ids (pids) running with the "yarn" user, you will see 2X

```
00:00:00 /bin/bash -c /opt/ibm/biginsights/jdk/bin/java
  00:00:00 /bin/bash -c /opt/ibm/biginsights/jdk/bin/java
   . . .
  00:07:48 /opt/ibm/biginsights/jdk/bin/java -Djava.net.pr
  00:00:11 /opt/ibm/biginsights/jdk/bin/java -Djava.net.pr
   . . .
```

Provisioning, Management, and Monitoring

- The **Apache Ambari project** is aimed at making Hadoop management simpler by developing software for provisioning, managing, and monitoring Apache Hadoop clusters. Ambari provides an intuitive, easy-to-use Hadoop management web UI backed by its RESTful APIs.
- Ambari enables System Administrators to:
  - Provision an Hadoop Cluster
    - Ambari provides a step-by-step wizard for installing Hadoop services across any number of hosts.
    - Ambari handles configuration of Hadoop services for the cluster.
  - Manage an Hadoop Cluster
    - Ambari provides central management for starting, stopping, and reconfiguring Hadoop services across the entire cluster.
  - Monitor an Hadoop Cluster
    - Ambari provides a dashboard for monitoring health and status of the Hadoop cluster.
    - Ambari leverages **Ganglia** for metrics collection.
    - Ambari leverages **Nagios** for system alerting and will send emails when your attention is needed (e.g., a node goes down, remaining disk space is low, etc).
- Ambari enables Application Developers and System Integrators to:
  - Easily integrate Hadoop provisioning, management, and monitoring capabilities to their own applications with the Ambari REST APIs



In this, your final video of this course, we will look at running an application with YARN. Here you see part of a large cluster. The nodes are numbered 133, 134, etc to suggest this.

An application — Application #1, since we will have another one shortly — connects to the cluster and wants to start a YARN job. The job is going to analyze line items from various transactions, e.g., a set of invoices. The ResourceManager is contacted.

The ResourceManager launches an ApplicationManager that will be responsible for this job. The ResourceManager is permanently running in this cluster; a new ApplicationManager is launched just for this job and will end when the job is complete.

The new ApplicationManager (#1 in this diagram) needs some resources. It sends a resource request to the ResourceManager. The ResourceManager replies by sending the IDs for a number of containers that have been allocated to this job. This reply message gives the Container IDs, and thereby implies where they containers are available.

From here on, the ApplicationManager works with, controls, and manages the containers until this job finishes, when this particular ApplicationMaster finishes as well.

Now let's see what happens when a second Application (#2) starts to run.

The client program contacts the ResourceManager which then launches a second and new ApplicationManager.

ApplicationManager #2 sends resource requests to the ResourceManager which responds with a list of Container IDs. After that, ApplicationMaster #2 works directly with the containers assigned to it. The sequence of steps is summarized numerically in the diagram shown here, To run an application on YARN, a client contacts the resource manager and asks it to run an application master process (step 1). The resource manager then finds a NodeManager that can launch the ApplicationMaster in a container (steps 2a and 2b). Precisely what the application master does once it is running depends on the application itself. It could simply run a computation in the container it is running in and return the result to the client. Or it could request more containers from the resource managers (step 3), and use them to run a distributed computation (steps 4a and 4b). This diagram is not for the faint hearted.

It shows some of the factors involved in the Container Launch Context.

The information passed includes: Container ID

Commands to start the application Environment, i.e., configuration information

Local resources to be used, e.g., application binary, HDFS file blocks, etc.

When a YARN process is launched, there are two process ids (pids) when you count the

processes in Linux. In this slide, we want to remind you of the
role of Ambari in Hadoop 2. With Hadoop 2 and YARN there is a greater
need to provision, manage, and monitor services because of the greater complexity of the Hadoop
2 environment. Beyond the scope of this course are other
services. An important service for a YARN environment
is Ambari Slider which provides a mechanism for dynamically changing requirements at run time for long running jobs. We would be remiss, if we did not mention
Apache Spark as Spark is a new, in-memory framework that is an alternative to MapReduce.
There are a number of courses in BigDataUniversity.com on Spark.
Now it is time for Lab Exercise 2 where we will run more complex MapReduce and YARN jobs,
including how to compile and run complete applications.
This now completes the video and the lecture part of this course. We hope that you enjoyed it and learned a lot.