Course objectives

- Describe the MapReduce model v1
- List the limitations of 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
- Compare YARN / Hadoop 2 / MR2 vs Hadoop 1 / MR1

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

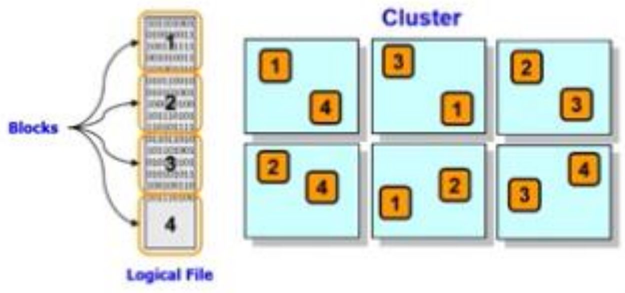
   Lab Exercise 2

Agenda

- **MapReduce Introduction**
- MapReduce Tasks
- WordCount Example
- Splits
- Execution

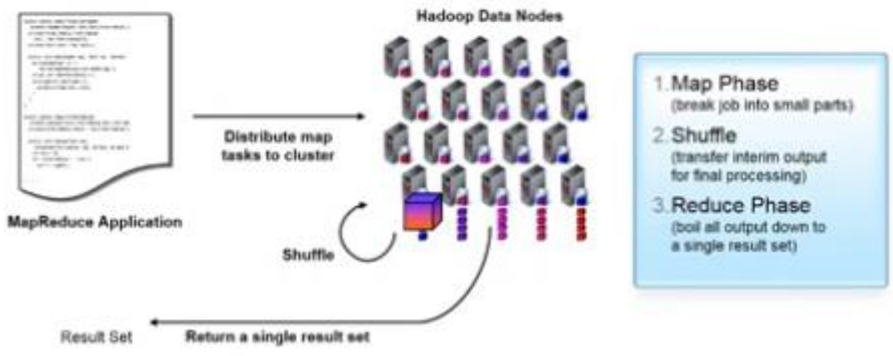
MapReduce – the Distributed File System

- 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



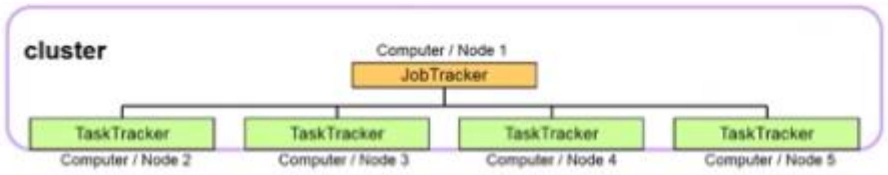
MapReduce v1 explained

- **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**


Hadoop Data Nodes
Distribute map tasks to cluster
MapReduce Application
Shuffle
Result Set
Return a single result set
1. Map Phase (break job into small parts)
2. Shuffle (transfer interim output for final processing)
3. Reduce Phase (boil all output down to a single result set)

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)

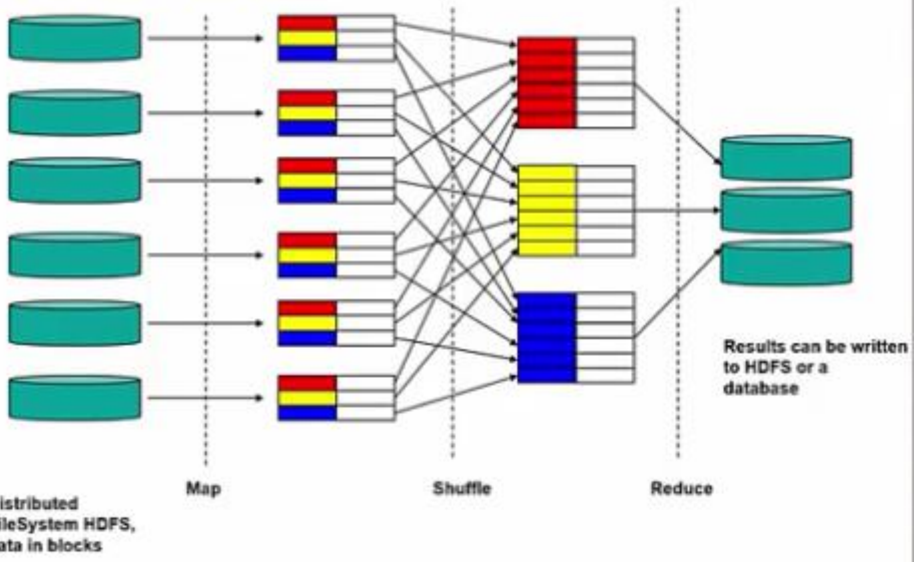
The MapReduce execution environments

- 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

Agenda

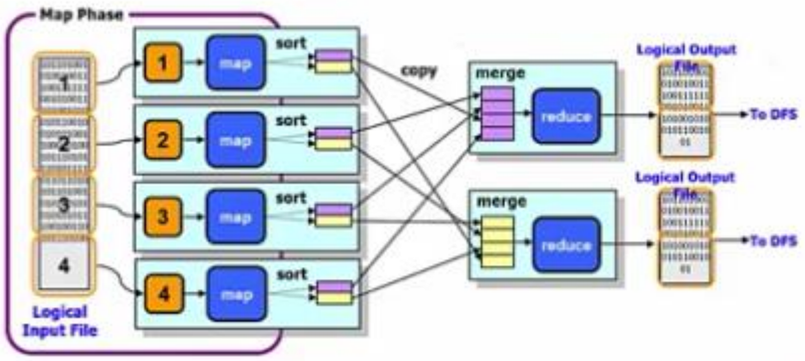
- MapReduce introduction
- **MapReduce phases**
  - **Map**
  - **Shuffle**
  - **Reduce**
  - **Combiner**
- WordCount example
- Splits
- Execution

MapReduce 1 overview


Results can be written to HDFS or a database
Distributed FileSystem HDFS, data in blocks
Map
Shuffle
Reduce

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



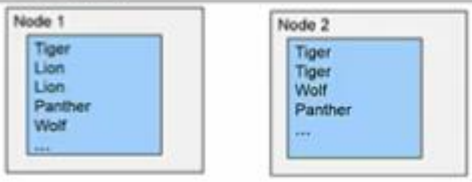
Agenda

- MapReduce introduction
- MapReduce phases
- **WordCount example**
- Splits
- Execution

WordCount example

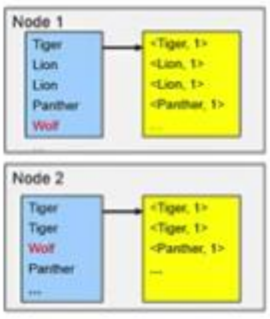
- In this example we have a list of animal names
  - MapReduce can automatically split files on line breaks
  - Our file has been split into two blocks on two nodes
- We want to count how often each big cat is mentioned. In SQL that would be:

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



Map Task

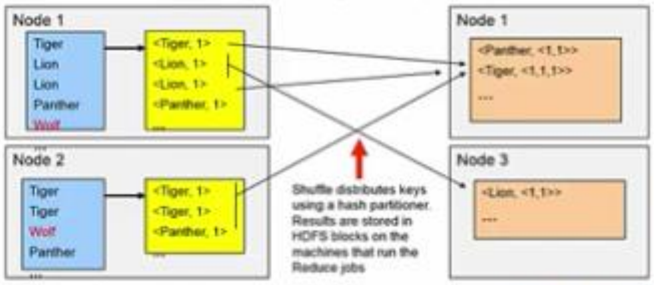
- We have two requirements for our Map task
  - Filter out the non big-cat rows
  - Prepare count by transforming to **<*Text*(name), *Integer*(1)>**



The Map Tasks are executed locally on each split

Shuffle

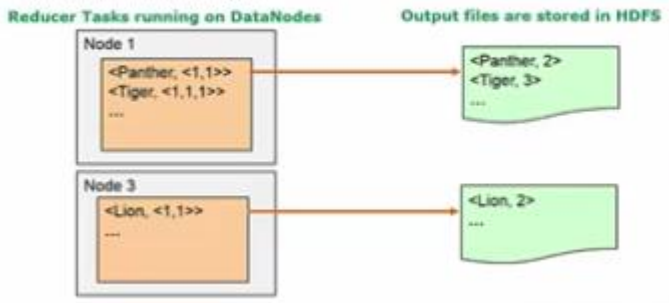
- Shuffle moves all values of one key to the same target node
- Distributed by a Partitioner Class (normally hash distribution)
- Reduce Tasks can run on any node -- here on Nodes 1 and 3
  - The number of Map and Reduce tasks do not need to be identical
  - Differences are handled by the hash partitioner



Shuffle distributes keys using a hash partitioner. Results are stored in HDFS blocks on the machines that run the Reduce jobs

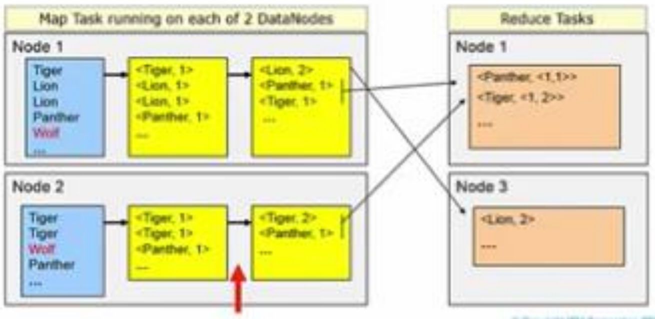
Reduce

- 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

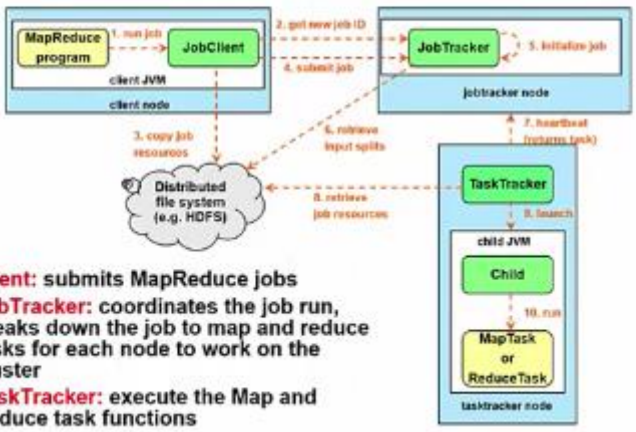
- For performance, a pre-aggregate in the Map task can be helpful
- Reduces the amount of data sent over the network
  - Also reduces Merge effort, since data is premerged in Map
  - Done in the Map task, before Shuffle



Source code for WordCount.java

```
package org.myorg;

import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapred.*;
import org.apache.hadoop.util.*;

public class WordCount {

  public static class Map extends MapReduceBase implements Mapper<LongWritable, Text,
        Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable>
              output,Reporter reporter) throws IOException {
      String line = value.toString();
      StringTokenizer tokenizer = new StringTokenizer(line);
      while (tokenizer.hasMoreTokens()) {
          word.set(tokenizer.nextToken());
          output.collect(word, one);
      }
    }
  }

  public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable,
        Text, IntWritable> {
    public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text,
        IntWritable> output, Reporter reporter) throws IOException {
      int sum = 0;
      while (values.hasNext()) {
        sum += values.next().get();
      }
      output.collect(key, new IntWritable(sum));
    }
  }

```

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

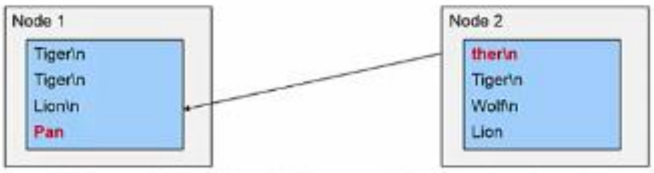
- There are three main Java classes provided in Hadoop to read data in MapReduce:
  - `InputSplitter` dividing a file into splits
    - Splits are normally the block size but depends on number of requested Map tasks, whether any compression allows splitting, etc.
  - `RecordReader` takes a split and 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.

Splits

- Files in MapReduce are stored in Blocks (128 MB)
- MapReduce divides data into fragments or **splits**.
  - One map task is executed on each split
- Most files have records with defined **split points**
  - Most common is the end of line character
- The `InputSplitter` class is responsible for taking a HDFS file and transforming it into splits.
  - Aim is to process as much data as possible locally

RecordReader

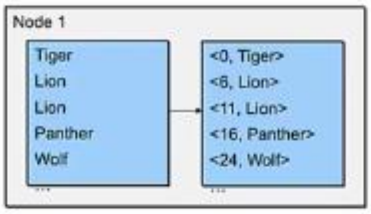
- Most of the time a Split will *not* happen at a block end
- Files are read into Records by the RecordReader class
  - Normally the RecordReader will start and stop at the split points.
- `LineRecordReader` will read over the end of the split till the line end.
  - HDFS will send the missing piece of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



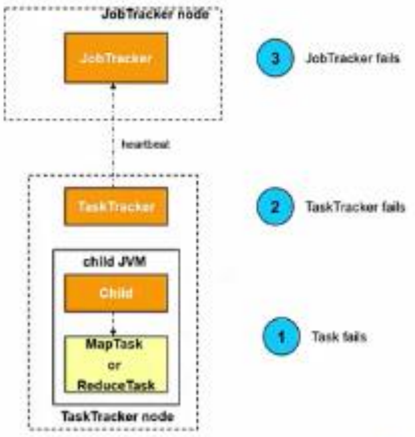
In this example RecordReader1 will not stop at "Pan" but will read on until the end of the line. Likewise RecordReader2 will ignore the first line

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



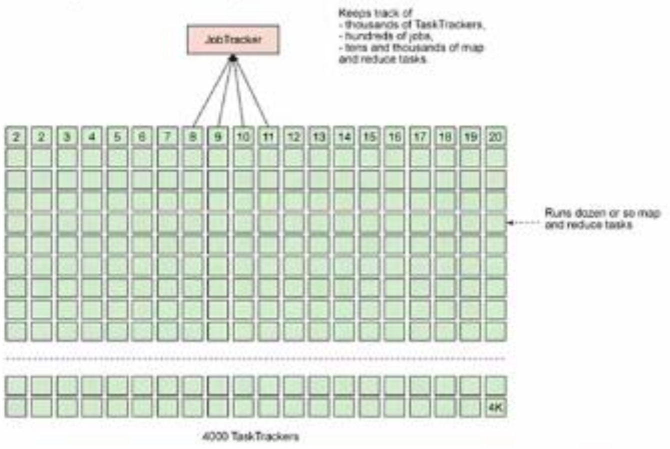
Issues with the original MapReduce paradigm

- 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

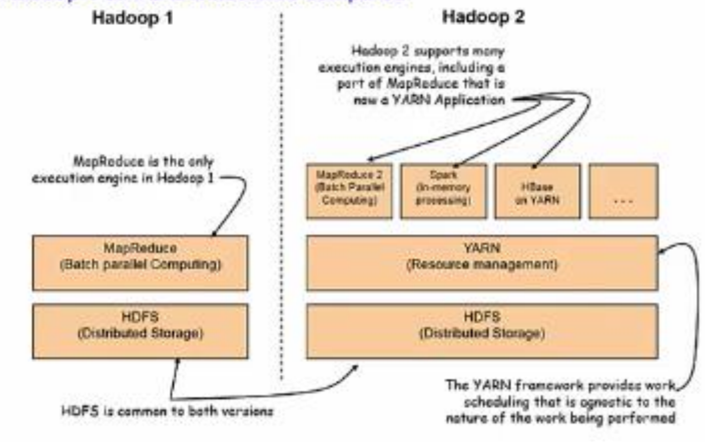
Scalability in MRv1: Busy JobTracker

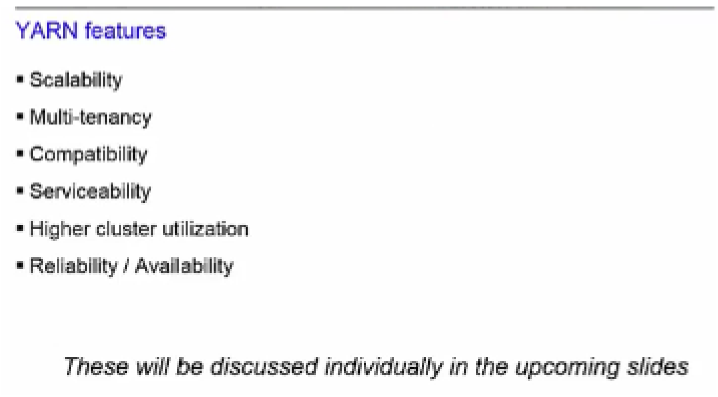

JobTracker
Keeps track of
- thousands of TaskTrackers,
- hundreds of jobs,
- tens and thousands of map and reduce tasks
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
Runs dozen or so map and reduce tasks
4K
4000 TaskTrackers

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

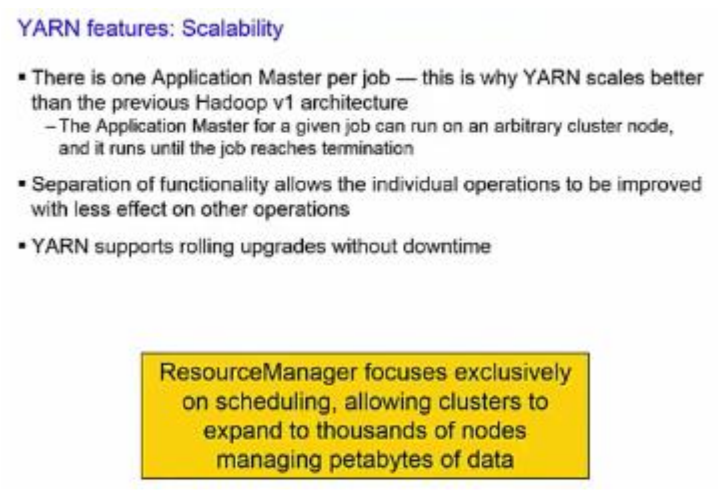

Hadoop 1
Hadoop 2
Hadoop 2 supports many execution engines, including a port of MapReduce that is now a YARN Application
MapReduce is the only execution engine in Hadoop 1
MapReduce 2 (Batch Parallel Computing)
Spark (In-memory processing)
HBase on YARN
...
MapReduce (Batch parallel Computing)
YARN (Resource management)
HDFS (Distributed Storage)
HDFS (Distributed Storage)
HDFS is common to both versions
The YARN framework provides work scheduling that is agnostic to the nature of the work being performed



YARN features

- Scalability
- Multi-tenancy
- Compatibility
- Serviceability
- Higher cluster utilization
- Reliability / Availability

These will be discussed individually in the upcoming slides



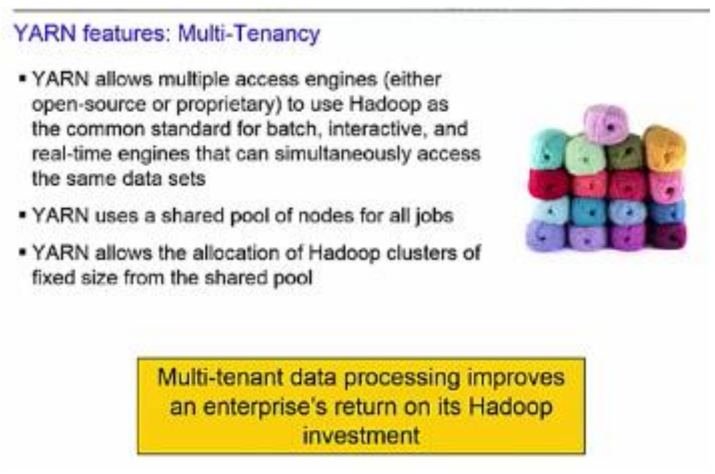
YARN features: Scalability

- There is one Application Master per job — this is why YARN scales better than the previous Hadoop v1 architecture
  - The Application Master for a given job can run on an arbitrary cluster node, and it runs until the job reaches termination
- Separation of functionality allows the individual operations to be improved with less effect on other operations
- YARN supports rolling upgrades without downtime

ResourceManager focuses exclusively on scheduling, allowing clusters to expand to thousands of nodes managing petabytes of data

MR1 Limit 4000 Nodes, 40000 Tasks

MR2 Limit 10000 Nodes, 100000 Tasks

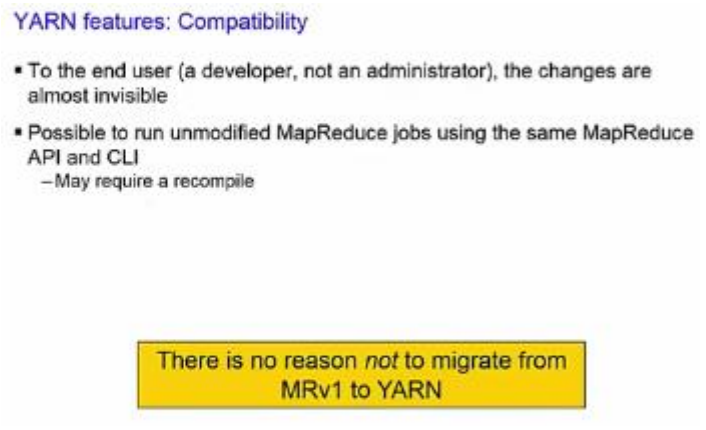


YARN features: Multi-Tenancy

- 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

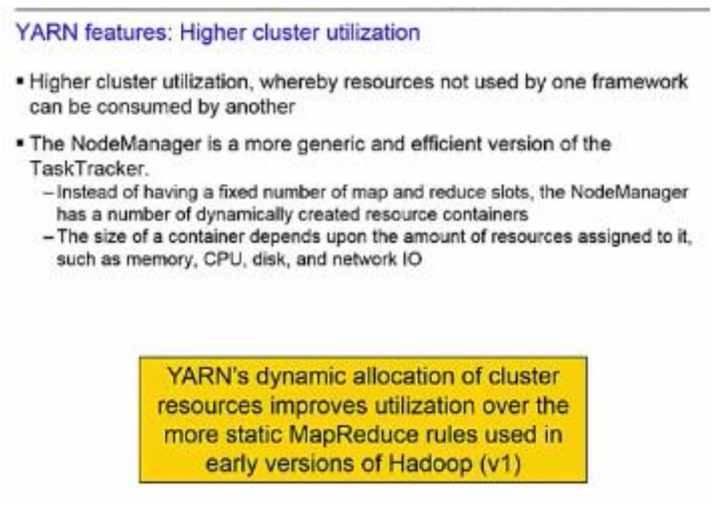
Yarn, Apache Mesos



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

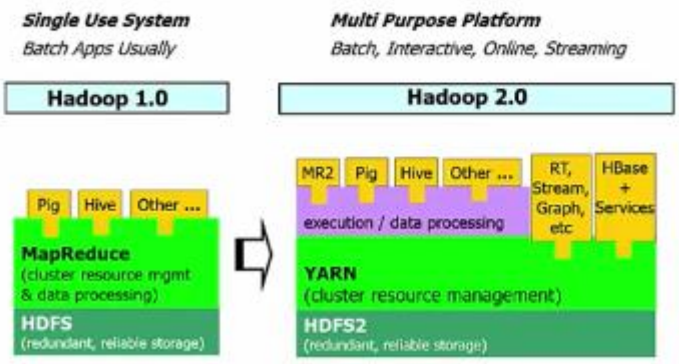
YARN's dynamic allocation of cluster resources improves utilization over the more static MapReduce rules used in early versions of Hadoop (v1)

YARN features: Reliability / Availability

- High availability for the ResourceManager
  - An application recovery is performed after the restart of ResourceManager
  - The ResourceManager stores information about running applications and completed tasks in HDFS
  - If the ResourceManager is restarted, it recreates the state of applications and re-runs only incomplete tasks
- Highly available NameNode, making the Hadoop cluster much more efficient, powerful, and reliable

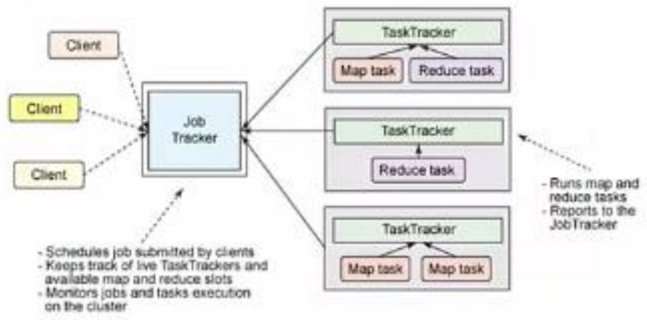
High Availability is work in progress, and is close to completion – features have been actively tested by the community

Hadoop v1 to Hadoop v2


Single Use System
Batch Apps Usually
Multi Purpose Platform
Batch, Interactive, Online, Streaming
Hadoop 1.0
Hadoop 2.0
Pig
Hive
Other ...
MapReduce
(cluster resource mgmt & data processing)
HDFS
(redundant, reliable storage)
MR2
Pig
Hive
Other ...
RT, Stream, Graph, etc
HBase + Services
execution / data processing
YARN
(cluster resource management)
HDFS2
(redundant, reliable storage)

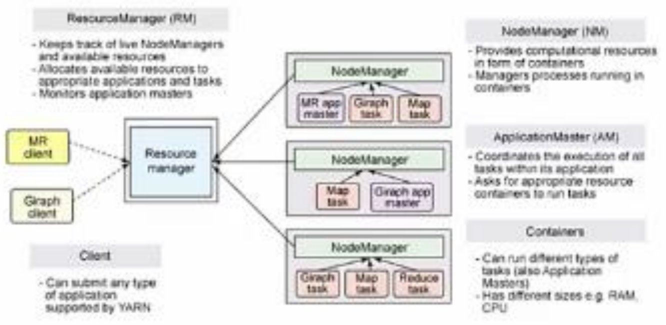
Architecture of MRv1

- Classic version of MapReduce (MRv1)

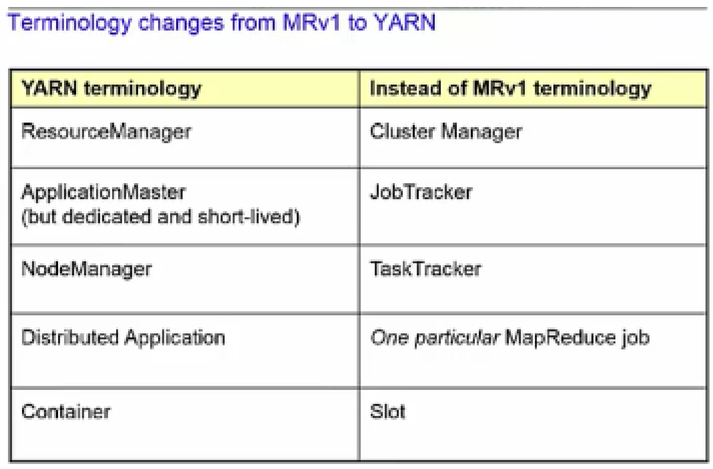

Client
Client
Client
Job Tracker
TaskTracker
Map task
Reduce task
TaskTracker
Reduce task
TaskTracker
Map task
Map task
- Runs map and reduce tasks
- Reports to the JobTracker
- Schedules job submitted by clients
- Keeps track of live TaskTrackers and available map and reduce slots
- Monitors jobs and tasks execution on the cluster

YARN architecture

- High level architecture of YARN


ResourceManager (RM)
- Keeps track of live NodeManagers and available resources
- Allocates available resources to appropriate applications and tasks
- Monitors application masters
MR client
Graph client
Resource manager
Client
- Can submit any type of application supported by YARN
NodeManager
MR app master
Graph task
Map task
NodeManager
Map task
Graph app master
NodeManager
Graph task
Map task
Reduce task
NodeManager (NM)
- Provides computational resources in form of containers
- Managers processes running in containers
ApplicationMaster (AM)
- Coordinates the execution of all tasks within its application
- Asks for appropriate resource containers to run tasks
Containers
- Can run different types of tasks (also Application Masters)
- Has different sizes e.g. RAM, CPU

Read about Apache Twill!



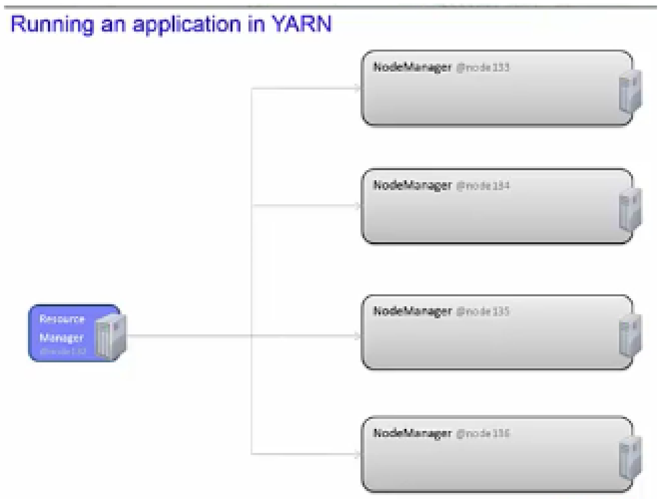
Terminology changes from MRv1 to YARN

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

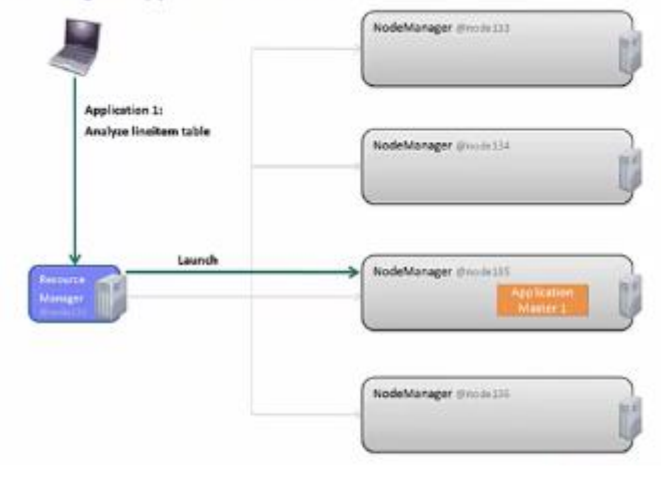
YARN in BigInsights

- 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

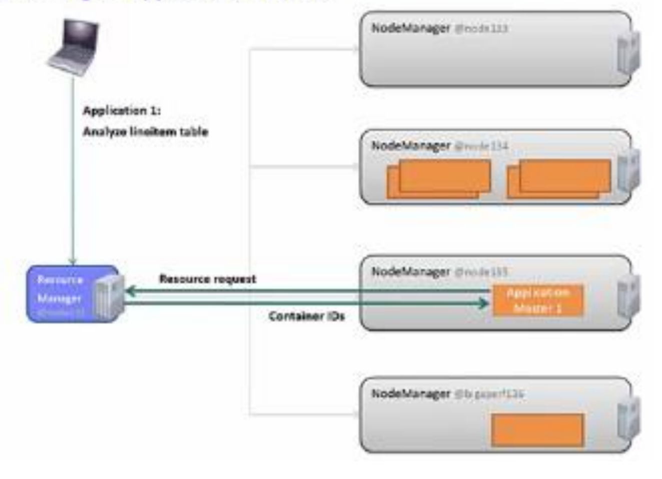
Running an application in YARN



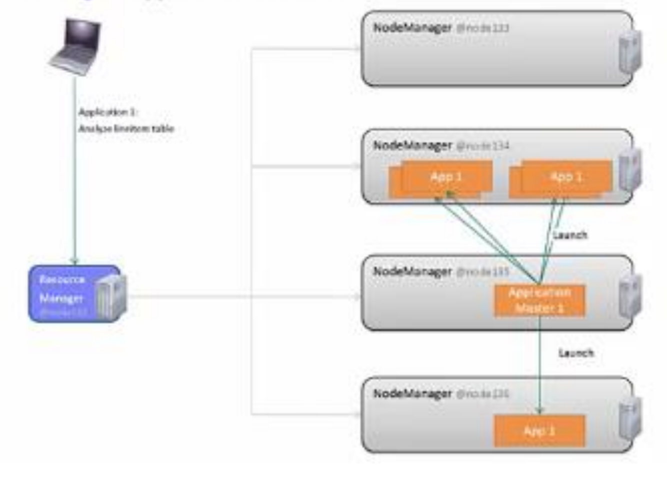
Running an application in YARN



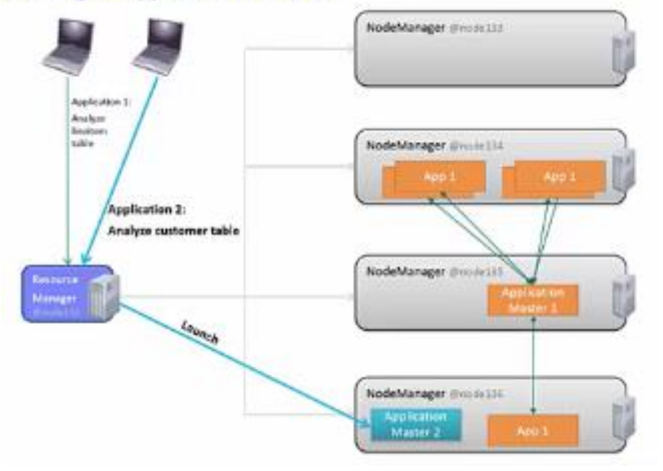
Running an application in YARN



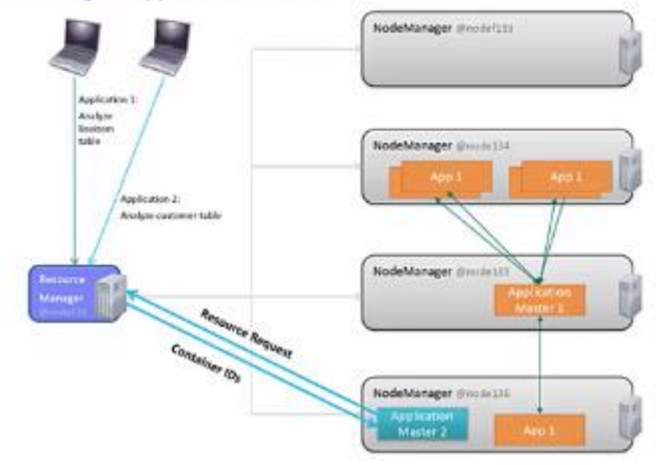
Running an application in YARN



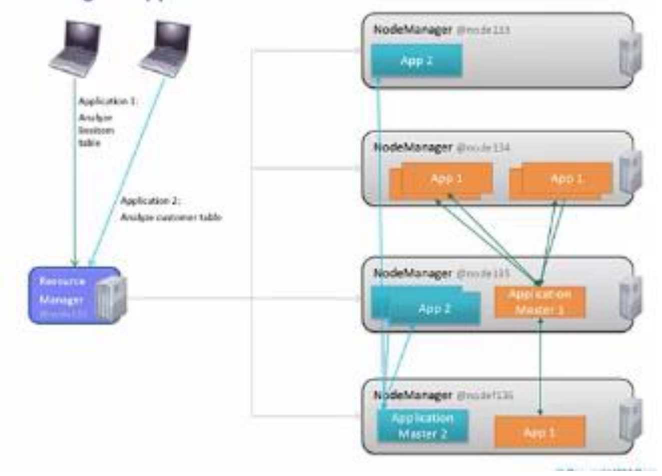
Running an application in YARN



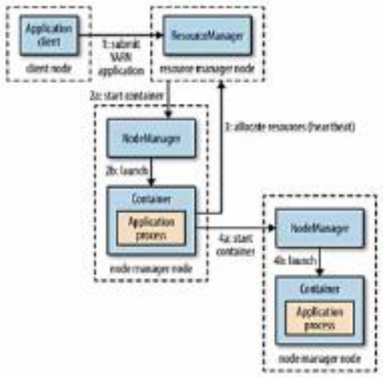
Running an application in YARN



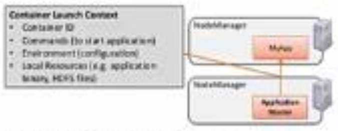
Running an application in YARN



How YARN runs an application



Container Java command line



Container Launch Context
- Container ID
- Commands (to start application)
- Environment (configuration)
- Local Resources (e.g. application binary, HDFS files)

- 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
   00:08:11 /opt/ibm/biginsights/jdk/bin/java -Djava.net.pr
   . . .
```