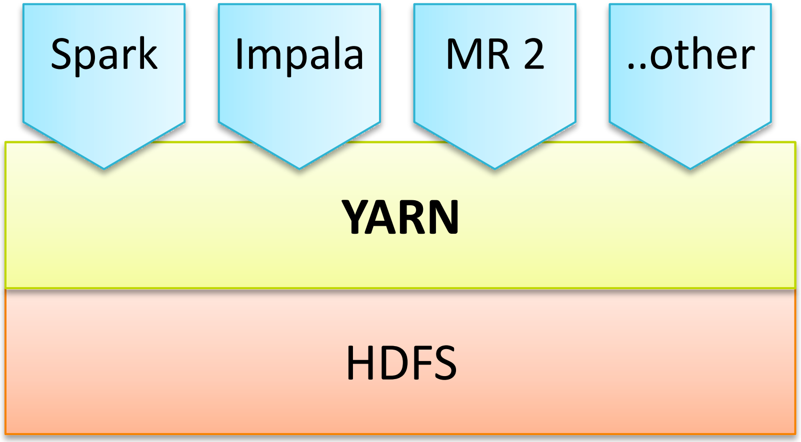
MapReduce and Spark on YARN

Hadoop Computational Frameworks

- **HDFS provides scalable storage in your Hadoop cluster**
- **Computational frameworks provide the distributed computing**
  - Batch processing
  - SQL queries
  - Search
  - Machine learning
  - Stream processing
- **Computational frameworks compete for resources**
- **YARN provides resource management for computational frameworks that support it**
  - Examples discussed in this chapter:
    - MapReduce
    - Apache Spark

What is YARN?

- **Yet Another Resource Negotiator (YARN)**
- **A platform for managing resources in a Hadoop cluster**
- **Supports a growing number of Hadoop distributed processing frameworks, including:**
  - MapReduce v2
  - Spark
  - Impala
  - Others



Why YARN? (1)

- **YARN allows you to run diverse workloads on the same Hadoop cluster**
  - Jobs using different frameworks will probably have different resource profiles
- **Examples:**
  - A MapReduce job or an Impala query that scans a large table
    - Likely heavily disk-bound
    - Requires little memory
  - A Spark job executing an iterative machine learning algorithm
    - Will probably attempt to store the entire dataset in memory
    - May use spurts of CPU to perform complex computations
- **YARN allows you to share cluster memory and CPU resources dynamically between processing frameworks**
  - MapReduce, Impala, Spark, and others

Why YARN? (2)

- **Achieve more predictable performance**
  - Avoid ‘oversubscribing’ nodes
    - Requesting more processing power or RAM than is available
  - Protect higher-priority workloads with better isolation
- **Increase cluster utilization**
  - Resource needs and capacities can be configured less conservatively than would otherwise be possible

Notable Computational Frameworks on YARN

- **MapReduce**
  - The original framework for writing Hadoop applications
  - Proven, widely used
  - Sqoop, Hive, Pig, other tools use MapReduce to interact with HDFS
- **Spark**
  - A newer programming framework for writing Hadoop applications
  - Production-ready
  - Supports processing of streaming data
  - Faster than MapReduce

What is Apache Spark?

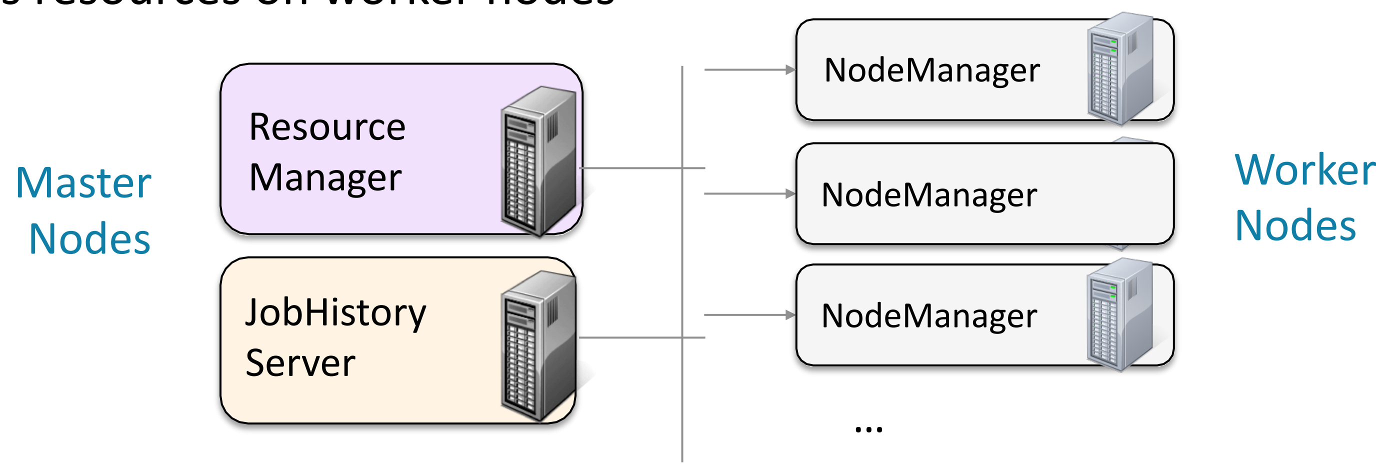
- **A fast, general engine for large-scale data processing on a cluster**
  - One of the fastest-growing Apache projects
  - Includes map and reduce as well as non-batch processing models
- **High-level programming framework**
  - Programmers can focus on logic not plumbing
  - Works directly with HDFS
  - Near real-time processing
    - Configurable in-memory data caching for efficient iteration
- **Application processing is distributed across worker nodes**
  - Distributed storage, horizontally scalable, fault tolerance

Spark Applications

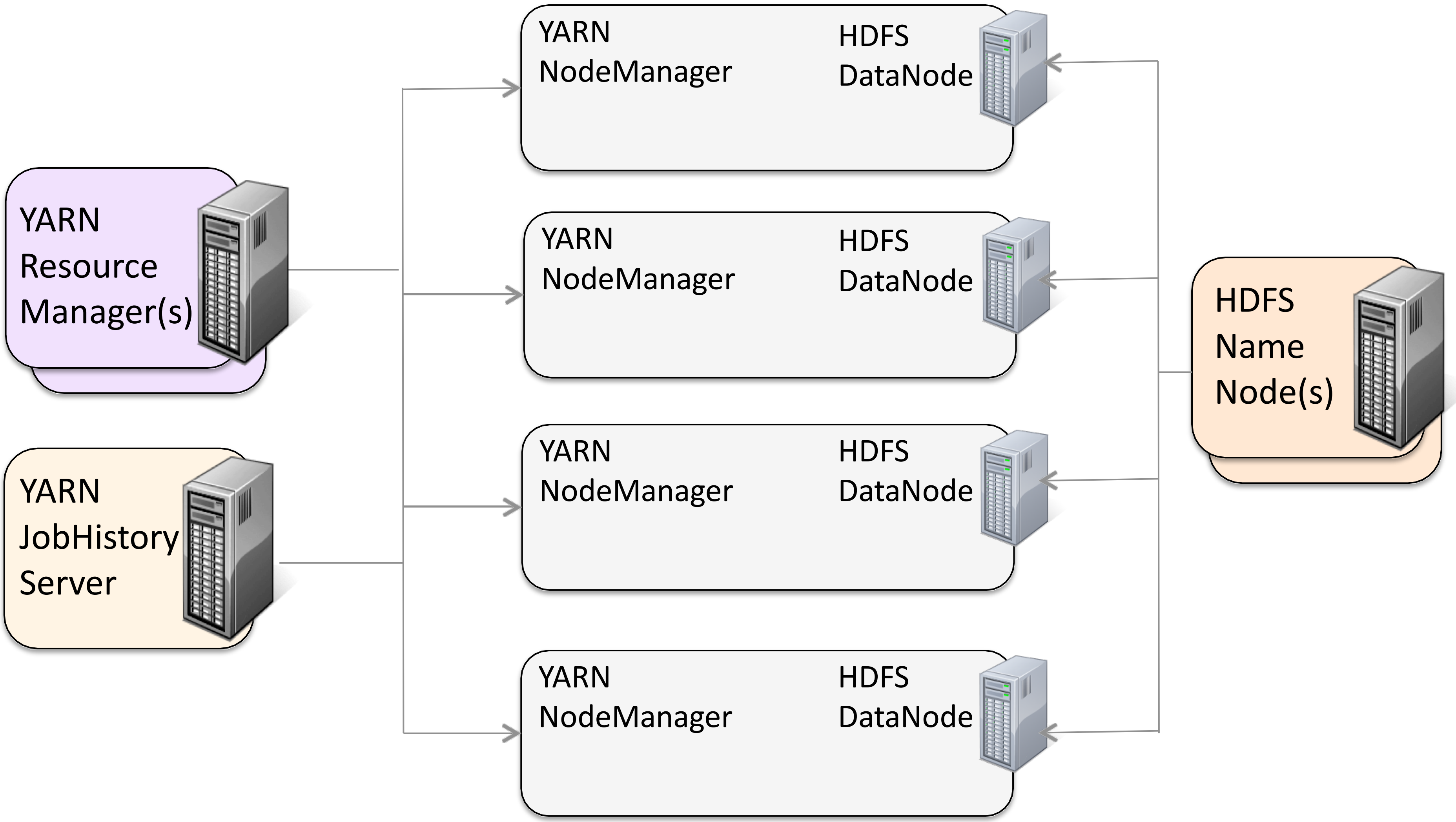
- **Spark Shell**
  - Interactive: for learning, exploring data
  - Python or Scala
- **Spark Applications**
  - Support large scale data processing
  - Python, Scala, or Java
  - A Spark Application consists of one or more jobs
    - A job consists of one or more tasks
- **Every Spark application has a Spark Driver**
  - In “yarn-client” mode, the driver runs on the client
  - In “yarn-cluster” mode, the driver runs on the cluster, on the ApplicationMaster
    - In this case, if the client disconnects the application will continue to run

YARN Daemons

- **ResourceManager – one per cluster**
  - Initiates application startup
  - Schedules resource usage on worker nodes
- **JobHistoryServer – one per cluster**
  - Archives MapReduce jobs' metrics and metadata
- **NodeManager – one per worker node**
  - Starts application processes
  - Manages resources on worker nodes



A Typical YARN Cluster



YARN ResourceManager – Key Points

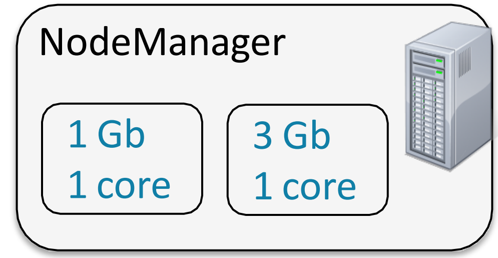
- **What the ResourceManager does:**
  - Manages nodes
    - Tracks heartbeats from NodeManagers
  - Runs a scheduler
    - Determines how resources are allocated
  - Manages containers
    - Handles ApplicationMasters' requests for resources
    - Deallocates containers when they expire or when the application completes
  - Manages ApplicationMasters
    - Creates a container for ApplicationMasters and tracks heartbeats
  - Manages cluster-level security

YARN NodeManagers – Key Points

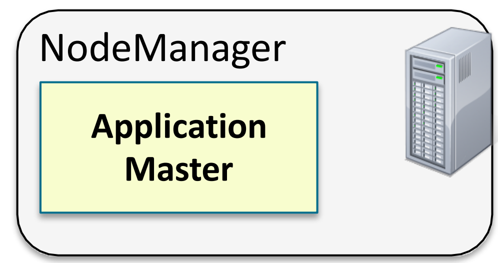
- **What NodeManagers do:**
  - Communicate with the ResourceManager
    - Register and provide info on node resources
    - Send heartbeats and container status
  - Manage processes in containers
    - Launch ApplicationMasters on request from the ResourceManager
    - Launch processes into containers on request from ApplicationMasters
    - Monitor resource usage by containers; kill runaway processes
  - Provide logging services to applications
    - Aggregate logs for an application and save them to HDFS
  - Run auxiliary services
  - Maintain node level security

Running an Application in YARN

- **Containers**
  - Allocated by the ResourceManager
  - Require a certain amount of resources (memory, CPU) on a worker node
  - YARN Applications run in one or more containers



- **Application Master**
  - One per YARN application
  - Runs in a container
  - Framework/application specific
  - Communicates with the ResourceManager scheduler to request containers to run application tasks
  - Ensures NodeManager(s) complete tasks



YARN Container Lifespan

- **MapReduce's use of containers**
  - One container is requested and created *for each task* in a job
  - Each Map or Reduce task gets its own JVM that runs in a container
  - Each container is deleted once a task completes
- **Spark's use of containers**
  - One container is requested and created *for each executor* granted to the Spark application
  - An executor is a JVM
    - Many Spark tasks can run in a single executor – concurrently and over the lifetime of the container
  - Each container stays alive for the lifespan of the application

YARN and Data Locality

- **The YARN Scheduler aims for data locality**
  - Objective: Bring the compute to the data
- **ApplicationMasters know where the HDFS blocks required to complete an application task are located**
  - ApplicationMasters inform the YARN scheduler which node is preferred to accomplish data locality
  - Challenge: Spark must ask YARN for executors *before* jobs are run
    - Application developer can inform YARN which files will be processed
- **When resource availability permits, the YARN Scheduler assigns containers to nodes closest to the data**
  - If the node is not available, YARN will prefer at least the same rack where the task input data resides
  - Scheduling is discussed in more detail later in this course

Summary: Cluster Resource Allocation

- **Resource Manager (master)**
  - Grants containers
  - Performs cluster scheduling
- **Application Master (runs within a container)**
  - Negotiates with the Resource Manager to obtain containers on behalf of the application
  - Presents containers to Node Managers
- **Node Managers (workers)**
  - Manage life-cycle of containers
  - Launch Map and Reduce tasks or Spark executors in containers
  - Monitor resource consumption

Summary: Requesting Resources

- **A resource request is a fine-grained request for memory and CPU sent to the Resource Manager to be fulfilled**
- **If the request is successful, a container is granted**
- **A resource request is composed of several fields that specify**
  - The amount of a given resource required
  - Data locality information, i.e., the preferred node or rack on which to run

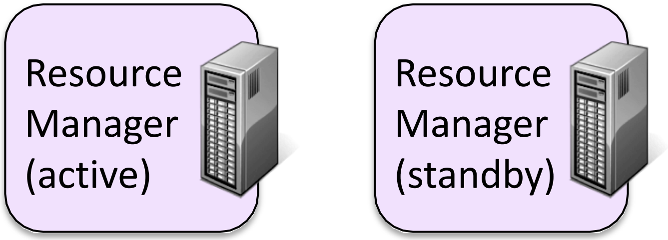
Field Name	Sample Value
priority	integer
capability	<2 gb, 1 vcore>
resourceName	host22, Rack5, *
numContainers	integer

YARN Fault Tolerance (1)

Failure	Action Taken
ApplicationMaster stops sending heartbeats	ResourceManager reattempts the whole application (default: 2 times)
YARN application fails	ResourceManager reattempts the whole application (default: 2 times)
MR task exits with exceptions	ApplicationMaster reattempts the task in a new container on a different node (default: 4 times)
An MR task stops responding	ApplicationMaster reattempts the task in a new container on a different node (default: 4 times)
MR task fails too many times	Task aborted
Spark executor fails	Spark launches new executors (default: Spark tolerates 2 * number of requested executors failing before Spark fails the app)
A Spark task fails	Spark Task Scheduler resubmits task to run on different executor

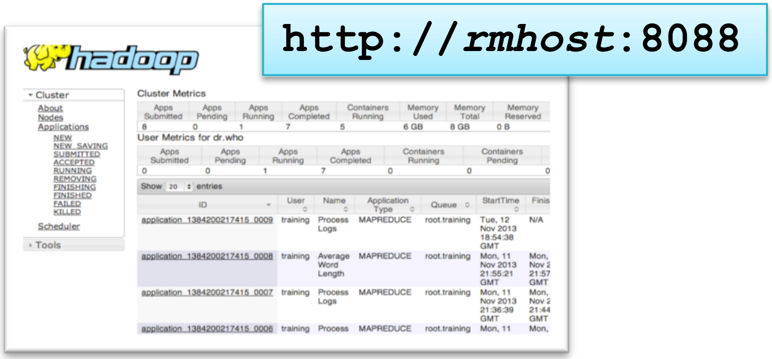
YARN Fault Tolerance (2)

- **NodeManager**
  - If the NodeManager stops sending heartbeats to the ResourceManager, it is removed from list of active nodes
  - Tasks on the node will be treated as failed by the ApplicationMaster
  - If the ApplicationMaster node fails, it will be treated as a failed application
- **ResourceManager**
  - No applications or tasks can be launched if the ResourceManager is unavailable
  - Can be configured with high availability (HA)

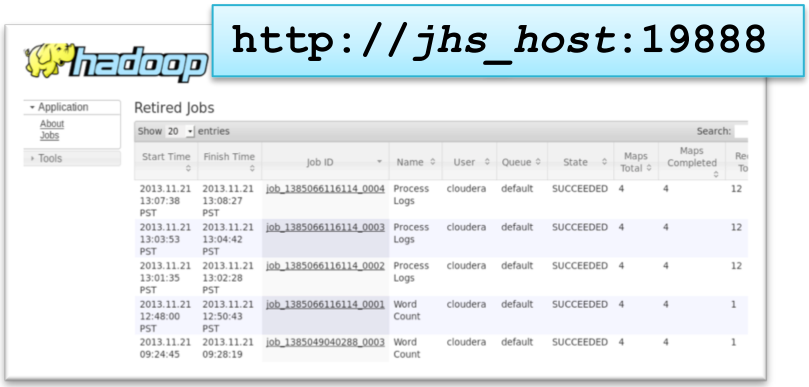


YARN Application Web UIs

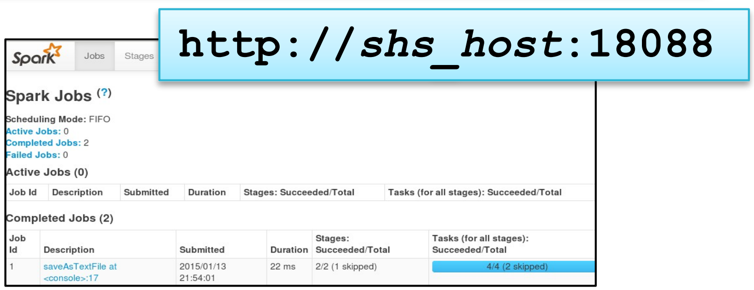
- **ResourceManager Web UI**
  - Provides cluster utilization metrics on nodes, application status, application scheduling, and links to logs
  - Embeds links to the UIs listed below



- **MapReduce JobHistory Server Web UI**
  - http://jhs_host:19888
  - Provides details on retired jobs including state, time metrics, Map and Reduce task details and logs



- **Spark History Server Web UI**
  - http://shs_host:18088
  - Provides details on Spark jobs, stages, storage, environment, and executors

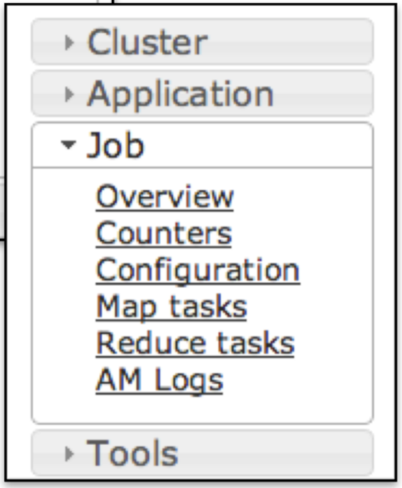


The ResourceManager Web UI

- **The ResourceManager Web UI Menu**
- **Choose a link under "Applications" (e.g., "RUNNING" or "FINISHED")**
- **Then click on the "application ID" to see application metrics including:**
  - Links to app-specific logs
  - If the app has *completed*, an app "history" link that takes you to the applicable history server web UI
  - If the app is *still running*, the "Tracking URL" links to either…
    - The MR Job details in the same UI
    - The Spark Job details in a Spark shell application UI

Main ResourceManager Web UI Menu

MR Job Sub-Menu

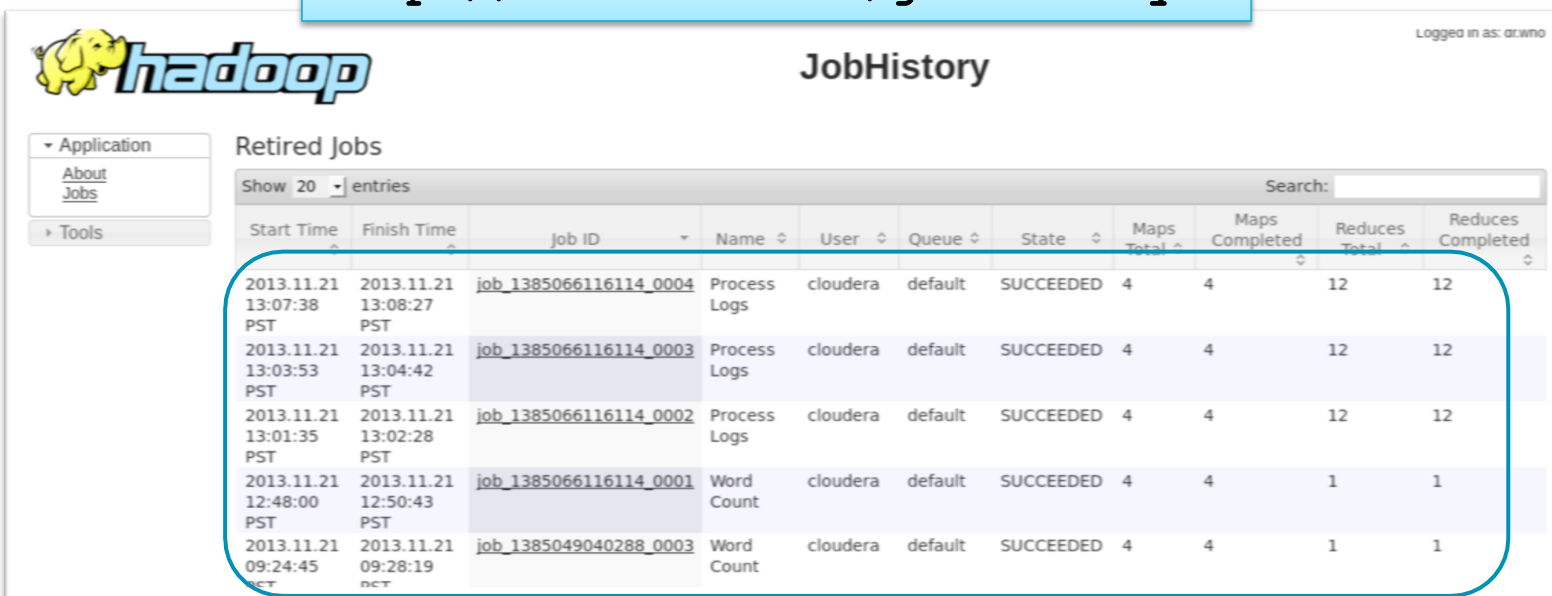


MapReduce Job HistoryServer Web UI

- **The ResourceManager does not keep track of job history**
- **HistoryServer Web UI for MapReduce**
  - Archives jobs' metrics and metadata
  - Can be accessed through Job History Web UI or Hue

Job History Server

`http://rmhost:19888/jobhistory`



Start Time	Finish Time	Job ID	Name	User	Queue	State	Maps Total	Maps Completed	Reduces Total	Reduces Completed
2013.11.21 13:07:38 PST	2013.11.21 13:08:27 PST	job_1385066116114_0004	Process Logs	cloudera	default	SUCCEEDED	4	4	12	12
2013.11.21 13:03:53 PST	2013.11.21 13:04:42 PST	job_1385066116114_0003	Process Logs	cloudera	default	SUCCEEDED	4	4	12	12
2013.11.21 13:01:35 PST	2013.11.21 13:02:28 PST	job_1385066116114_0002	Process Logs	cloudera	default	SUCCEEDED	4	4	12	12
2013.11.21 12:48:00 PST	2013.11.21 12:50:43 PST	job_1385066116114_0001	Word Count	cloudera	default	SUCCEEDED	4	4	1	1
2013.11.21 09:24:45 PST	2013.11.21 09:28:19 PST	job_1385049040288_0003	Word Count	cloudera	default	SUCCEEDED	4	4	1	1

Spark History Server Web UI

`http://sparkHistoryServerHost:18088`

Spark shell (application_1421209... application

Spark Jobs (?)

Scheduling Mode: FIFO
Active Jobs: 0
Completed Jobs: 2
Failed Jobs: 0

Active Jobs (0)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total

Completed Jobs (2)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	saveAsTextFile at <console>:17	2015/01/13 21:54:01	22 ms	2/2 (1 skipped)	4/4 (2 skipped)
0	sortByKey at <console>:16	2015/01/13 21:54:01	81 ms	2/2	4/4

Failed Jobs (0)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total

Hands-On Exercise: Launching a Cluster: YARN – 60 Minutes