Extremely Detailed Notes on Hadoop with a Focus on YARN

Gemini

October 12, 2025

1 Introduction to the Hadoop Ecosystem

Apache Hadoop is an open-source framework designed for storing and processing vast amounts of data, often referred to as Big Data, across clusters of commodity hardware. [15] The core components of the Hadoop ecosystem work together to provide a robust and scalable platform for distributed computing. [12] These fundamental components are:

- **Hadoop Distributed File System (HDFS):** The storage layer of Hadoop, responsible for storing data across multiple machines. [12]
- **MapReduce:** A programming model for processing large datasets in parallel. [12, 15]
- **Yet Another Resource Negotiator (YARN):** The resource management and job scheduling layer, which manages and allocates cluster resources to various applications. [12, 15]

This document provides a brief overview of HDFS and MapReduce to establish the context for a deep and detailed exploration of YARN, which is the central focus. YARN acts as the operating system for Hadoop, managing the cluster's resources and enabling diverse data processing engines to run on a single platform. [1]

2 HDFS (Hadoop Distributed File System)

HDFS is a distributed file system that provides high-throughput access to application data. [12] It is the primary storage system used by Hadoop applications.

2.1 Core Concepts of HDFS

- **Master-Slave Architecture:** HDFS follows a master-slave architecture. [12]
  - **NameNode:** The master server that manages the file system namespace and regulates access to files by clients. [15] It maintains the metadata for the files stored in HDFS.
  - **DataNodes:** These are the slave nodes, typically one per node in the cluster, which store the actual data. [15]
- **Data Blocks:** HDFS splits large files into blocks of a fixed size (typically 128MB or 256MB) and distributes them across the DataNodes.
- **Replication for Fault Tolerance:** Each block is replicated multiple times (usually three) across different DataNodes to ensure data availability in case of a node failure. [2]

2.2 HDFS in the Context of YARN

YARN and HDFS are complementary components. HDFS provides the reliable, distributed storage, while YARN provides the computational framework to process that data. YARN schedules tasks on nodes where the data is located (data locality), which is a critical feature for minimizing network I/O and improving performance. [5] YARN allows various data processing engines like MapReduce, Spark, and Flink to run and process data stored in HDFS. [2, 6]

3 MapReduce

MapReduce is a programming model and processing engine for distributed computing. [12, 15] It simplifies the process of writing parallel and distributed applications.

3.1 Core Concepts of MapReduce

- **Map Phase:** The initial phase where the input data is split and processed. The master node takes the input, divides it into smaller sub-problems, and distributes them to worker nodes. These worker nodes, in turn, process the smaller problem and pass the answer back to the master node. [12]
- **Reduce Phase:** The second phase where the output from the map phase is aggregated to produce the final result. The master node then takes the answers to all the sub-problems and combines them in a way to get the output. [12]

3.2 MapReduce in the Context of YARN

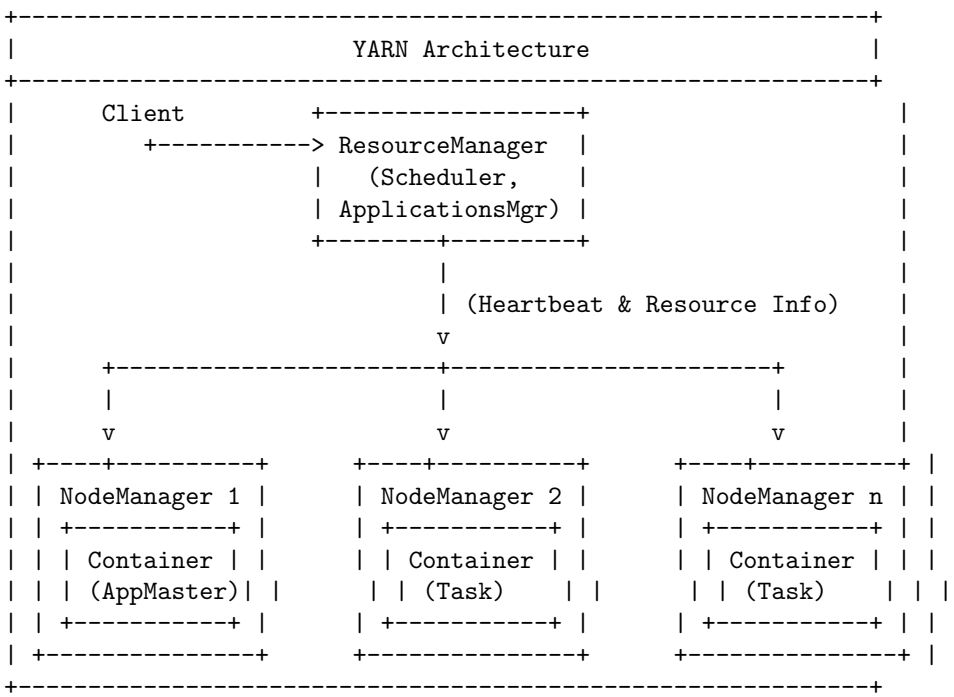
In Hadoop 1.x, MapReduce was tightly coupled with resource management. The JobTracker was responsible for both job scheduling and task monitoring, which created a bottleneck. [6, 19] With the introduction of YARN in Hadoop 2.0, this changed significantly. [6] MapReduce became just one of many applications that can run on YARN. [1, 3] YARN took over the responsibility of resource management, allowing MapReduce to focus solely on data processing. [6] A MapReduce job running on YARN has a dedicated ApplicationMaster that negotiates resources from the YARN ResourceManager and manages the execution of map and reduce tasks within YARN containers. [11]

4 YARN (Yet Another Resource Negotiator)

YARN is the architectural center of Hadoop and acts as a large-scale, distributed operating system for big data applications. [6] Introduced in Hadoop 2.0, its primary goal was to decouple resource management from the processing engine, overcoming the limitations of the classic MapReduce framework. [1, 7]

4.1 Detailed YARN Architecture

The fundamental idea of YARN is to split the functionalities of resource management and job scheduling/monitoring into separate daemons. [8]



4.1.1 ResourceManager (RM)

The ResourceManager is the master daemon that has the ultimate authority to manage and allocate resources among all the applications in the system. [8, 9] It runs on a master node and is responsible for managing the global resources of the cluster. [2] The RM has two main components:

- **Scheduler:** This is a pure scheduler responsible for allocating resources to various running applications based on familiar constraints of capacity, queues, etc. [8] It does not perform any monitoring or tracking of the application's status. [8] The scheduler has a pluggable policy, allowing for different scheduling algorithms. [8]
- **ApplicationsManager (ASM):** The ASM is responsible for accepting job submissions and negotiating the first container for executing the application-specific ApplicationMaster. [7, 8] It also handles restarting the ApplicationMaster container on failure.

4.1.2 NodeManager (NM)

The NodeManager is the slave daemon that runs on each worker node in the cluster. [2, 9] Its primary responsibilities include:

- Launching and managing containers on that node. [9]
- Monitoring the resource usage (CPU, memory, disk, network) of each container. [8]
- Reporting the resource usage and the node's health status to the ResourceManager. [8]

4.1.3 ApplicationMaster (AM)

When an application is started, a dedicated, per-application ApplicationMaster is launched. [8] This is a framework-specific entity and its responsibilities include:

- Negotiating resources (in the form of containers) from the ResourceManager's Scheduler. [8, 11]
- Working with the NodeManagers to execute and monitor the tasks of the application within the allocated containers. [8, 11]

4.1.4 Container

A Container is the basic unit of resource allocation in YARN, representing a collection of physical resources such as a specific amount of memory and CPU cores on a single node. [9, 10, 13] An application's tasks run within these containers, which ensures that each application gets its allocated share of resources and cannot use more than what is assigned. [1]

4.2 YARN Application Submission Workflow

The process of running an application on YARN involves a coordinated effort between the client, ResourceManager, ApplicationMaster, and NodeManagers. [9, 25]

1. **Application Submission:** The client submits an application to the ResourceManager, providing the necessary information to launch the ApplicationMaster, such as the application JAR, command-line options, and resource requirements. [9, 24, 29]
2. **AM Allocation:** The ResourceManager's ApplicationsManager receives the submission and allocates a container to start the ApplicationMaster. [9, 11] It instructs a NodeManager to launch the AM in that container.
3. **AM Registration:** The ApplicationMaster, once started, registers itself with the ResourceManager. This allows the client to query the RM for the AM's status and details.
4. **Resource Negotiation:** The ApplicationMaster determines the resource requirements for its tasks and sends a series of resource requests to the ResourceManager for additional containers. [9, 11]

5. **Container Allocation:** The ResourceManager's Scheduler allocates containers on available NodeManagers based on the requested resources and the scheduling policy.

6. **Task Launch:** The ApplicationMaster receives the allocated container information from the RM. It then communicates directly with the corresponding NodeManagers to launch the tasks in those containers. [9, 11]

7. **Execution and Monitoring:** The tasks execute within their containers and report their progress and status back to the ApplicationMaster. [14] The AM aggregates this information and reports the overall application status to the ResourceManager and the client.

8. **Application Completion:** Once all tasks are complete, the ApplicationMaster unregisters itself from the ResourceManager and shuts down, releasing its own container. The YARN framework then cleans up any remaining resources. [25]

4.3 YARN Schedulers

YARN schedulers are pluggable components that determine how cluster resources are allocated to different applications and queues. [17]

4.3.1 FIFO Scheduler

The First-In, First-Out (FIFO) scheduler places applications in a queue and runs them in the order of submission. [5]

- **How it works:** The first application gets all the resources it needs. Subsequent applications have to wait until the first one finishes.
- **Use Case:** Suitable for dedicated clusters running a single large application.
- **Limitation:** Not ideal for shared clusters, as a long-running, resource-intensive job can block all other jobs, leading to poor turnaround times for smaller, urgent tasks. [4]

4.3.2 Capacity Scheduler

The Capacity Scheduler is designed for multi-tenant clusters, allowing multiple organizations or users to share a cluster with guaranteed capacity allocations. [5, 18]

- **How it works:** The cluster is divided into a hierarchy of queues, each with a configured capacity (a percentage of the total cluster resources). [18] Each queue is guaranteed its minimum capacity, but can also consume idle resources from other queues, a feature known as **queue elasticity**. [5]
- **Use Case:** Ideal for enterprise environments where different departments or projects have specific resource requirements and service level agreements (SLAs). [20]

4.3.3 Fair Scheduler

The Fair Scheduler's goal is to provide fair access to cluster resources for all running applications over time. [5]

- **How it works:** It dynamically balances resources among all running jobs. [4] When a new small job is submitted while a large job is running, the scheduler quickly allocates resources to the new job so it can start running. As the small job finishes, the resources are returned to the large job.
- **Use Case:** Excellent for shared clusters with a mix of short and long-running jobs from various users, especially for ad-hoc queries. [20]

4.4 Worked Examples

4.4.1 Example 1: MapReduce Word Count on YARN

Let's trace the execution of a classic Word Count MapReduce job on YARN.

1. **Submission:** A user submits the job using a command like:
   `hadoop jar wordcount.jar WordCount /input /output`
2. The client communicates with the YARN ResourceManager to get a new application ID. [29] It copies the JAR file and configuration to HDFS.
3. **AM Launch:** The RM allocates a container and launches the MapReduce ApplicationMaster. [11]
4. **Task Calculation:** The AM calculates the number of map tasks based on the input splits of the data in HDFS. It also determines the number of reduce tasks.
5. **Container Request:** The AM requests containers from the RM for its map and reduce tasks. It will specify locality preferences for the map tasks, asking for containers on the DataNodes that hold the data blocks. [5]
6. **Map Task Execution:** The RM allocates containers on the appropriate NodeManagers. The AM instructs these NMs to launch a 'YarnChild' process, which runs the map task in its container. [11] The map task reads its input split from the local HDFS DataNode, processes it, and writes intermediate key-value pairs to the local disk.
7. **Reduce Task Execution:** Once map tasks complete, the AM requests containers for the reduce tasks. The reduce tasks are launched in their containers. They pull the intermediate data from the map tasks (the "shuffle" phase), sort it, and perform the reduction (summing the counts for each word).
8. **Output and Completion:** The reduce tasks write the final output to HDFS. As tasks complete, their containers are released. Once all tasks are done, the AM unregisters from the RM and the application is complete.

4.4.2 Example 2: Scheduler Scenario

Scenario: A small YARN cluster has 2 NodeManagers.

- Node 1: 16 GB RAM, 8 vCores
- Node 2: 16 GB RAM, 8 vCores
- Total Cluster Resources: 32 GB RAM, 16 vCores

Two jobs are submitted at nearly the same time:

- **Job A (Large Batch Job):** Submitted first. Requires 20 containers, each needing 1 GB RAM and 1 vCore. Total: 20 GB RAM, 20 vCores.
- **Job B (Small Urgent Query):** Submitted second. Requires 4 containers, each needing 2 GB RAM and 1 vCore. Total: 8 GB RAM, 4 vCores.

Solution with FIFO Scheduler:

- Job A is first in the queue, so the scheduler allocates all available cluster resources to it. It will get 16 containers (since there are only 16 vCores available).
- Job B will be in a waiting state. It will not receive any resources until Job A completes and releases its containers.
- This demonstrates the blocking nature of the FIFO scheduler, where small, urgent jobs can be starved. [4]

Solution with Capacity Scheduler: Let's configure two queues: 'prod' for Job A and 'dev' for Job B.

- ‘root.prod‘: Capacity 70%
- ‘root.dev‘: Capacity 30%
- ‘prod‘ queue is guaranteed 70% of 32 GB RAM (22.4 GB) and 16 vCores (≈11 vCores).
- ‘dev‘ queue is guaranteed 30% of 32 GB RAM (9.6 GB) and 16 vCores (≈5 vCores).
- When both jobs are submitted, Job A will start running in the ‘prod‘ queue and Job B will start running concurrently in the ‘dev‘ queue.
- Job B will get its required 8 GB RAM and 4 vCores from its queue's guaranteed capacity.
- Job A will get resources from its ‘prod‘ queue. If the ‘dev‘ queue were idle, Job A could have used more than its 70% capacity due to elasticity. [5]
- This setup ensures both jobs run simultaneously, meeting the SLA for the ‘dev‘ queue.

Solution with Fair Scheduler:

- The Fair Scheduler will attempt to give an equal share of resources to both running jobs.
- Initially, it will divide the cluster resources (32 GB, 16 vCores) equally between Job A and Job B.
- Job B only needs 8 GB and 4 vCores, which is less than its fair share. It will receive its requested resources and start immediately.
- The remaining resources (32-8=24 GB, 16-4=12 vCores) will be allocated to Job A.
- This approach dynamically balances the load, ensuring that smaller jobs get a chance to run quickly without needing pre-configured queues. [4]

4.5 Advantages of YARN

- **Scalability:** By decentralizing job management to the ApplicationMasters, YARN's architecture is more scalable than the JobTracker-based design of Hadoop 1. [6, 19]
- **Flexibility and Multi-tenancy:** YARN can run multiple data processing frameworks (MapReduce, Spark, Flink, etc.) on the same cluster, allowing organizations to run diverse workloads. [1, 6, 7]
- **Improved Cluster Utilization:** Resources are managed more dynamically and efficiently. Instead of fixed slots for map and reduce tasks, YARN allocates containers based on the application's actual needs, leading to better utilization. [1, 6, 19]
- **Better Resource Management:** YARN provides a centralized system for managing cluster resources, which leads to improved performance and efficiency. [1, 6]
- **High Availability:** YARN supports high availability for the ResourceManager, which eliminates the single point of failure that existed with the JobTracker in Hadoop 1. [1, 19]
- **Compatibility:** YARN is backward compatible with MapReduce applications written for Hadoop 1, ensuring a smooth migration path. [6, 22]

4.6 Disadvantages of YARN

- **Complexity:** YARN's architecture is more complex than the original MapReduce framework. It requires additional configuration and tuning, which can be challenging for new users. [1, 6]
- **ApplicationMaster Overhead:** Each application requires its own ApplicationMaster, which consumes resources. For applications with many small jobs, this can introduce significant overhead. [1]
- **Potential Single Point of Failure:** While YARN supports a high-availability configuration for the ResourceManager, if not properly configured, the RM can still be a single point of failure. [6]
- **Latency:** The process of negotiating resources and launching containers can introduce latency, which might be a concern for very low-latency applications. [6]