YARN Architecture

YARN, which stands for Yet Another Resource Negotiator, is a core component of Hadoop 2.0 and beyond. It was introduced to address the limitations of Hadoop 1.0, particularly the bottleneck caused by the Job Tracker. YARN separates the resource management layer from the processing layer, making it a more efficient and scalable solution for big data processing.

Key Components of YARN Architectur e

Resource Manager: The Resource Manager is the master daemon of YARN and is responsible for resource assignment and management among all applications. Whenever it receives a processing request, it forwards it to the corresponding node manager and allocates resources for the completion of the request accordingly. It has two main components:

- **Scheduler:** Allocates resources to running applications based on available resources and does not monitor or track the status of applications.
- **Application Manager:** Manages the lifecycle of applications, including accepting job submissions and negotiating the first container from the Resource Manager.

Node Manager: The Node Manager is responsible for managing individual nodes in a Hadoop cluster. It monitors resource usage, performs log management, and communicates with the Resource Manager through heartbeats. It also manages the lifecycle of containers on its node.

It takes care of individual node on Hadoop cluster and manages application and workflow and that particular node. Its primary job is to keep-up with the Resource Manager. It registers with the Resource Manager and sends heartbeats with the health status of the node. It monitors resource usage, performs log management and also kills a container based on directions from the resource manager. It is also responsible for creating the container process and start it on the request of Application master.

Application Master: An application is a single job submitted to a framework. The application master is responsible for negotiating resources with the resource manager, tracking the status and monitoring progress of a single application. The application master requests the container from the node manager by sending a Container Launch Context(CLC) which includes everything an application needs to run. Once the application is started, it sends the health report to the resource manager from time-to-time.

Containers: Containers are collections of physical resources such as RAM, CPU cores, and disk space on a single node. They are invoked by the Container Launch Context (CLC), which includes all the necessary information for running an application.

Application Workflow in YARN

- Client Submits Application: The client submits an application to the Resource Manager.

- Resource Allocation: The Resource Manager allocates a container to start the Application Manager.
- Application Manager Registration: The Application Manager registers itself with the Resource Manager.
- Resource Negotiation: The Application Manager negotiates containers from the Resource Manager.
- Container Launch: The Application Manager notifies the Node Manager to launch containers.
- Execution: Application code is executed in the container.
- Monitoring: The client contacts the Resource Manager or Application Manager to monitor the application's status.
- Completion: Once processing is complete, the Application Manager unregisters with the Resource Manager

Advantages

- Flexibility: YARN offers flexibility to run various types of distributed processing systems such as Apache Spark, Apache Flink, Apache Storm, and others. It allows multiple processing engines to run simultaneously on a single Hadoop cluster.
- Resource Management: YARN provides an efficient way of managing resources in the Hadoop cluster. It allows administrators to allocate and monitor the resources required by each application in a cluster, such as CPU, memory, and disk space.
- Scalability: YARN is designed to be highly scalable and can handle thousands of nodes in a cluster. It can scale up or down based on the requirements of the applications running on the cluster.
- Improved Performance: YARN offers better performance by providing a centralized resource management system. It ensures that the resources are optimally utilized, and applications are efficiently scheduled on the available resources.
- Security: YARN provides robust security features such as Kerberos authentication, Secure Shell (SSH) access, and secure data transmission. It ensures that the data stored and processed on the Hadoop cluster is secure.

Job scheduling: Job scheduling in YARN refers to the process of allocating cluster resources (CPU, memory, containers) to different jobs and applications based on defined scheduling policies. Since multiple users and applications may run simultaneously, YARN ensures fair distribution of resources while meeting business requirements.

How YARN Scheduling Works (Step-by-Step)

- A user submits a job to the ResourceManager.
- The ResourceManager assigns a container for the ApplicationMaster.
- The ApplicationMaster requests resources for tasks.
- The Scheduler in ResourceManager applies the scheduling policy (FIFO, Capacity, or Fair) to allocate resources.

- The tasks are launched on NodeManagers as containers.
- Job progress is tracked, and results are sent back to the user.

Optimizing Hadoop Performance for Large-Scale Data

Hardware Configuration

Proper hardware configuration is crucial for optimizing Hadoop performance. Key considerations include:

- CPU: Use processors with high clock speeds and multiple cores to handle parallel processing.
- Memory: Allocate sufficient RAM to reduce disk I/O and improve job execution.
- Storage: Use fast storage devices, such as solid-state drives (SSDs), to improve data access and reduce latency.
- Network: Ensure a high-bandwidth network to facilitate efficient data transfer between nodes.

HDFS Optimization

Optimizing the Hadoop Distributed File System (HDFS) can significantly improve overall performance:

- Block Size: Increase the default block size (typically 128MB) to reduce the number of blocks and improve read/write efficiency.
- Replication Factor: Adjust the replication factor based on the criticality of data and available storage resources.
- Data Locality: Ensure data is stored close to the processing nodes to minimize network overhead.

MapReduce Optimization

Optimizing the MapReduce framework can enhance the performance of data processing tasks:

- Input Splits: Tune the input split size to match the block size and improve data locality.
- Mapper and Reducer Configurations: Adjust the number of mappers and reducers based on the task complexity and available resources.
- Compression: Enable compression for intermediate data to reduce network and storage requirements.

YARN Optimization

Optimizing the YARN resource manager can help manage cluster resources more efficiently:

- Resource Allocation: Allocate appropriate CPU, memory, and other resources to the application containers.
- Queue Configuration: Configure YARN queues to prioritize and manage workloads effectively.
- Scheduling Policies: Choose the appropriate scheduling algorithm (e.g., FIFO, Fair, Capacity) based on the workload requirements.

MapReduce Programming model: MapReduce is the programming model used within the Hadoop framework to perform the map and reduce steps for large-scale data processing. It is designed to process extensive datasets in a distributed and parallel manner, leveraging the Hadoop Distributed File System (HDFS) for storage.

The map step involves processing input data and converting it into intermediate key-value pairs. The reduce step then aggregates these intermediate results to produce the final output. This model is highly scalable and efficient, enabling the processing of petabytes of data across multiple servers.

In the Hadoop framework, MapReduce is the programming model. MapReduce utilizes the map and reduce strategy for the analysis of data. In today's fast-paced world, there is a huge number of data available, and processing this extensive data is one of the critical tasks to do so. However, the MapReduce programming model can be the solution for processing extensive data while maintaining both speed and efficiency. Understanding this programming model, its components, and execution workflow in the Hadoop framework will be helpful to gain valuable insights.

MapReduce is a parallel, distributed programming model in the Hadoop framework that can be used to access the extensive data stored in the Hadoop Distributed File System (HDFS). The Hadoop is capable of running the MapReduce program written in various languages such as Java, Ruby, and Python. One of the beneficial factors that MapReduce aids is that MapReduce programs are inherently parallel, making the very large scale easier for data analysis.

When the MapReduce programs run in parallel, it speeds up the process. The process of running MapReduce programs is explained below.

Dividing the input into fixed-size chunks: Initially, it divides the work into equal-sized pieces. When the file size varies, dividing the work into equal-sized pieces isn't the straightforward method to follow, because some processes will finish much earlier than others while some may take a very long run to complete their work. So one of the better approaches is that one that requires more work is said to split the input into fixed-size chunks and assign each chunk to a process.

Combining the results: Combining results from independent processes is a crucial task in MapReduce programming because it may often need additional processing such as aggregating and finalizing the results.

Key Components of MapReduce

- Mapper: Processes input data and generates intermediate key-value pairs.
- Reducer: Aggregates the mapper's output to produce the final result.
- Shuffling and Sorting: Ensures that all values with the same key are sent to the same reducer.
- Combiner: Performs local aggregation to minimize data transfer between the mapper and reducer.

There are two key components in the MapReduce. The MapReduce consists of two primary phases such as the map phase and the reduces phase. Each phase contains the key-value pairs as its input and output and it also has the map function and reducer function within it.

Mapper: Mapper is the first phase of the MapReduce. The Mapper is responsible for processing each input record and the key-value pairs are generated by the InputSplit and RecordReader. Where these key-value pairs can be completely different from the input pair. The MapReduce output holds the collection of all these key-value pairs.

Reducer: The reducer phase is the second phase of the MapReduce. It is responsible for processing the output of the mapper. Once it completes processing the output of the mapper, the reducer now generates a new set of output that can be stored in HDFS as the final output data.

Execution Workflow

- Input data is split into chunks and distributed across nodes.
- Mappers process the chunks and generate intermediate key-value pairs.
- The framework shuffles and sorts the mapper outputs.
- Reducers process the sorted data to generate the final output, which is stored in HDFS.

MapReduce is a core component of Hadoop and is widely used for tasks like log analysis, text mining, and machine learning.

Execution workflow of MapReduce

Now let’s understand how the MapReduce Job execution works and what all the components it contains. Generally, MapReduce processes the data in different phases with the help of its different components. Take a look at the below figure which illustrates the steps of the job execution workflow of MapReduce in Hadoop.

- Input Files: The data for MapReduce tasks are present in the input files. These input files reside in HDFS. The format for input files is arbitrary, while line-based log files and binary format can also be used.
- InputFormat: The InputFormat is used to define how the input files are split and read. It selects the files or objects that are used for input. In general, the InputFormat is used to create the Input Split.
- Record Reader: The RecordReader can communicate with the Input Split in the Hadoop MapReduce. It can also convert the data into key-value pairs so that the mapper can read. By default, the RecordReader utilizes the TextInputFormat for converting data into key-value pairs. The Record Reader communicates with the Input Split until the file reading is completed. It then assigns a byte offset (unique number) to each line present in the file. Then these key-value pairs are sent to the mapper for further processing.
- Mapper: From the RecordReader, the mapper receives the input records. The Mapper is responsible for processing those input records from the RecordReader and it generates the new key-value pair. The Key-value pair generated by the mapper can be completely different from the input pair. The output of the mapper which is intermediate output is said to be stored in the local disk since it is the temporary data.
- Combiner: The Combiner in MapReduce is also known as Mini-reducer. The Hadoop MapReduce combiner performs the local aggregation on the mapper's output which minimizes the data transfer between the mapper and reducer. Once the Combiner completes its process, the output of the combiner is passed to the partitioner for further work.
- Partitioner: In Hadoop MapReduce, the partitioner is used when we are working with more than one reducer. The Partitioner extracts the output from the combiner and then it performs partitioning. The partitioning of output takes place based on the key and then it is sorted. With the help of a hash function, the key (or subset of the key) is used to derive the partition. Since MapReduce execution works with the help of key-value, each combiner output is partitioned and a record having the same key value moves into the same partition and then each partition is sent to the reducer. Partitioning of the output of the combiner allows the even distribution of the map output over the reducer.
- Shuffling and Sorting: The Shuffling performs the shuffling operation on the mapper's output before it is sent to the reducer phase. Once all the mapper has completed their work and their output is said to be shuffled on the reducer nodes, then this intermediate output is merged and sorted. This sorted output is passed as input to the reducer phase.
- Reducer: It takes the set of intermediate key-value pairs from the mapper as the input and then it runs the reducer function on each of the key-value pairs to generate the output. This output of the reducer phase is the final output and it is stored in the HDFS.

- Record Writer: The Record Writer holds the power of writing these output key-value pairs from the reducer phase to the output files.
- Output format: The Output format determines how these output values are written in the output files by the record reader. The Output format instances provided by Hadoop are generally used to write files on either HDFS or on the local disk. Thus, the final output of the reducer is written on the HDFS by output format instances.