CHENNAI INSTITUTE OF TECHNOLOGY
(Autonomous)

UNIT IV MAP REDUCE APPLICATIONS

MapReduce workflows - unit tests with MRUnit - test data and local tests - anatomy of MapReduce job run - classic Map-reduce - YARN - failures in classic Map-reduce and YARN -job scheduling - shuffle and sort - task execution - MapReduce types - input formats - output formats.

MapReduce Overview

MapReduce is the processing engine of Hadoop that processes and computes large volumes of data. It is one of the most common engines used by Data Engineers to process Big Data. It allows businesses and other organizations to run calculations to: Determine the price for their products that yields the highest profits Know precisely how effective their advertising is and where they should spend their ad dollars Make weather predictions Mine web clicks, sales records purchased from retailers, and Twitter trending topics to determine what new products the company should produce in the upcoming season.
Before MapReduce, these calculations were complicated. Now, programmers can tackle problems like these with relative ease. Data scientists have coded complex algorithms into frameworks so that programmers can use them.

There are two phases in the MapReduce programming model:

1. Mapping
2. Reducing

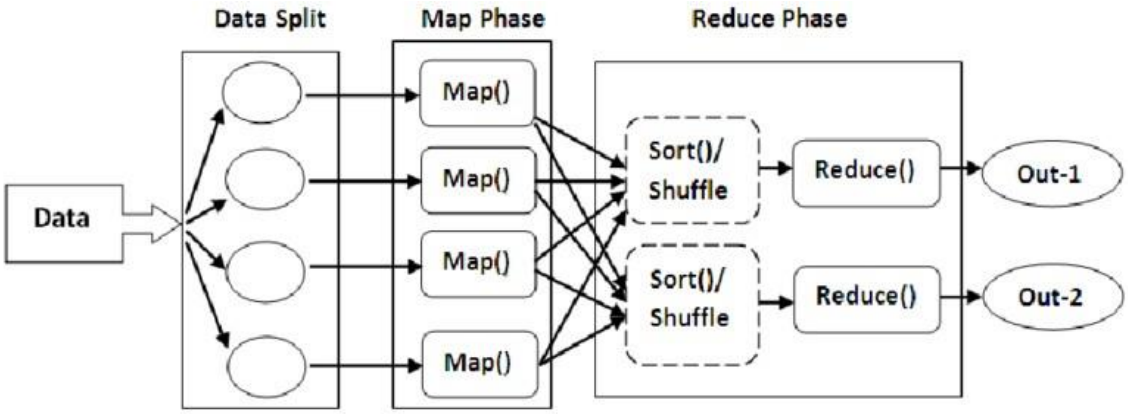
Mapping and Reducing
A mapper class handles the mapping phase; it maps the data present in different datanodes. A reducer class handles the reducing phase; it aggregates and reduces the output of different datanodes to generate the final output.
Data that is stored on multiple machines pass through mapping. The final output is obtained after the data is shuffled, sorted, and reduced.

Input Data
Hadoop accepts data in various formats and stores it in HDFS. This input data is worked upon by multiple map tasks.

Map Tasks
Map reads the data, processes it, and generates key-value pairs. The number of map tasks depends upon the input file and its format.
Typically, a file in a Hadoop cluster is broken down into blocks, each with a default size of 128 MB. Depending upon the size, the input file is split into multiple chunks. A map task then runs for each chunk. The mapper class has mapper functions that decide what operation is to be performed on each chunk. **Reduce Tasks**
In the reducing phase, a reducer class performs operations on the data generated from the map tasks through a reducer function. It shuffles, sorts, and aggregates the intermediate key-value pairs (tuples) into a set of smaller tuples.

Output
The smaller set of tuples is the final output and gets stored in HDFS.

MapReduce Workflows
The MapReduce workflow is as shown:



1. The input data that needs to be processed using MapReduce is stored in HDFS. The processing can be done on a single file or a directory that has multiple files.
2. The input format defines the input specification and how the input files would be split and read.
3. The input split logically represents the data to be processed by an individual mapper.
4. RecordReader communicates with the input split and converts the data into key-value pairs suitable to be read by the mapper.
5. The mapper works on the key-value pairs and gives an intermittent output, which goes for further processing.
6. Combiner is a mini reducer that performs mini aggregation on the key-value pairs generated by the mapper.
7. Partitioner decides how outputs from combiners are sent to the reducers.
8. The output of the partitioner is shuffled and sorted. This output is fed as input to the reducer.
9. The reducer combines all the intermediate values for the intermediate keys into a list called tuples.

10. The RecordWriter writes these output key-value pairs from reducer to the output files. The output data gets stored in HDFS.

Unit Tests With MR Unit

Unit Test MapReduce using MRUnit

In order to make sure that your code is correct, you need to Unit test your code first. And like you unit test your Java code using JUnit testing framework, the same can be done using MRUnit to test MapReduce Jobs.

MRUnit is built on top of JUnit framework. So we will use the JUnit classes to implement unit test code for MapReduce. If you are familiar with JUnits then you will find unit testing for MapReduce jobs also follows the same pattern.

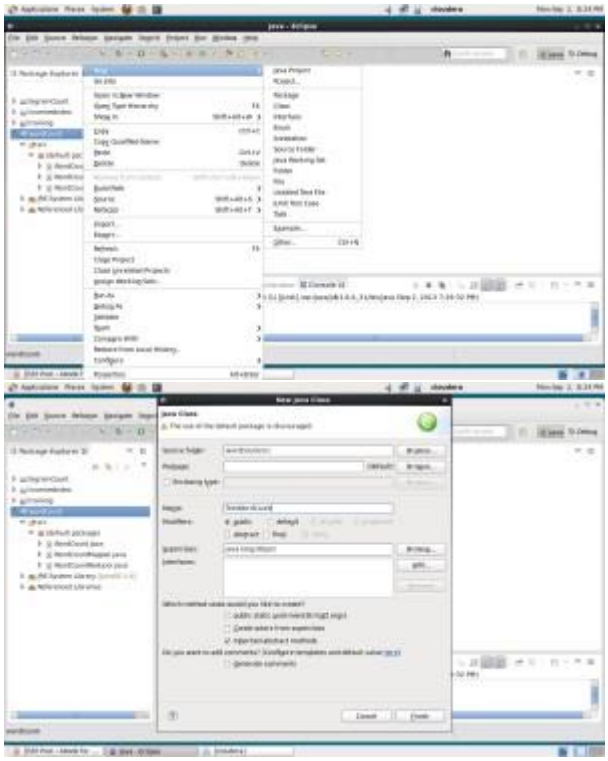
Let us discuss the template that can be used for writing any unit test for MapReduce job.

To Unit test MapReduce jobs:

1. Create a new test class to the existing project
2. Add the mrunit jar file to build path
3. Declare the drivers
4. Write a method for initializations & environment setup
5. Write a method to test mapper
6. Write a method to test reducer
7. Write a method to test the whole MapReduce job
8. Run the test

Create a new test class to the existing project

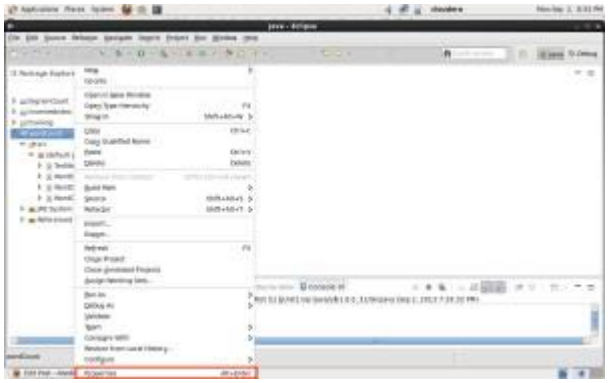
First create a new class with the name “TestWordCount” in the existing wordCount project.



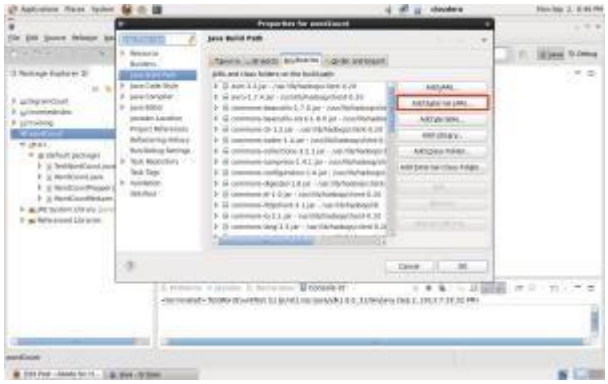
Add the mrunit jar file to build path

Download the latest mrunit jar file from http://apache.cs.utah.edu/mrunit/mrunit-1.0.0/. Unzip the folder and you will find mrunit jar file inside the lib directory. We need to add that jar to the build path.

Right click on the project and click on “Properties”.



Click on Add external Jar file and add the jar file you recently downloaded.



Declare the drivers

Instead of running the actual driver class, for unit testing we will declare drivers to test mapper, reducer and the whole MapReduce job.

```
1 MapDriver<LongWritable, Text, Text, IntWritable> mapDriver;
2
3 ReduceDriver<Text, MapReduceDriver<LongWritable, Text, Text, IntWritable,
Text, IntWritable> IntWritable, Text, IntWritable> reduceDriver;
mapReduceDriver;
```

Note that you need to import the following:

```
1 import org.apache.hadoop.mrunit.mapreduce.MapDriver;
2 import
org.apache.hadoop.mrunit.mapreduce.MapReduceDriver;
3 import org.apache.hadoop.mrunit.mapreduce.ReduceDriver;
```

and not

```
1 import org.apache.hadoop.mrunit.MapDriver;
2 import
org.apache.hadoop.mrunit.MapReduceDriver;
3 import org.apache.hadoop.mrunit.ReduceDriver;
```

As the word count mapper takes LongWritable offset and Text of line, we have given the same as the generic parameters of the mapDriver. Same is the case with the reduceDriver and mapReduceDriver.

Write a method for initializations & environment setup

This is the code that runs before any (and every) test runs and can be used for all the initializations that you want to do. You will need to add @Before annotation before this method.

```
@Before
public void setUp()
{
    WordCountMapper mapper = new WordCountMapper();
    mapDriver = new MapDriver<LongWritable, Text, Text, IntWritable>();
mapDriver.setMapper(mapper);

    WordCountReducer reducer = new WordCountReducer();     reduceDriver =
new ReduceDriver<Text, IntWritable, Text, IntWritable>();
reduceDriver.setReducer(reducer);

    mapReduceDriver = new MapReduceDriver<LongWritable, Text,
Text, IntWritable, Text, IntWritable>();
mapReduceDriver.setMapper(mapper);    }
mapReduceDriver.setReducer(reducer);
```

In the first three lines of the above code, we create an object of the mapDriver and set the mapper for the driver. **Note** that we are not setting the reducer class. It is because the mapDriver will only test the mapper logic.

Similarly we set the reducer class to the reduceDriver. However, when we want to test the MapRedcue job as a whole, we need to test both the mapper and the reducer. Therefore we need to set both the mapper and reducer class to the mapReduceDriver.

Write a method to test mapper

To declare a method as a test method it must be annotated with @Test.

```
@Test
public void testMapper() throws IOException
{
    mapDriver.withInput(new LongWritable(1), new Text("orange orange

apple"));    mapDriver.withOutput(new Text("orange"), new IntWritable(1));
mapDriver.withOutput(new Text("orange"), new IntWritable(1));
mapDriver.withOutput(new Text("apple"), new IntWritable(1));
mapDriver.runTest();
}
```

The mapDriver takes as input a LongWritable and a Text “apple orange orange” in the form of key value pair. We want out wordCountMapper to output each word as key and “1” as the value. So we set the output of the driver accordingly. Finally runTest method runs the mapDriver.

Write a method to test reducer

In the wordCount example, we get a word and list of IntWritable values (all 1's) associated with it. The reducer code is then supposed to give the final ouput with word as key and its count as value. To test the reducer functionality we use the code given below.

```
@Test
public void testReducer() throws IOException
{
    List values = new ArrayList();
values.add(new    IntWritable(1));
values.add(new IntWritable(1));
    reduceDriver.withInput(new      Text("orange"),       values);
reduceDriver.withOutput(new  Text("orange"),  new  IntWritable(2));
reduceDriver.runTest();
}
```

Write a method to test the whole MapReduce job

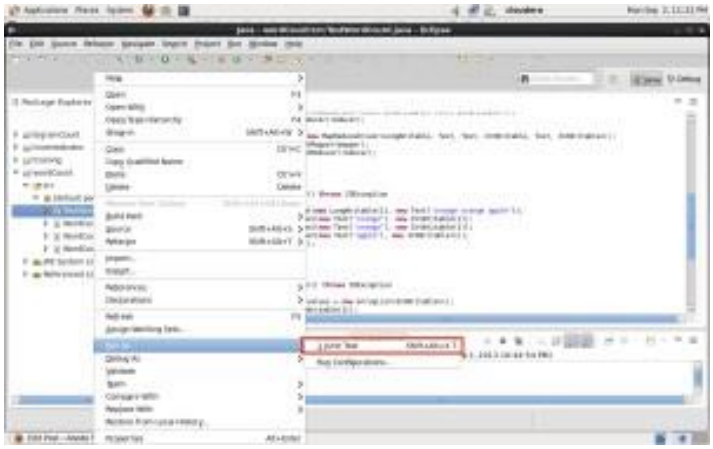
In order to test the complete MapReduce job, we give the input offset as key and line of text as value to the mapReduceDriver. And the final output is supposed to be the word as key and its count as the value. We therefore set addInput and addOutput appropriately.

```
@Test
public void testMapperReducer() throws IOException
{
    mapReduceDriver.addInput(new LongWritable(1), new Text("orange orange


apple"));    mapReduceDriver.addOutput(new Text("orange"), new
IntWritable(2));     mapReduceDriver.addOutput(new Text("apple"), new
IntWritable(1));     mapReduceDriver.runTest();
}
```

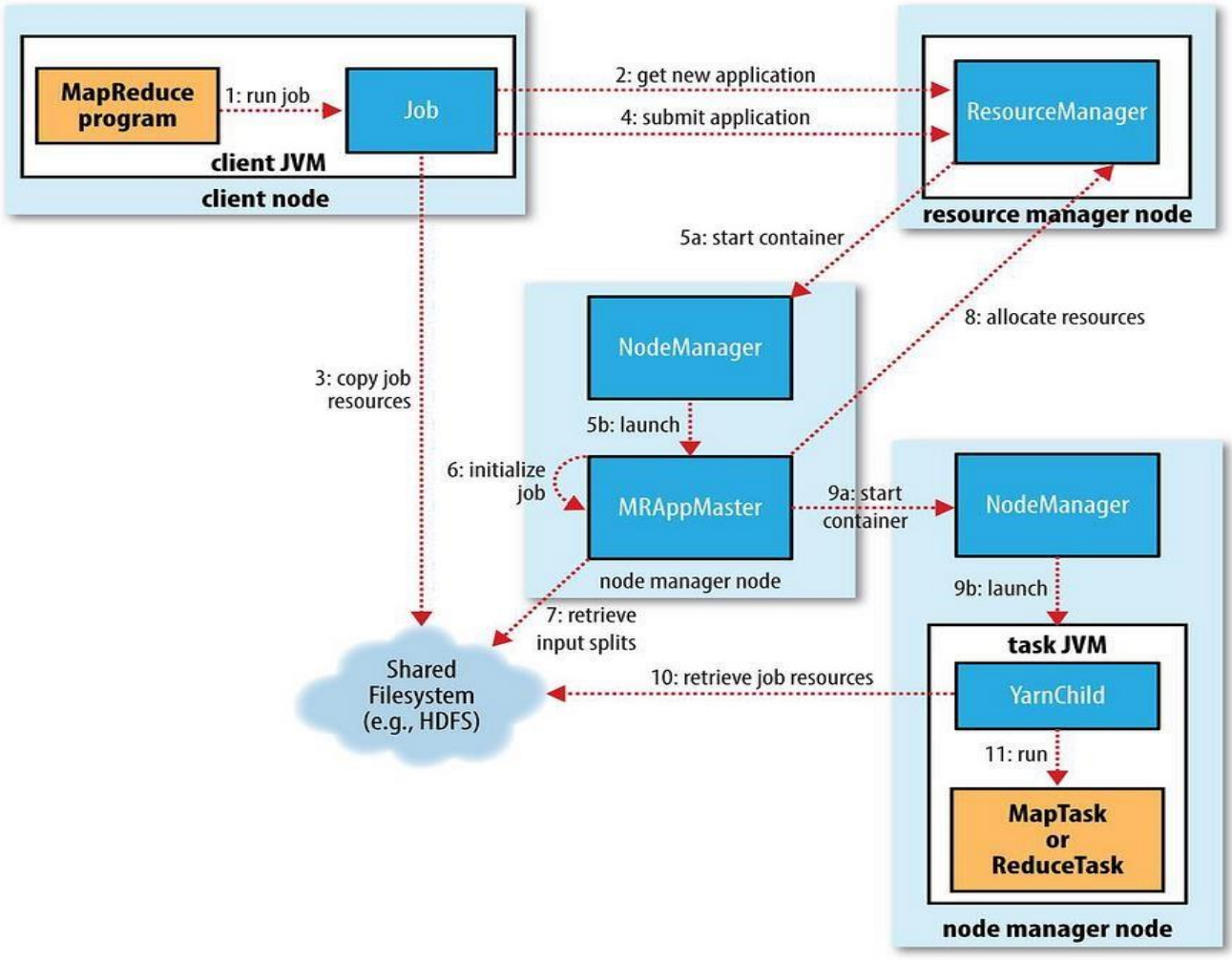
Run the test

To run the test, right click on the class "TestWordCount" and goto "Run as" and select "Junit Test".



If the mapper, reducer and mapReduce job as a whole is correct, then you should get an output where you see no errors or failures.

Anatomy of Map reduce job run



There are five independent entities:

- The client, which submits the `MapReduce` job.
- The YARN resource manager, which coordinates the allocation of compute resources on the cluster.
- The YARN node managers, which launch and monitor the compute containers on machines in the cluster.
- The `MapReduce` application master, which coordinates the tasks running the `MapReduce` job The application master and the `MapReduce` tasks run in containers that are scheduled by the resource manager and managed by the node managers.
- The distributed filesystem, which is used for sharing job files between the other entities.

Job Submission :

- The `submit()` method on Job creates an internal `JobSubmitter` instance and calls `submitJobInternal()` on it.
- Having submitted the job, `waitForCompletion` polls the job's progress once per second and reports the progress to the console if it has changed since the last report.
- When the job completes successfully, the job counters are displayed Otherwise, the error that caused the job to fail is logged to the console.

The job submission process implemented by `JobSubmitter` does the following:

- Asks the resource manager for a new application ID, used for the `MapReduce` job ID.
- Checks the output specification of the job For example, if the output directory has not been specified or it already exists, the job is not submitted and an error is thrown to the `MapReduce` program.
- Computes the input splits for the job If the splits cannot be computed (because the input paths don't exist, for example), the job is not submitted and an error is thrown to the `MapReduce` program.
- Copies the resources needed to run the job, including the job JAR file, the configuration file, and the computed input splits, to the shared filesystem in a directory named after the job ID.
- Submits the job by calling `submitApplication()` on the resource manager.

Job Initialization :

- When the resource manager receives a call to its `submitApplication()` method, it hands off the request to the YARN scheduler.
- The scheduler allocates a container, and the resource manager then launches the application master's process there, under the node manager's management.
- The application master for `MapReduce` jobs is a Java application whose main class is `MRAppMaster`.
- It initializes the job by creating a number of bookkeeping objects to keep track of the job's progress, as it will receive progress and completion reports from the tasks.
- It retrieves the input splits computed in the client from the shared filesystem.
- It then creates a map task object for each split, as well as a number of reduce task objects determined by the `mapreduce.job.reduces` property (set by the `setNumReduceTasks()` method on Job).

Task Assignment:

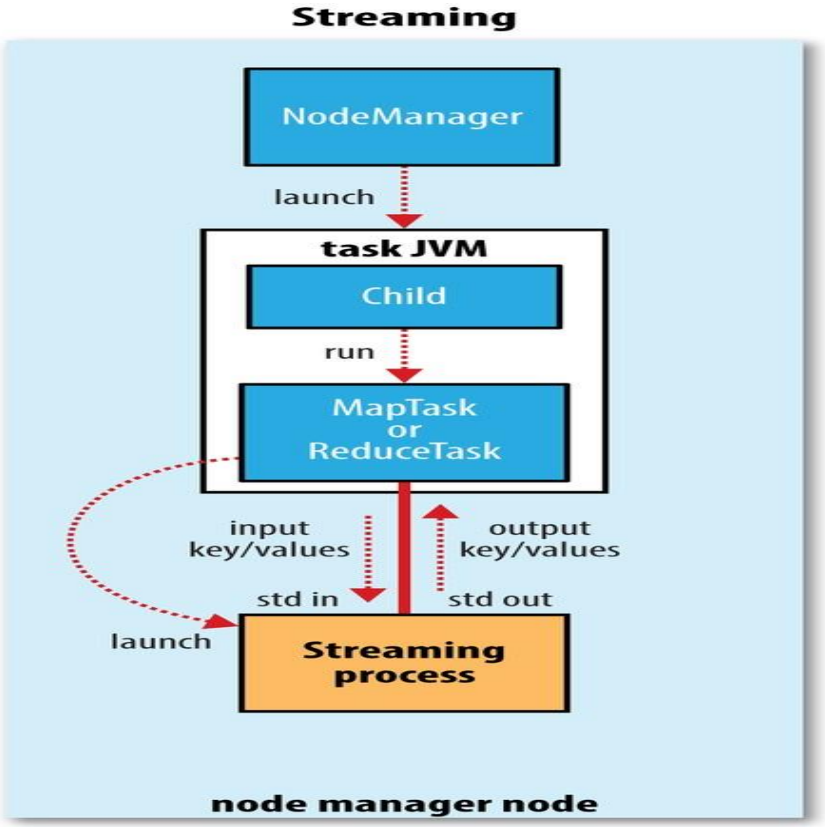
- If the job does not qualify for running as an uber task, then the application master requests containers for all the map and reduce tasks in the job from the resource manager.

- Requests for map tasks are made first and with a higher priority than those for reduce tasks, since all the map tasks must complete before the sort phase of the reduce can start.
- Requests for reduce tasks are not made until 5% of map tasks have completed.

Task Execution:

- Once a task has been assigned resources for a container on a particular node by the resource manager's scheduler, the application master starts the container by contacting the node manager.
- The task is executed by a Java application whose main class is `YarnChild`. Before it can run the task, it localizes the resources that the task needs, including the job configuration and JAR file, and any files from the distributed cache.
- Finally, it runs the map or reduce task.

Streaming:



- Streaming runs special map and reduce tasks for the purpose of launching the user supplied executable and communicating with it.
- The Streaming task communicates with the process (which may be written in any language) using standard input and output streams.

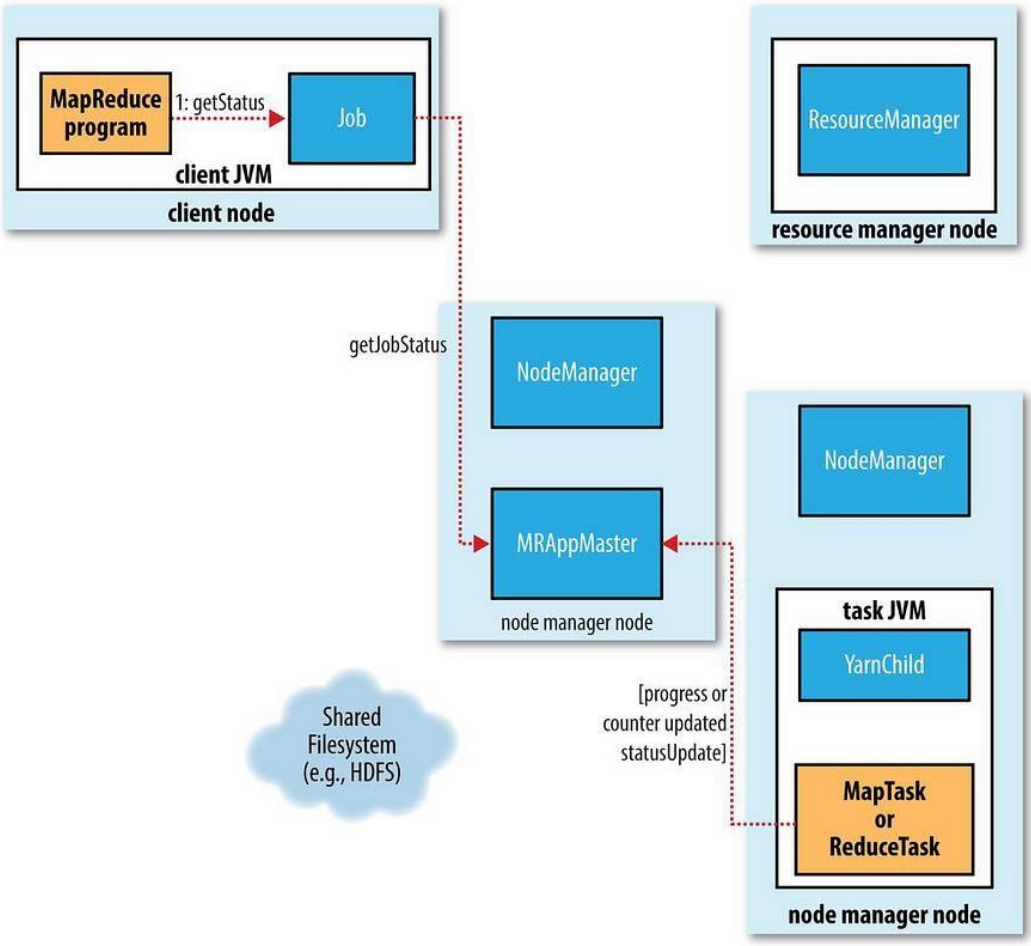
- During execution of the task, the Java process passes input key value pairs to the external process, which runs it through the user defined map or reduce function and passes the output key value pairs back to the Java process.
- From the node manager's point of view, it is as if the child process ran the map or reduce code itself.

Progress and status updates :

- `MapReduce` jobs are long running batch jobs, taking anything from tens of seconds to hours to run.
- A job and each of its tasks have a status, which includes such things as the state of the job or task (e g running, successfully completed, failed), the progress of maps and reduces, the values of the job's counters, and a status message or description (which may be set by user code).
- When a task is running, it keeps track of its progress (i e the proportion of task is completed).
- For map tasks, this is the proportion of the input that has been processed.
- For reduce tasks, it's a little more complex, but the system can still estimate the proportion of the reduce input processed.

It does this by dividing the total progress into three parts, corresponding to the three phases of the shuffle.

- As the map or reduce task runs, the child process communicates with its parent application master through the umbilical interface.
- The task reports its progress and status (including counters) back to its application master, which has an aggregate view of the job, every three seconds over the umbilical interface.



How status updates are propagated through the MapReduce System

- The resource manager web UI displays all the running applications with links to the web UIs of their respective application masters, each of which displays further details on the `MapReduce` job, including its progress.
- During the course of the job, the client receives the latest status by polling the application master every second (the interval is set via `mapreduce.client.progressmonitor.pollinterval`).

Job Completion:

- When the application master receives a notification that the last task for a job is complete, it changes the status for the job to Successful.
- Then, when the Job polls for status, it learns that the job has completed successfully, so it prints a message to tell the user and then returns from the `waitForCompletion()` .
- Finally, on job completion, the application master and the task containers clean up their working state and the OutputCommitter's `commitJob ()` method is called.
- Job information is archived by the job history server to enable later interrogation by users if desired.

Classic map reduce

A job run in classic MapReduce is illustrated in Fig. At the highest level, **there are four independent entities**:

- The client, which submits the MapReduce job.
- The jobtracker, which coordinates the job run. The jobtracker is a Java application whose main class is JobTracker.
- The tasktrackers, which run the tasks that the job has been split into. Tasktrackers are Java applications whose main class is TaskTracker.
- The distributed filesystem , which is used for sharing job files between the other entities.

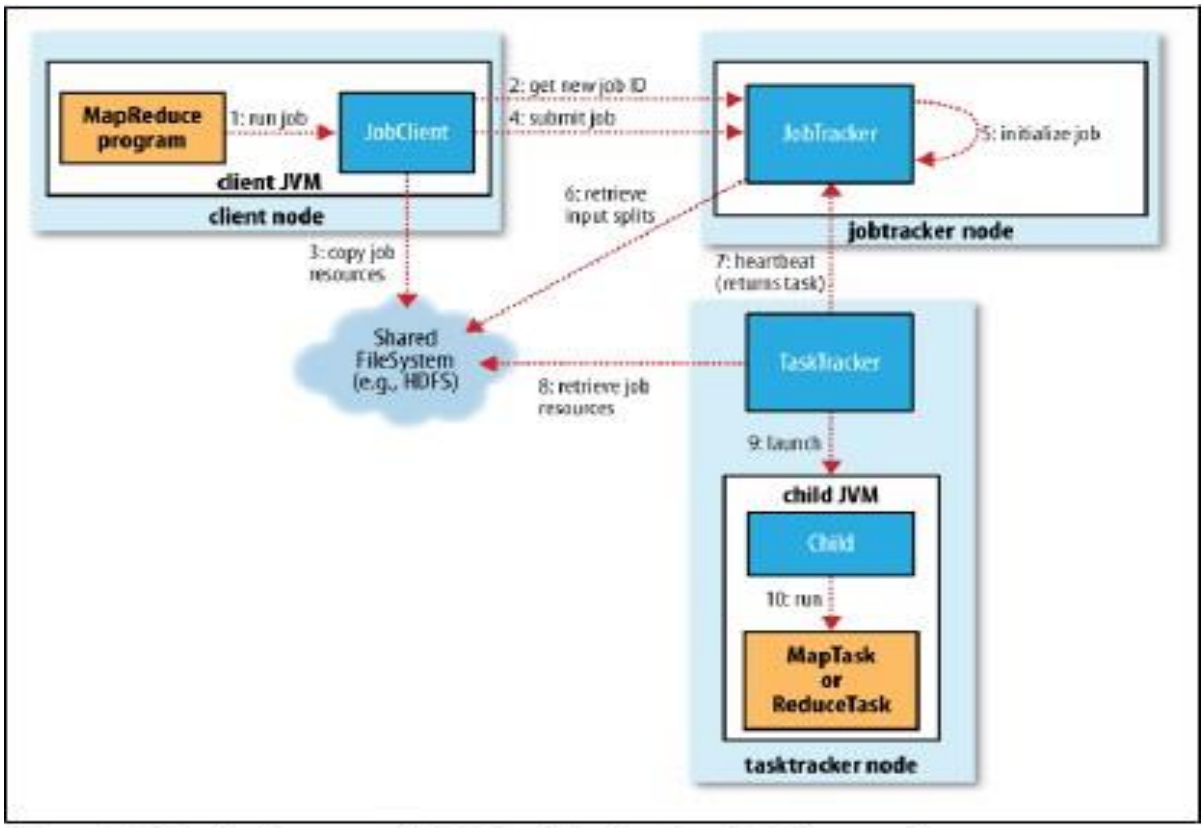


Figure 6-1. How Hadoop runs a MapReduce job using the classic framework

Job Submission:

The submit() method on Job creates an internal JobSummitter instance and calls submitJobInternal() on it (step 1 in Figure).

The job submission process implemented by JobSummitter does the following:

- Asks the jobtracker for a new job ID (by calling getNewJobId() on JobTracker) (step 2).

- Computes the input splits for the job. Copies the resources needed to run the job, including the job JAR file, the configuration file, and the computed input splits, to the jobtracker's filesystem in a directory named after the job ID. (step 3).Tells the jobtracker that the job is ready for execution (by calling submitJob() on JobTracker) (step 4).

Job Initialization:

When the JobTracker receives a call to its submitJob() method, it puts it into an internal queue from where the job scheduler will pick it up and initialize it. Initialization involves creating an object to represent the job being run (step 5).

To create the list of tasks to run, the job scheduler first retrieves the input splits computed by the client from the shared filesystem (step 6). It then creates one map task for each split.

Task Assignment:

Tasktrackers run a simple loop that periodically sends heartbeat method calls to the jobtracker. Heartbeats tell the jobtracker that a tasktracker is alive As a part of the heartbeat, a tasktracker will indicate whether it is ready to run a new task, and if it is, the jobtracker will allocate it a task, which it communicates to the tasktracker using the heartbeat return value (step 7).

Task Execution:

Now that the tasktracker has been assigned a task, the next step is for it to run the task. First, it localizes the job JAR by copying it from the shared filesystem to the tasktracker's filesystem. It also copies any files needed from the distributed cache by the application to the local disk; see (step 8).

TaskRunner launches a new Java Virtual Machine (step 9) to run each task in (step 10).

Progress and Status Updates:

MapReduce jobs are long-running batch jobs, taking anything from minutes to hours to run. Because this is a significant length of time, it's important for the user to get feedback on how the job is progressing. A job and each of its tasks have a *status.*

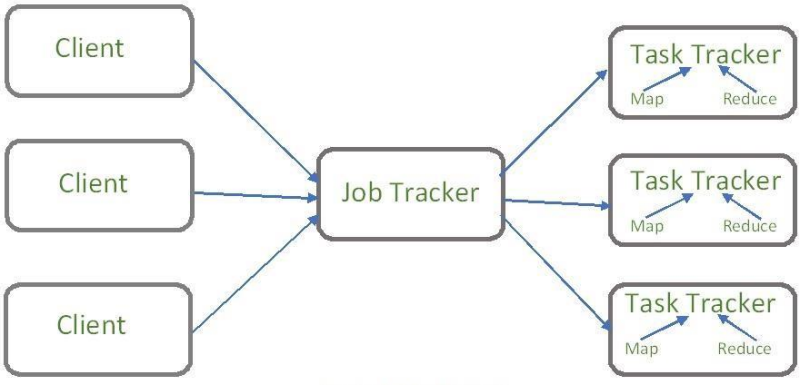
When a task is running, it keeps track of its *progress*, that is, the proportion of the task completed.

Job Completion:

When the jobtracker receives a notification that the last task for a job is complete (this will be the special job cleanup task), it changes the status for the job to "successful."

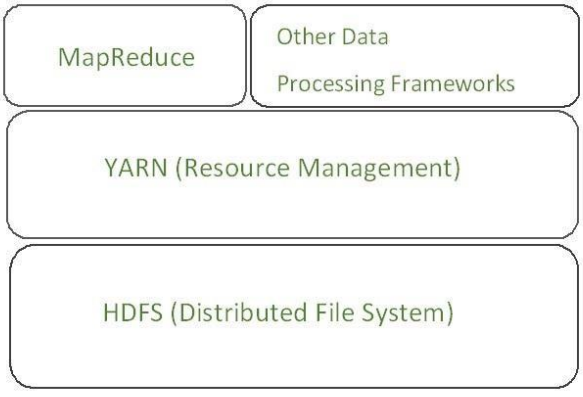
YARN Architecture

YARN stands for "Yet Another Resource Negotiator". It was introduced in Hadoop 2.0 to remove the bottleneck on Job Tracker which was present in Hadoop 1.0. YARN was described as a "Redesigned Resource Manager" at the time of its launching, but it has now evolved to be known as large-scale distributed operating system used for Big Data processing.



Hadoop 1.0 architecture

YARN architecture basically separates resource management layer from the processing layer. In Hadoop 1.0 version, the responsibility of Job tracker is split between the resource manager and application manager.



Hadoop 2.0

YARN also allows different data processing engines like graph processing, interactive processing, stream processing as well as batch processing to run and process data stored in HDFS (Hadoop Distributed File System) thus making the system much more efficient. Through its various components, it can dynamically allocate various resources and schedule the application processing. For large volume data processing, it is quite necessary to manage the available resources properly so that every application can leverage them.

YARN Features: YARN gained popularity because of the following features
Scalability: The scheduler in Resource manager of YARN architecture allows Hadoop to extend and manage thousands of nodes and clusters.
Compatibility: YARN supports the existing map-reduce applications without disruptions thus making it compatible with Hadoop 1.0 as well.
Cluster Utilization: Since YARN supports Dynamic utilization of cluster in Hadoop, which enables optimized Cluster Utilization.
Multi-tenancy: It allows multiple engine access thus giving organizations a benefit of multitenancy.

Hadoop YARN Architecture

Hadoop Yarn architecture

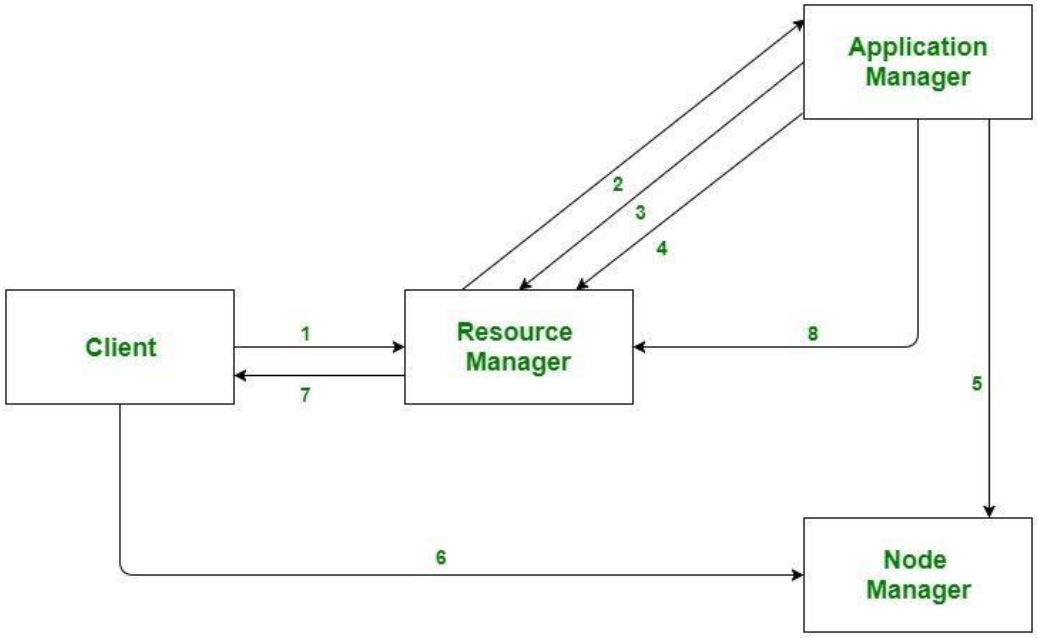
The main components of YARN architecture include:
Client: It submits map-reduce jobs.
Resource Manager: It is the master daemon of YARN and is responsible for resource assignment and management among all the applications. Whenever it receives a processing request, it forwards it to the corresponding node manager and allocates resources for the completion of the request accordingly. It has two major components:
Scheduler: It performs scheduling based on the allocated application and available resources. It is a pure scheduler, means it does not perform other tasks such as monitoring or tracking and does not guarantee a restart if a task fails. The YARN scheduler supports plugins such as Capacity Scheduler and Fair Scheduler to partition the cluster resources.
Application manager: It is responsible for accepting the application and negotiating the first container from the resource manager. It also restarts the Application Master container if a task fails.

Node Manager: It take 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.

Container: It is a collection of physical resources such as RAM, CPU cores and disk on a single node. The containers are invoked by Container Launch Context(CLC) which is a record that contains information such as environment variables, security tokens, dependencies etc.

Application workflow in Hadoop YARN:



1. Client submits an application
2. The Resource Manager allocates a container to start the Application Manager
3. The Application Manager registers itself with the Resource Manager
4. The Application Manager negotiates containers from the Resource Manager
5. The Application Manager notifies the Node Manager to launch containers
6. Application code is executed in the container
7. Client contacts Resource Manager/Application Manager to monitor application's status
8. Once the processing is complete, the Application Manager un-registers 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.

Disadvantages :

Complexity: YARN adds complexity to the Hadoop ecosystem. It requires additional configurations and settings, which can be difficult for users who are not familiar with YARN.
Overhead: YARN introduces additional overhead, which can slow down the performance of the Hadoop cluster. This overhead is required for managing resources and scheduling applications.
Latency: YARN introduces additional latency in the Hadoop ecosystem. This latency can be caused by resource allocation, application scheduling, and communication between components.
Single Point of Failure: YARN can be a single point of failure in the Hadoop cluster. If YARN fails, it can cause the entire cluster to go down. To avoid this, administrators need to set up a backup YARN instance for high availability.
Limited Support: YARN has limited support for non-Java programming languages. Although it supports multiple processing engines, some engines have limited language support, which can limit the usability of YARN in certain environments.

Failures in classic Map-reduce and YARN

There are generally 3 types of failures in mapreduce.

1. Task Failure
2. TaskTracker Failure
3. JobTracker Failure

1. Task Failure

In Hadoop, task failure is similar to an employee making a mistake while doing a task. Consider you are working on a large project that has been broken down into smaller jobs and assigned to

different employees in your team. If one of the team members fails to do their task correctly, the entire project may be compromised. Similarly, in Hadoop, if a job fails due to a mistake or issue, it could affect overall data processing, causing delays or faults in the final result.

Reasons for Task Failure

Here are some reasons for Task failure.

Limited memory: A task can fail if it runs out of memory while processing data.

Failures of disk: If the disk that stores data or intermediate results fails, tasks that depend on that data may fail.

Issues with software or hardware: Bugs, mistakes, or faults in software or hardware components can cause task failures.

How to Overcome Task Failure

Increase memory allocation: Assign extra memory to jobs to ensure they have the resources to process the data.

Implement fault tolerance mechanisms: Using data replication and checkpointing techniques to defend against disc failures and retrieve lost data.

Regularly update software and hardware: Keep the Hadoop framework and supporting hardware up to date to fix bugs, errors, and performance issues that can lead to task failures.

2. **TaskTracker Failure**

A TaskTracker in Hadoop is similar to an employee responsible for executing certain tasks in a large project. If a TaskTracker fails, it signifies a problem occurred while an employee worked on their assignment. This can interrupt the entire project, much as when a team member makes a mistake or encounters difficulties with their task, producing delays or problems with the overall project's completion. To avoid TaskTracker failures, ensure the TaskTracker's hardware and software are in excellent working order and have the resources they need to do their jobs successfully.

Reasons for TaskTracker Failure

Here are some reasons for TaskTracker failure.

Hardware issues: Just as your computer's parts can break or stop working properly, the TaskTracker's hardware (such as the processor, memory, or disc) might fail or stop operating properly. This may prohibit it from carrying out its duties.

Software problems or errors: The software operating on the TaskTracker may contain bugs or errors that cause it to cease working properly. It's similar to when an app on your phone fails and stops working properly.

Overload or resource exhaustion: It may struggle to keep up if the TaskTracker becomes overburdened with too many tasks or runs out of resources such as memory or processing power. It's comparable to being overburdened with too many duties or running out of storage space on your gadget.

How to Overcome TaskTracker Failure

Update software and hardware on a regular basis: Keep the Hadoop framework and associated hardware up to date to correct bugs, errors, and performance issues that might lead to task failures.

Upgrade or replace hardware: If TaskTracker's hardware is outdated or insufficiently powerful, try upgrading or replacing it with more powerful components. It's equivalent to purchasing a new, upgraded computer to handle jobs more efficiently.

Restart or reinstall the program: If the TaskTracker software is causing problems, a simple restart or reinstall may be all that is required. It's the same as restarting or reinstalling an app to make it work correctly again.

3. JobTracker Failure

A JobTracker in Hadoop is similar to a supervisor or manager that oversees the entire project and assigns tasks to TaskTrackers (employees). If a JobTracker fails, it signifies the supervisor is experiencing a problem or has stopped working properly. This can interrupt the overall project's coordination and development, much as when a supervisor is unable to assign assignments or oversee their completion. To avoid JobTracker failures, it is critical to maintain the JobTracker's hardware and software, ensure adequate resources, and fix any issues or malfunctions as soon as possible to keep the project going smoothly.

Reasons for JobTracker Failure

Here are some reasons for JobTracker failure.

Database connectivity: The JobTracker stores job metadata and state information in a backend database (usually Apache Derby or MySQL). JobTracker failures can occur if there are database connectivity issues, such as network problems or database server failures.

Security problems: JobTracker failures can be caused by security issues such as authentication or authorization failures, incorrectly configured security settings or key distribution and management issues.

How to Overcome JobTracker Failure

Avoiding Database Connectivity: To avoid database connectivity failures in the JobTracker, ensure optimized database configuration, robust network connections, and high availability techniques are implemented. Retrying connections, monitoring, and backups are all useful. **To overcome security-related problems:** implement strong authentication and authorization, enable SSL/TLS for secure communication, keep software updated with security patches, follow key management best practices, conduct security audits, and seek expert guidance for vulnerability mitigation and compliance with security standards.

Failures in YARN

Dealing with failures in distributed systems is comparatively more challenging and time consuming. Also, the Hadoop and YARN frameworks run on commodity hardware and cluster size nowadays; this size can vary from several nodes to several thousand nodes. So handling failure

scenarios and dealing with ever-growing scaling issues is very important. In this section, we will focus on failures in the YARN framework: the causes of failures and how to overcome them.

ResourceManager failures
ApplicationMaster failures
NodeManager failures
Container failures
Hardware failures

Job scheduling

MapReduce is a framework that helps us to write applications to process large amounts of data in parallel on huge clusters of commodity hardware in an authentic manner.

The MapReduce algorithm consists of two essential tasks, Map & Reduce.

The Map takes a set of data and turns it into a different set of data, in which individual elements are divided into tuples. While Reduce takes the output from a map as an input and merges those data tuples into a smaller set of tuples. As the name suggests, the Map job is done before the Reduce task.

Hadoop Schedulers

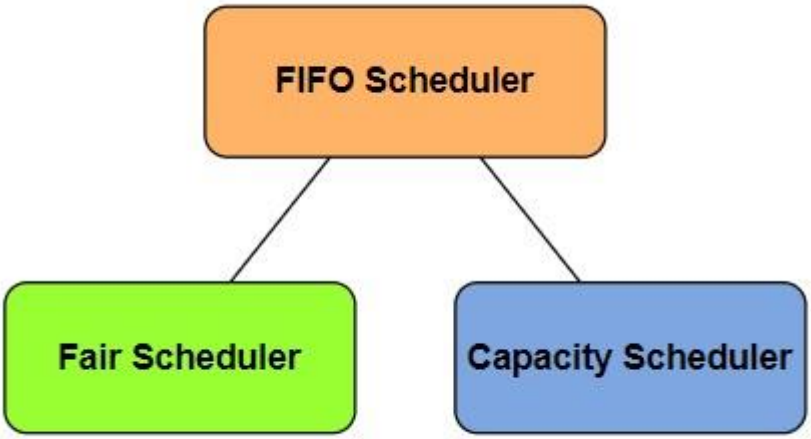
Hadoop is a general-purpose system that allows high-performance data processing over a set of distributed nodes. Besides, it is a multi-tasking system that processes multiple data sets for different jobs for multiple users in parallel. Earlier, in Hadoop, there was a single schedular supported, which was intermixed with the JobTracker logic. This implementation was perfect for the traditional batch jobs in Hadoop.

For scheduler users' jobs, previous versions had a very simple way. Generally, they ran in order of submission using a Hadoop FIFO scheduler.

Types of Job Scheduling in MapReduce

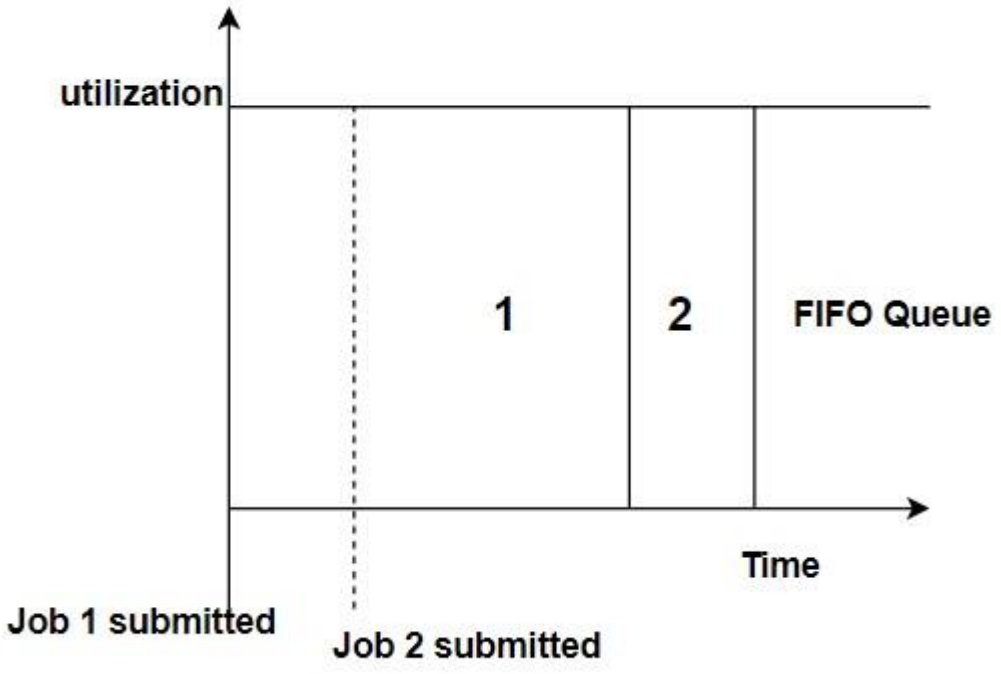
There are mainly three different types of job scheduling in MapReduce:

1. First-in-first-out (FIFO)
2. Capacity Scheduler
3. Fair Scheduler


FIFO Scheduler
Fair Scheduler
Capacity Scheduler

1.

FIFO Scheduler



In Hadoop, FIFO is the default scheduler policy used. This scheduler gives preferences to the tasks coming first than those coming later. This scheduler keeps the application in the queue, and in order of their submission, it executes them. (first in, first out). Despite priority and size in this scheduler, the request of the first tasks in the queue is allocated first. The next task in the queue is served only when the first task is satisfied.

Advantages

- Jobs are served according to their submission.
- This scheduler is easy to understand also does not require any configuration.

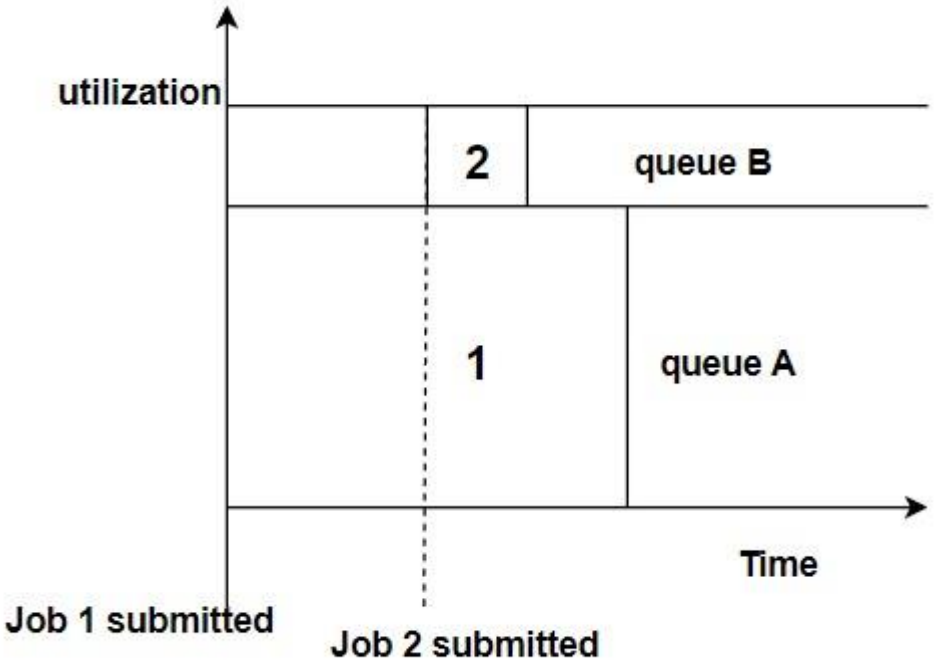
Disadvantages

For shared clusters, this scheduler might not work best. If the larger tasks come before, the shorter task, then the larger tasks will use all the resources in the cluster. Due to this, the shorter tasks will be in the queue for a longer time and has to wait for their turn, which will lead to starvation.

The balance of resource allocation between long and short applications is not considered.

2.

Capacity Scheduler



This scheduler permits multiple tenants to share a huge Hadoop cluster securely. This scheduler supports hierarchical queues to portray the structure of groups/organizations that utilizes the resources of the cluster. This queue hierarchy consists of three types of queues: root, parent, and leaf.

The root queue means the cluster itself, the parent queue means the group or organization, or sub-group or sub-organizations, and the leaf queue accepts application submissions. The capacity scheduler enables the sharing of the large cluster while providing capacity assurance to every organization by allocating a fraction of cluster resources to every queue.

Also, whenever there is a request for free resources already present on the queue who have completed their tasks, these resources are assigned to the applications on queues running below capacity. This gives elasticity to the organization in a cost-effective way.

Advantages

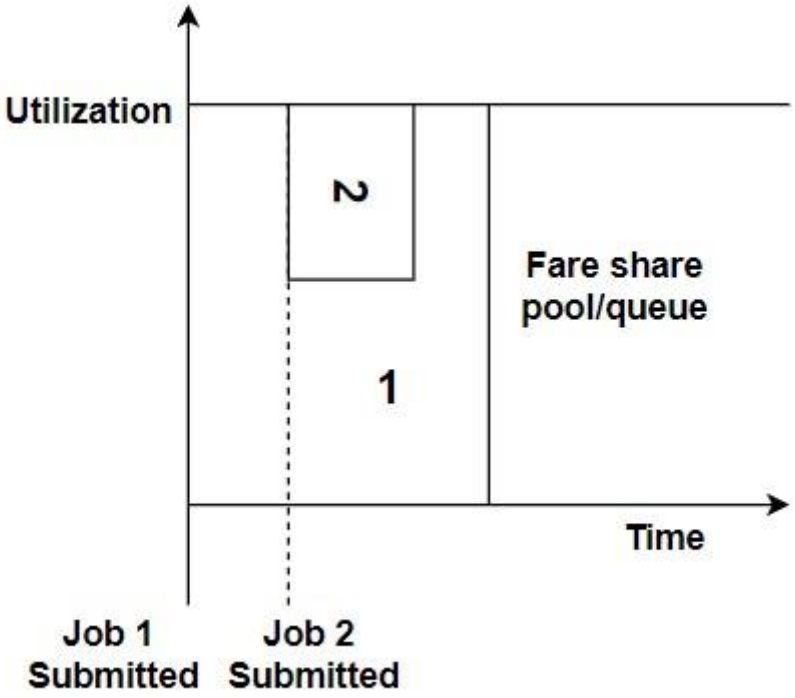
- This scheduler provides a capacity assurance and safeguards to the organization utilizing cluster.
- It maximizes the throughput and utilization of resources in the Hadoop cluster.

Disadvantages

Compared to the other two schedulers, a capacity scheduler is considered complex.

3.

Fair Scheduler



A fair scheduler permits YARN applications to share resources in large Hadoop clusters evenly. With this scheduler, you are not required to reserve a set amount of capacity because it dynamically balances resources between all the ongoing applications. All the resources in this scheduler are assigned in such a way that all the applications get an equal amount of resources. By default, this scheduler takes scheduling fairness decisions only based on memory. The entire cluster resources are used when the single application is running. When the other tasks are submitted, the free-up resources get assigned to the new apps in order to distribute the same amount of resources for each application. It enables the short app to complete in an adequate amount of time without starving the long-lived apps.

Same as Capacity scheduler also supports a hierarchical queue to portray the structure of the long shared cluster.

In this scheduler, when an application is present in the queue, then the application gets its minimum share, but when the full guaranteed share is not required, then the excess share is distributed between other ongoing applications.

4.

Advantages

- It gives a reasonable way to share the cluster between the no. of users.
- The fair scheduler can work with application priorities. Priorities are used as a weight to recognize the fraction of the total resources every application must get.

Disadvantages

- Configuration is required.

When to use Each Job Scheduling in MapReduce

- The capacity scheduler is the correct choice because we want to secure guaranteed access with the potential in order to reuse unused capacity.
- The fair scheduler works well when we use large and small clusters for the same organization with limited workloads. Also, it is helpful in the presence of various jobs.

Limitations of Job Scheduling in MapReduce

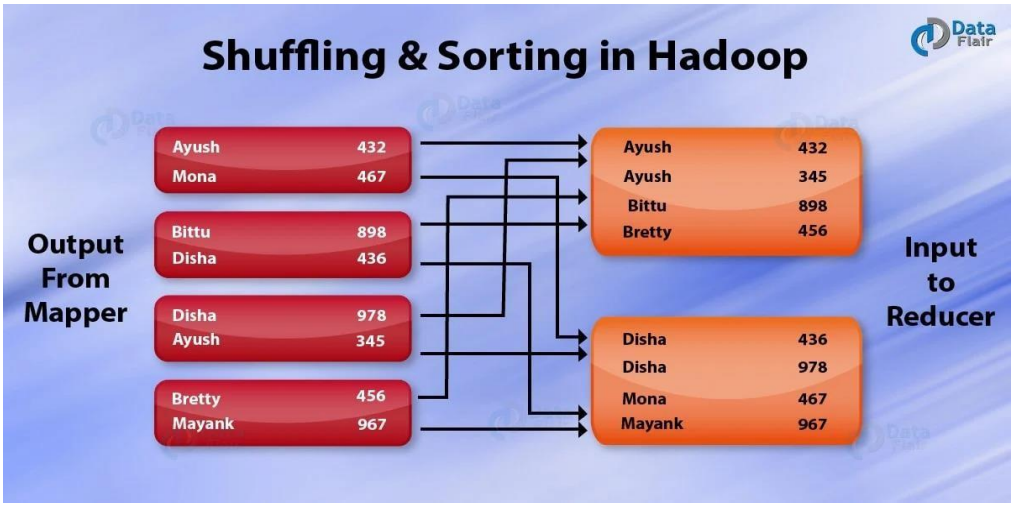
In Hadoop, whole storage is accomplished at HDFS. When the client requests a MapReduce job, then the master node(name node) transfers the MapReduce code to the slave's node, i.e., to the node in which the real data connected to the job exists. Due to large data sets, the issue of cross-switch network traffic was common in Hadoop. To handle this problem, the concept of data locality came into the picture.

Data locality transfers the computation near the node where the real/actual data exists. This not only rises the throughput but also decreases the network traffic.

Shuffling and Sorting in Hadoop MapReduce

In Hadoop, the process by which the intermediate output from mappers is transferred to the reducer is called Shuffling. Reducer gets 1 or more keys and associated values on the basis of reducers. Intermediated key-value generated by mapper is sorted automatically by key. In this blog, we will discuss in detail about shuffling and Sorting in Hadoop MapReduce.

Here we will learn what is sorting in Hadoop, what is shuffling in Hadoop, what is the purpose of Shuffling and sorting phase in MapReduce, how MapReduce shuffle works and how MapReduce sort works.



What is Shuffling and Sorting in Hadoop MapReduce?

Before we start with Shuffle and Sort in MapReduce, let us revise the other phases of MapReduce like Mapper, reducer in MapReduce, Combiner, partitioner in MapReduce and inputFormat in MapReduce.

Shuffle phase in Hadoop transfers the map output from Mapper to a Reducer in MapReduce. Sort phase in MapReduce covers the merging and sorting of map outputs. Data from the mapper are grouped by the key, split among reducers and sorted by the key. Every reducer obtains all values associated with the same key. Shuffle and sort phase in Hadoop occur simultaneously and are done by the MapReduce framework.

Shuffling in MapReduce

The process of transferring data from the mappers to reducers is known as shuffling i.e. the process by which the system performs the sort and transfers the map output to the reducer as input. So, MapReduce shuffle phase is necessary for the reducers, otherwise, they would not have any input (or input from every mapper). As shuffling can start even before the map phase has finished so this saves some time and completes the tasks in lesser time.

Sorting in MapReduce

The keys generated by the mapper are automatically sorted by MapReduce Framework, i.e. Before starting of reducer, all intermediate key-value pairs in MapReduce that are generated by mapper get sorted by key and not by value. Values passed to each reducer are not sorted; they can be in any order.

Sorting in Hadoop helps reducer to easily distinguish when a new reduce task should start. This saves time for the reducer. Reducer starts a new reduce task when the next key in the sorted input data is different than the previous. Each reduce task takes key-value pairs as input and generates key-value pair as output.

Note that shuffling and sorting in Hadoop MapReduce is not performed at all if you specify zero reducers (setNumReduceTasks(0)). Then, the MapReduce job stops at the map phase, and the map phase does not include any kind of sorting (so even the map phase is faster).

MapReduce types

MapReduce is a computer technique that is used to create and process massive data sets. It includes unique formats for both input and output. It is important to remember that the availability and use of input and output formats are determined by the specific MapReduce types implementation as well as the tools or libraries used in your data processing environment.

What is MapReduce in Hadoop?

MapReduce is an important component of Hadoop. MapReduce is a massively parallel data processing system that processes dispersed data in a quicker, scalable, and fault-tolerant manner. By breaking work into independent sub-tasks, MapReduce can process a massive volume of data in parallel. MapReduce programs developed in languages such as Java, Ruby, Python, and C++ can be run by Hadoop.

The MapReduce algorithm divides the processing into two tasks: map and reduce. The reduction task occurs after the map task has been done, as the term MapReduce implies. Each task has keyvalue pairs as input and output, which the programmer can customize. Map translates one collection of data into another, where individual items are broken down into key-value pairs. The reduce task then takes the output of a map as input in the form of key-value pairs and merges it into a smaller collection of key-value pairs.

MapReduce Phases

The MapReduce program is executed in three main phases: the mapping phase, the shuffling and sorting phase, and the reducing phase.

Map phase: This is the program's first phase. This phase consists of two steps: splitting and mapping. For efficiency, the input file is separated into smaller equal portions known as input splits. Because Mappers only understand (key, value) pairs, Hadoop employs a RecordReader that uses TextInputFormat to convert input splits into key-value pairs.

Shuffle and sorting phase: Shuffle and sort are intermediary phases in MapReduce. The Shuffle process aggregates all Mapper output by grouping important Mapper output values, and the value is appended to a list of values. So, the Shuffle output format will be a map <key, List<list of values>>. The Mapper output key will be combined and sorted.

Reduce phase: The result of the shuffle and sorting phases is sent as input into the Reducer phase, which processes the list of values. Each key could be routed to a distinct Reducer. Reducers can set the value, which is then consolidated in the final output of a MapReduce job and saved in HDFS as the final output.

What is Hadoop InputFormat?

Hadoop InputFormat provides the input specification for Map-Reduce job execution. InputFormat specifies how to divide and read input files. InputFormat is the initial phase in MapReduce job execution. It is also in charge of generating input splits and separating them into records. One of the core classes in MapReduce is InputFormat, which provides the following functionality:

InputFormat determines which files or other objects to accept as input.
It also specifies data splits. It specifies the size of each Map task as well as the potential execution server.

The RecordReader is defined by Hadoop InputFormat. It is also in charge of reading actual records from input files.

Types of Input Format in MapReduce

In Hadoop, there are various MapReduce types for InputFormat that are used for various purposes. Let us now look at the MapReduce types of InputFormat:

FileInputFormat
It serves as the foundation for all file-based InputFormats. FileInputFormat also provides the input directory, which contains the location of the data files. When we start a MapReduce task, FileInputFormat returns a path with files to read. This InpuFormat will read all files. Then it divides these files into one or more InputSplits.

TextInputFormat
It is the standard InputFormat. Each line of each input file is treated as a separate record by this InputFormat. It does not parse anything. TextInputFormat is suitable for raw data or line-based records, such as log files. Hence:

Key: It is the byte offset of the first line within the file (not the entire file split). As a result, when paired with the file name, it will be unique.

Value: It is the line's substance. It does not include line terminators.

KeyValueTextInputFormat
It is comparable to TextInputFormat. Each line of input is also treated as a separate record by this InputFormat. While TextInputFormat treats the entire line as the value, KeyValueTextInputFormat divides the line into key and value by a tab character ('/t'). Hence:

Key: Everything up to and including the tab character.

Value: It is the remaining part of the line after the tab character.

SequenceFileInputFormat
It's an input format for reading sequence files. Binary files are sequence files. These files also store binary key-value pair sequences. These are block-compressed and support direct serialization and deserialization of a variety of data types. Hence Key & Value are both user-defined.

SequenceFileAsTextInputFormat
It is a subtype of SequenceFileInputFormat. The sequence file key values are converted to Text objects using this format. As a result, it converts the keys and values by running 'tostring()' on them. As a result, SequenceFileAsTextInputFormat converts sequence files into text-based input for streaming.

NlineInputFormat
It is a variant of TextInputFormat in which the keys are the line's byte offset. And values are the line's contents. As a result, each mapper receives a configurable number of lines of TextInputFormat and KeyValueTextInputFormat input. The number is determined by the magnitude of the split. It is also dependent on the length of the lines. So, if we want our mapper to accept a specific amount of lines of input, we use NLineInputFormat.

N- It is the number of lines of input received by each mapper.

Each mapper receives exactly one line of input by default (N=1).

Assuming N=2, each split has two lines. As a result, the first two Key-Value pairs are distributed to one mapper. The second two key-value pairs are given to another mapper.

DB InputFormat
Using JDBC, this InputFormat reads data from a relational database. It also loads small datasets, which might be used to connect with huge datasets from HDFS using multiple inputs. Hence:

Key: LongWritables

Value: DBWritables.

Output Format in MapReduce
The output format classes work in the opposite direction as their corresponding input format classes. The TextOutputFormat, for example, is the default output format that outputs records as plain text files, although key values can be of any type and are converted to strings by using the toString() method. The tab character separates the key-value character, but this can be changed by modifying the separator attribute of the text output format.

SequenceFileOutputFormat is used to write a sequence of binary output to a file for binary output. Binary outputs are especially valuable if they are used as input to another MapReduce process.

DBOutputFormat handles the output formats for relational databases and HBase. It saves the compressed output to a SQL table.

Example

An example of MapReduce types of InputFormat is given below:

```
import org.apache.hadoop.conf.Configuration; import
org.apache.hadoop.fs.Path; import
org.apache.hadoop.mapreduce.Job; import
org.apache.hadoop.mapreduce.Mapper; import
org.apache.hadoop.mapreduce.lib.input.TextInputFormat; import
org.apache.hadoop.mapreduce.lib.output.TextOutputFormat; import
org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class Text_input_output_example {

  public static class MyMapper extends Mapper<LongWritable, Text, Text, Text> {

    protected void map(LongWritable key, Text value, Context context) throws IOException,
InterruptedException {
      // Put the map logic here.
      context.write(new Text("OutputKey"), value);
    }
  }

  public static void main(String[] args) throws Exception {
// Construct a Hadoop configuration.
    Configuration conf = new Configuration();

    Job job = Job.getInstance(conf, "Text_input_output_example");
job.setJarByClass(Text_input_output_example.class);

    // TextInputFormat should be used as the input format class.
    job.setInputFormatClass(TextInputFormat.class);

    job.setMapperClass(MyMapper.class);

    job.setMapOutputKeyClass(Text.class);
    job.setMapOutputValueClass(Text.class);

    // TextOutputFormat should be used as the output format class.
    job.setOutputFormatClass(TextOutputFormat.class);
```

```
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(Text.class);

    TextInputFormat.addInputPath(job, new Path("input_dir"));
    TextOutputFormat.setOutputPath(job, new Path("output_dir"));

    // Wait for the job to finish before exiting.
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

The code above uses the TextInputFormat to read input data from plain text files. You can specify your map logic in the MyMapper class, which is the mapper implementation. The result is saved as plain text files because the output format is set to TextOutputFormat.