CIT650 Introduction to Big Data

HADOOP VS YARN

Hadoop Environment Setup

.bash_profile

```
# Hadoop
export HADOOP_HOME=/Users/tom/Downloads/hadoop-3.2.3
export HADOOP_INSTALL=$HADOOP_HOME
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME export YARN_HOME=$HADOOP_HOME
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native
export HADOOP_OPTS="-Djava.library.path=$HADOOP_HOME/lib/nativ"
export PATH="$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin”

# Java
JAVA_HOME="/Library/Java/JavaVirtualMachines/jdk-11.0.17.jdk/Contents/Home"
export JAVA_HOME
export PATH="$PATH:$JAVA_HOME/bin"
```

Hadoop Environment Setup

hadoop-env.sh

```
export JAVA_HOME/Library/Java/JavaVirtualMachines/jdk-11.0.17.jdk/Contents/Home
```

core-site.xml

```
<configuration>
   <property>
      <name>fs.defaultFS</name>
      <value>hdfs://localhost:9000</value>
    </property>
</configuration>
```

hdfs-site.xml

```
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value>
  </property>
</configuration>
```

Hadoop Environment Setup

mapred-site.xml

```
<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
    <property>
       <name>mapreduce.application.classpath</name>
       <value>
        $HADOOP_MAPRED_HOME/share/hadoop/mapreduce/*:$HADOOP_MAPRED_HOME/share/hadoop/mapreduce/lib/*
       </value>
    </property>
</configuration>
```

Hadoop Environment Setup

yarn-site.xml

```
<configuration>
  <property>
      <name>yarn.nodemanager.aux-services</name>
      <value>mapreduce_shuffle</value>
  </property>
  <property>
      <name>yarn.nodemanager.env-whitelist</name>
      <value>
JAVA_HOME,HADOOP_COMMON_HOME,HADOOP_HDFS_HOME,HADOOP_CONF_DIR,CLASSPATH_PREPEND_DISTC
ACHE,HADOOP_YARN_HOME,HADOOP_MAPRED_HOME
    </value>
  </property>
</configuration>
```

Application Example

```
public class WCount_Mapper extends MapReduceBase implements Mapper<LongWritable,Text,Text,IntWritable>{
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    public void map(LongWritable key, Text value,OutputCollector<Text,IntWritable> output,
                    Reporter reporter) throws IOException{

        String line = value.toString();
        StringTokenizer  tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()){
            word.set(tokenizer.nextToken());
            output.collect(word, one);
        }
    }

}
```

```
public class WCount_Reducer  extends MapReduceBase implements Reducer<Text,IntWritable,Text,IntWritable> {
    public void reduce(Text key, Iterator<IntWritable> values,OutputCollector<Text,IntWritable> output,
                       Reporter reporter) throws IOException {

        int sum=0;
        while (values.hasNext()) {
            sum+=values.next().get();
        }
        output.collect(key,new IntWritable(sum));
    }
}
```

Application Example

```java
public class WCount_Runner {
    public static void main(String[] args) throws IOException{
        JobConf conf = new JobConf(WCount_Runner.class);
        conf.setJobName("WordCount");
        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(IntWritable.class);
        conf.setMapperClass(WCount_Mapper.class);
        conf.setCombinerClass(WCount_Reducer.class);
        conf.setReducerClass(WCount_Reducer.class);
        conf.setInputFormat(TextInputFormat.class);
        conf.setOutputFormat(TextOutputFormat.class);

        FileInputFormat.setInputPaths(conf, new Path(args[0])); //input file
        Path outputPath = new Path(args[1]); //output file
        FileSystem fs = FileSystem.get(conf);
        if (fs.exists(outputPath)) fs.delete(outputPath, true);
            FileOutputFormat.setOutputPath(conf, outputPath);
        JobClient.runJob(conf);
    }
}
```

```
zip -d wordcount.jar META-INF/LICENSE
```

*Bug in hadoop

```
hadoop jar wordcount.jar /user/tom/data.txt /user/tom/data.out
```

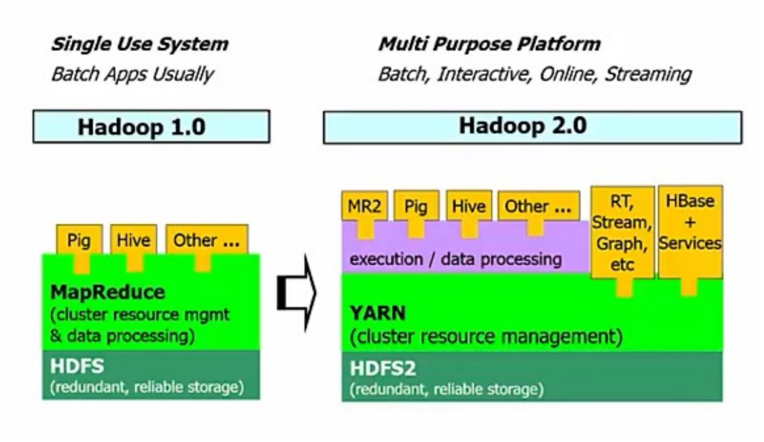
input file

output file

Yet Another Resource Negotiator (YARN)

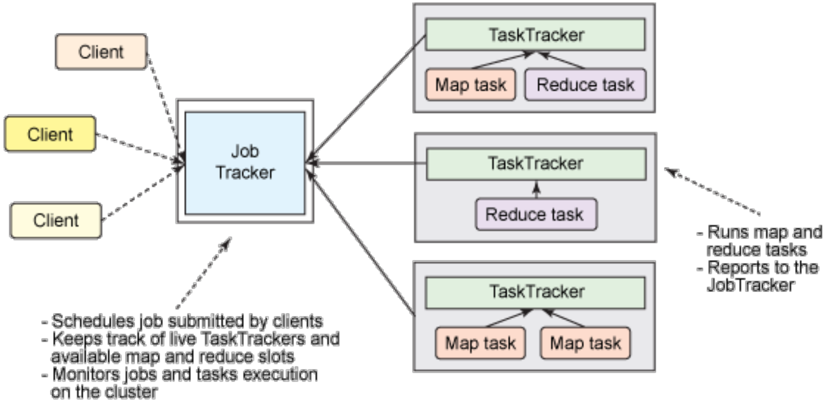
- In Hadoop versions prior to 2.0:
    - Lacked separation of concerns
    - MapReduce engine and JobTracker were tightly coupled
    - HDFS operated as a single monolithic system
- Hadoop 2.0 introduced a modular approach with separate subsystems:
    - MapReduce stayed as part pf Hadoop
    - YARN (Yet Another Resource Negotiator) is introduced including:
        - Resource Manager
        - Node Manager
        - Job Scheduling
        - Application Master
    - This modular design improved scalability and resource management in Hadoop.

Hadoop vs YARN



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

Hadoop Architecture



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

Hadoop Architecture

Client:
•Submits MapReduce jobs
•Queries job status

JobTracker:
•Central coordinator for job scheduling
•Manages task allocation and cluster resources
•Monitors task execution and retries on failure

TaskTracker:
•Executes tasks assigned by JobTracker
•Reports task status back to JobTracker
•Hosts a fixed number of map and reduce slots

Process Flow

1. Clients submit jobs to the JobTracker.

2. JobTracker assigns tasks to TaskTrackers based on:
    - data location
    - slot availability

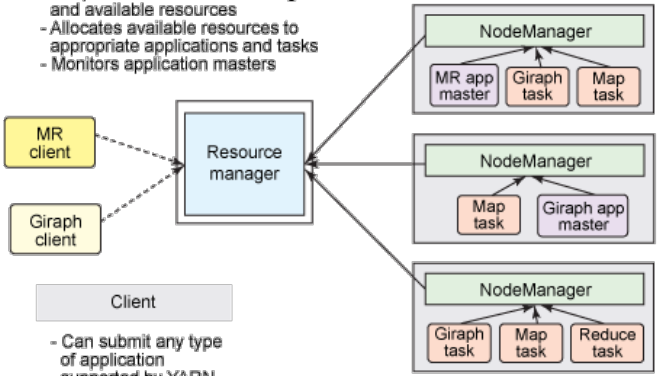
3. TaskTrackers execute tasks

4. TaskTrackers communicate progress and completion back to JobTracker

YARN Architecture

ResourceManager (RM)

- Keeps track of live NodeManagers and available resources
- Allocates available resources to appropriate applications and tasks
- Monitors application masters



Client

- Can submit any type of application supported by YARN

NodeManager (NM)

- Provides computational resources in form of containers
- Managers processes running in containers

ApplicationMaster (AM)

- Coordinates the execution of all tasks within its application
- Asks for appropriate resource containers to run tasks

Containers

- Can run different types of tasks (also Application Masters)
- Has different sizes e.g. RAM, CPU

YARN Architecture

1. **ResourceManager (RM)**:
    1. **Function**: The ResourceManager is the master that oversees the allocation of computing resources in the cluster and also manages the ApplicationMasters.
    2. **Tasks**:
        1. Tracks live NodeManagers and the cluster’s available resources.
        2. Allocates resources to various running applications.
        3. Monitors the ApplicationMasters which manage the user jobs.
2. **NodeManager (NM)**:
    1. **Function**: A NodeManager is a per-node slave which is responsible for launching the applications’ containers, monitoring their resource usage (CPU, memory, disk, network), and reporting the same to the ResourceManager.
    2. **Tasks**:
        1. Provides computational resources in the form of containers.
        2. Manages the processes running in the containers.

YARN Architecture

3.ApplicationMaster (AM):

1. **Function**: Each application (like a MapReduce job or a Giraph job) has its own instance of an ApplicationMaster.
2. **Tasks**:
    1. Coordinates the execution of all tasks within its application.
    2. Requests the necessary resources from the ResourceManager and works with the NodeManagers to execute and monitor the tasks.

4.Containers:

1. **Description**: Containers are the basic unit of processing capability in YARN, encapsulating resource elements like RAM and CPU.
2. **Function**: They can run various tasks assigned to them, which could be part of different applications.

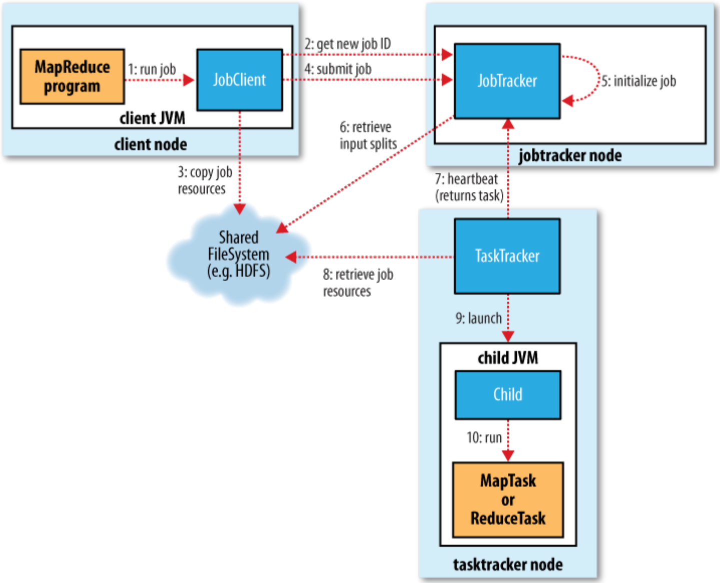
Process Flow

•The client submits an application to the ResourceManager.
•The ResourceManager then finds a NodeManager with available resources and asks it to start the ApplicationMaster for the application.
•The ApplicationMaster registers with the ResourceManager and requests containers.
•The ResourceManager allocates containers from NodeManagers based on the resource requirements.
•The ApplicationMaster communicates with the respective NodeManagers to launch and execute tasks within these containers.

Hadoop vs YARN

Hadoop 1 Deficiency	Impact	YARN Improvement	Result
Limited Scalability	The JobTracker could not efficiently handle a large number of nodes and jobs.	Separate ResourceManager for resource management.	Enhanced scalability to support thousands of nodes and a larger number of jobs.
Inefficient Resource Utilization	Resources were allocated in fixed Map and Reduce slots, leading to potential underutilization.	Dynamic resource allocation based on demand.	Improved resource utilization, matching allocation to actual workload needs.
Poor Cluster Utilization	Static slots for Map and Reduce tasks could remain idle, wasting resources.	Resources are treated as a pool and allocated to any task type.	Better overall cluster utilization, as resources are not wasted.
Lack of Flexibility	Hadoop 1 was tightly coupled with MapReduce, limiting the types of jobs that could run.	Framework-agnostic resource management allows for multiple processing engines.	Ability to run various types of workloads, not limited to MapReduce.
Single Point of Failure	The JobTracker was a critical component whose failure could disrupt all jobs.	High Availability configurations for ResourceManager and decentralized job management.	Increased robustness; a ResourceManager failure does not terminate running tasks.
Overloaded JobTracker	The JobTracker was responsible for too many tasks, leading to potential stability issues.	Splitting responsibilities between ResourceManager and ApplicationMaster.	Improved stability and robustness of the cluster operations.

How Hadoop 1 runs jobs

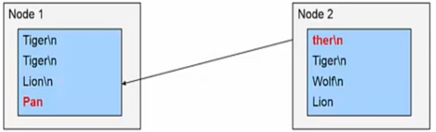
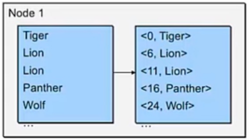


How hadoop 1 runs jobs

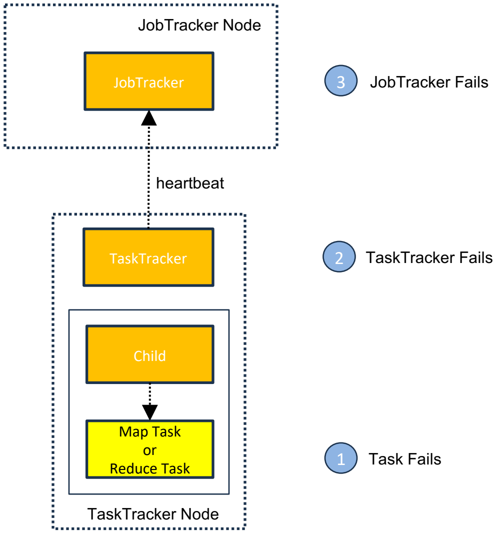
1. **Run Job**: The client submits a MapReduce job to the cluster.
2. **Get new job ID**: The JobClient requests a new job ID from the JobTracker.
3. **Copy job resources**: The job resources are copied to the shared filesystem (e.g., HDFS).
4. **Submit job**: The JobClient submits the job to the JobTracker with new job ID.
5. **Initialize job**: The JobTracker initializes the job and allocates resources.
6. **Retrieve input splits**: The JobTracker determines the input splits for the job.
7. **Heartbeat (returns task)**: The TaskTracker on a node checks in with the JobTracker, which assigns it a task.
8. **Retrieve job resources**: The TaskTracker retrieves the job resources from the shared filesystem.
9. **Launch**: The TaskTracker launches a new Java Virtual Machine (JVM) for the task.
10. **Run**: The new child JVM runs either a MapTask or a ReduceTask.

Classes

The three classes shown work together to enable Hadoop to process large datasets in a distributed manner.

Component	Job	
InputSplitter	Divide data into input splits (blocks)	
RecordReader	Read and parse data within input splits and extract records, and corrects any split errors	 Node 1 Tiger\n Tiger\n Lion\n Pan Node 2 ther\n Tiger\n Wolf\n Lion
InputFormat	Takes each record and converts to <key, value> pairs	 Node 1 Tiger Lion Lion Panther Wolf ... <0, Tiger> <6, Lion> <11, Lion> <16, Panther> <24, Wolf> ...

Fault Tolerance

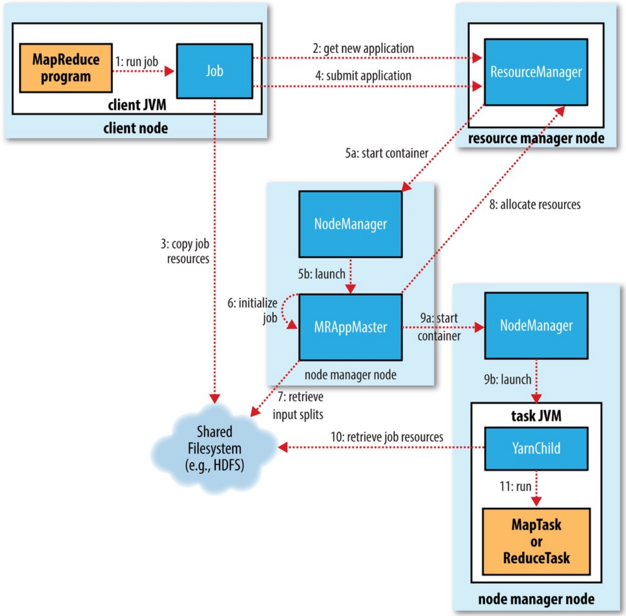


Fault Tolerance

If task tracker fails to communicate with the job tracker:

Task Type	**Re-Execution Condition**	**Reason**
Map Tasks	All map tasks (in progress or completed) will be re-assigned to other TaskTrackers and re-executed	The outputs of map tasks are stored locally on the TaskTracker that executed them. If the TaskTracker fails, these outputs might not have been copied to the machines where the reduce tasks will run, hence they need to be re-executed.
Reduce Tasks	All in-progress reduce tasks will be re-assigned to other TaskTrackers and re-executed	Reduce tasks in progress need to be re-executed because their intermediate state is lost with the TaskTracker. Once a reduce task is completed, its output is written to HDFS and is thus not lost on TaskTracker failure.

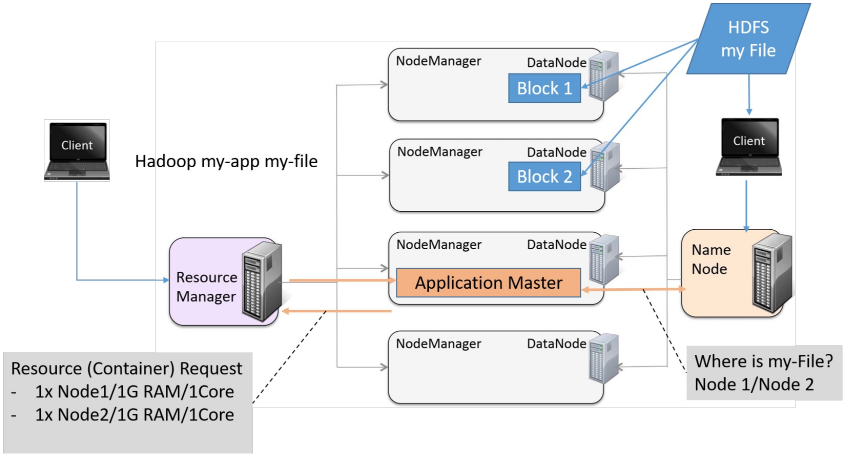
How yarn runs jobs



How yarn runs jobs

1.Run job: The client submits a job to the YARN cluster.
2.Get new application: The client node asks YARN for a new application ID.
3.Copy job resources: The job resources are copied to the shared filesystem (e.g., HDFS).
4.Submit application: The client submits the application (with the job) to the ResourceManager.
5a.Start container: The ResourceManager starts a new container on a NodeManager.
5b.Launch: The NodeManager launches the MRAppMaster.
6.Initialize job: The MRAppMaster initializes the job configuration and resources.
7.Retrieve input splits: The MRAppMaster retrieves the input splits for the job.
8.Allocate resources: The ResourceManager allocates resources for the job.
9a.Start container: The NodeManager starts a new container for a task.
9b.Launch: The NodeManager launches a task JVM (YarnChild).
10.Retrieve job resources: The task JVM retrieves the job resources from the shared filesystem.
11.Run: The task JVM runs either a MapTask or a ReduceTask.

YARN Handling a New Job



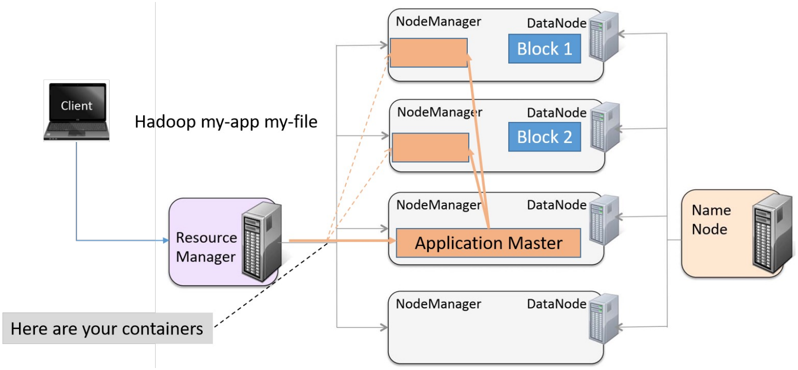
YARN Handling a New Job

1.A client submits a Hadoop job (Hadoop my-app my-file) to the cluster.
2.The ResourceManager receives the job submission and is responsible for allocating resources to it.
3.The client stores his input file into my-file HDFS (Hadoop Distributed File System)., ends up stored in different data nodes, data blocks (Block 1 and Block 2).
4.The ApplicationMaster is initiated within a container managed by a NodeManager
5.ApplicationMaster requests the resources to run the application from the ResourceManager

YARN Handling a New Job

6.ResourceManager processes the resource (container) request for the job, which includes allocations like "1x Node1/1G RAM/1 Core" and "1x Node2/1G RAM/1 Core".
7.ApplicationMaster requests my-file locations from NameNode
8.NameNode responds with the locations of the data blocks.

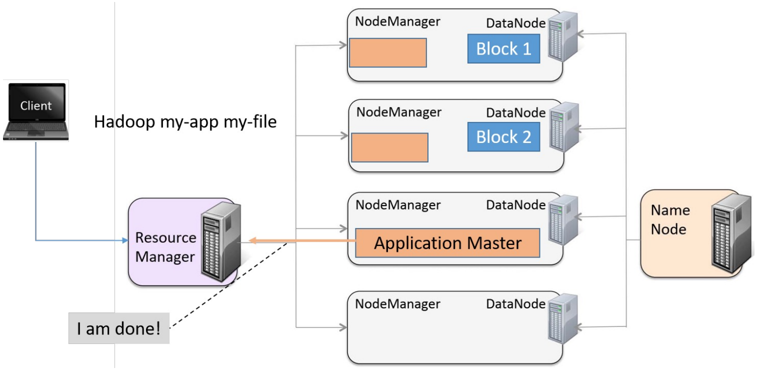
YARN Handling a New Job



YARN Handling a New Job

1.ApplicationMaster requests to initiate containers from the ResourceManager
2.The ResourceManager allocates containers for the job on NodeManagers as per the resource request.
3.Applications are initiated within a container managed by a NodeManager.
4.The ApplicationMaster coordinates the execution of the job, communicating with NodeManagers to start processing the data blocks (Block 1 and Block 2).

YARN Handling a New Job



YARN Handling a New Job

1. The ApplicationMaster continues to manage the job, overseeing the processing of data by the NodeManagers.
2. Once the job is complete, the ApplicationMaster sends a completion notification to the ResourceManager.
3. The ResourceManager acknowledges the completion of the job, and resources are released.
4. The client is notified that the job has finished, and can now retrieve the output from the specified HDFS output location

Resource Management in YARN

- Normally, a cluster runs several jobs
- Such jobs compete for cluster resources: CPU, memory, disk, network
- How would YARN decide about resource allocation?
  - Maximize utilization
  - Data locality
- What if resource are consumed by long running jobs?
  - Different scheduling policies

YARN Scheduling Policies

F I F O

S c h e d u l e r

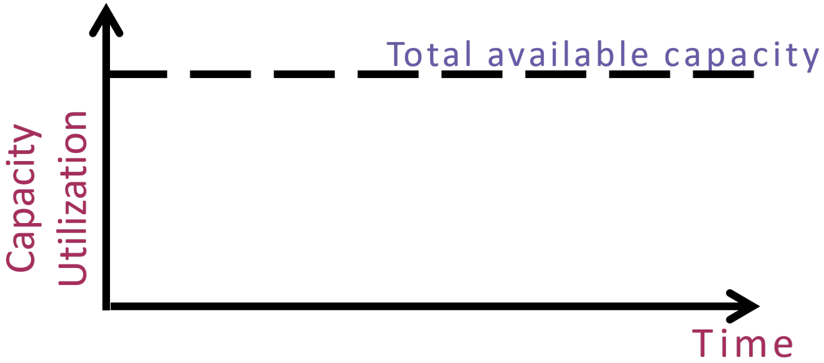
Capacity

Scheduler

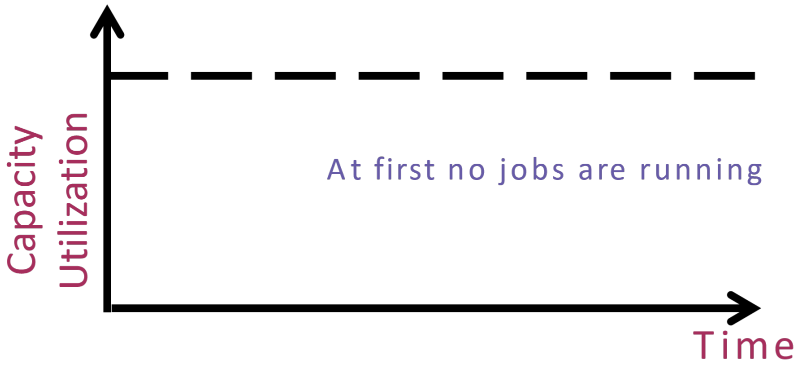
Fair

Scheduler

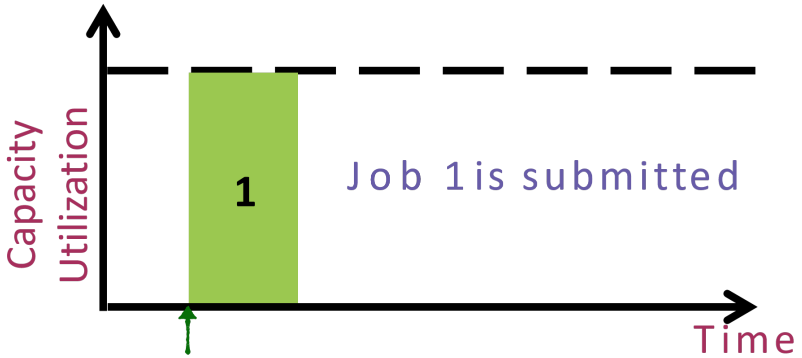
FIFO Scheduler



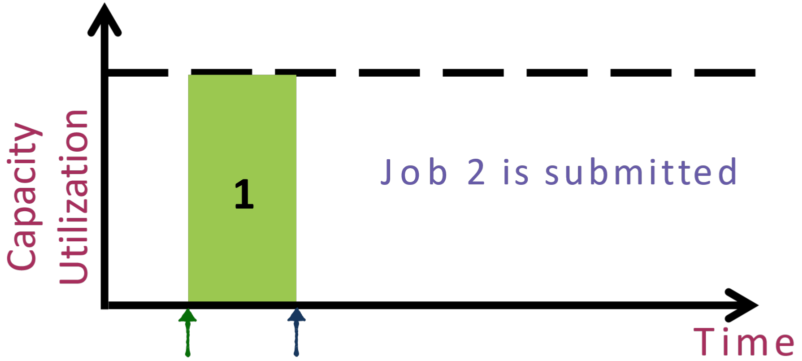
FIFO Scheduler



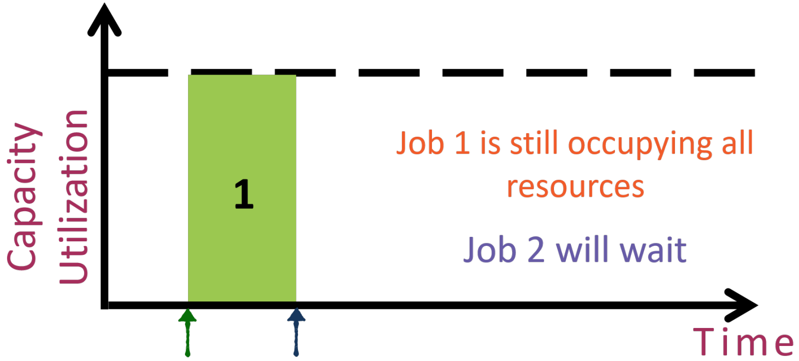
FIFO Scheduler



FIFO Scheduler



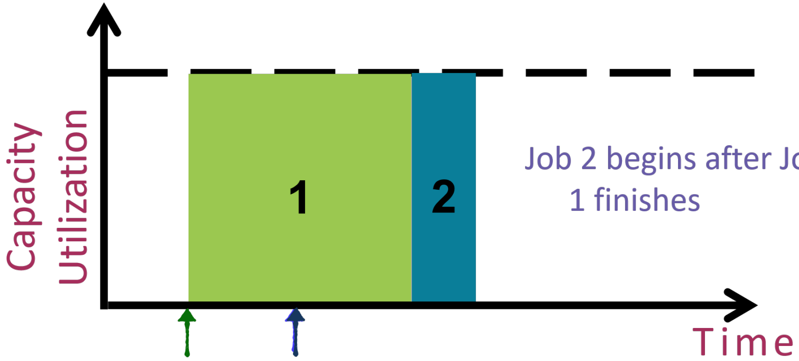
FIFO Scheduler



Job 1 is still occupying all resources

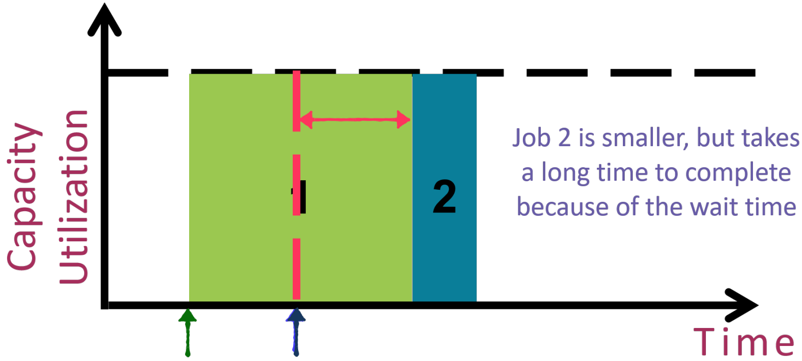
Job 2 will wait

FIFO Scheduler



Job 2 begins after Job 1 finishes

FIFO Scheduler



Job 2 is smaller, but takes a long time to complete because of the wait time

FIFO Scheduler

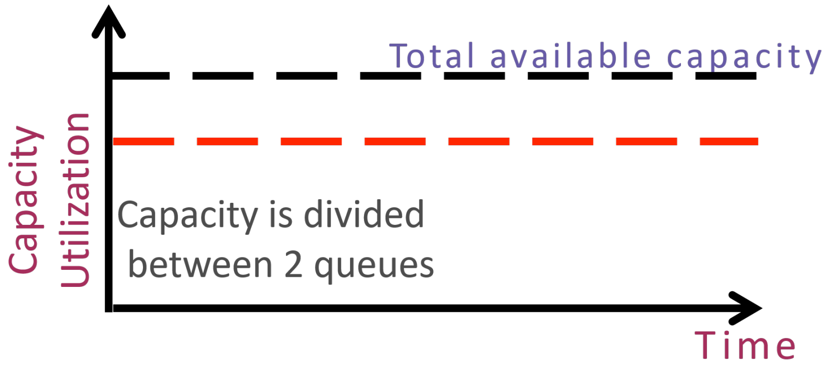
FIFO scheduler is rarely used

Huge wait times!

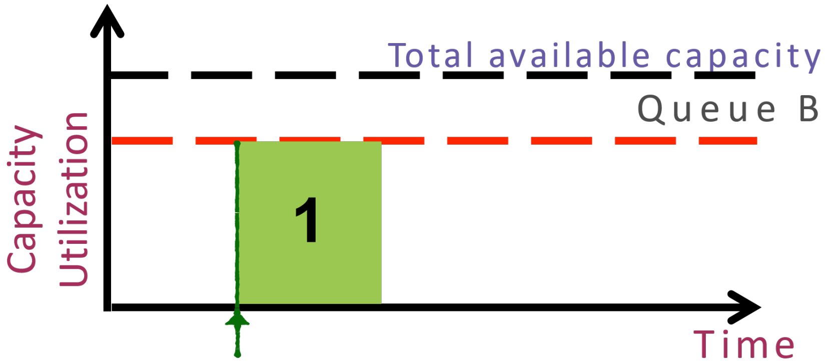
Capacity Scheduler

- Contains multiple queues
- Each queue contains multiple jobs
- Each queue guaranteed some portion of the cluster capacity, e.g.,
    - Queue 1 is given 80
    - Queue 2 is given 20
    - Higher-priority jobs go to Queue 1
    - Queues can be hierarchical
- For jobs within the same queue, FIFO typically used
- Administrators can configure queues
    - Hard versus soft limits

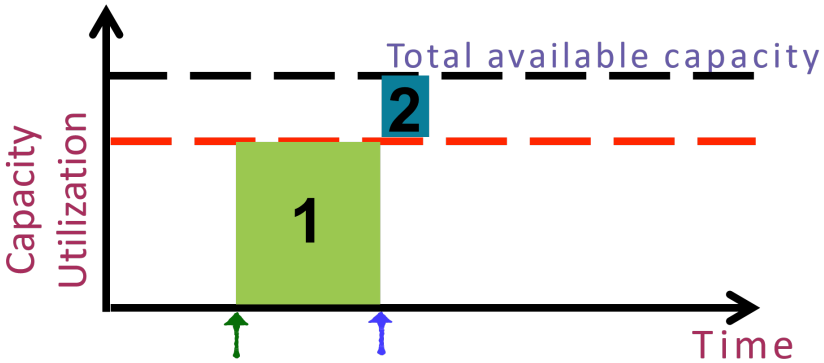
Capacity Scheduler



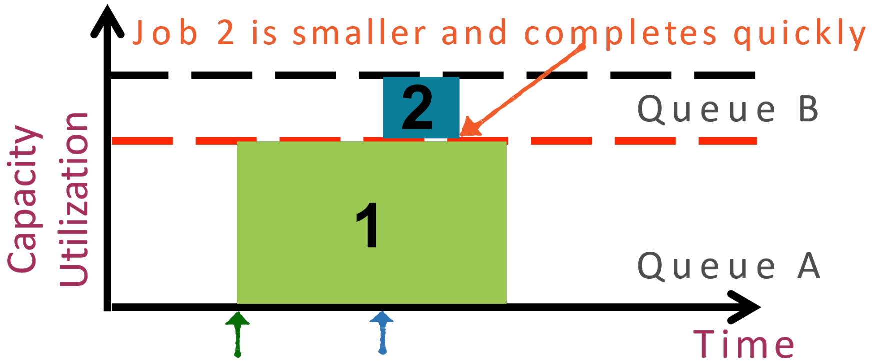
Capacity Scheduler



Capacity Scheduler



Capacity Scheduler



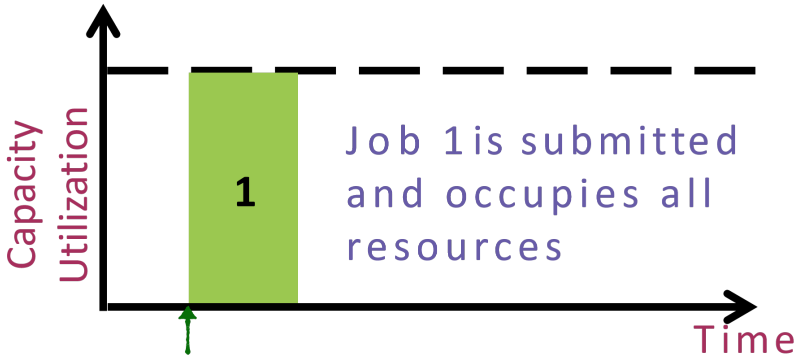
Capacity Scheduler

- Pros
  - Small jobs can finish faster without being stuck (no starvation)
  - Queues can exceed their limits if resources are available
- Cons
  - No preemption
  - Reconfiguring resource

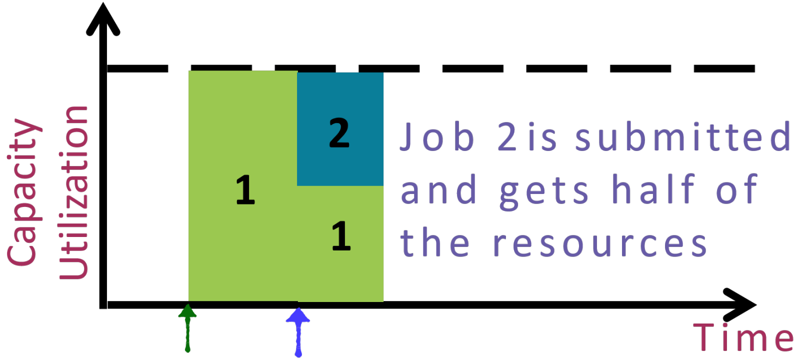
Fair Scheduler

- Goal: all jobs get equal share of resources
- When only one job present, occupies entire cluster
- As other jobs arrive, each job given equal
  - E.g., Each job might be given equal number of cluster-wide YARN containers
  - Each container = = 1 task of job
- Try to achieve zero wait time for any job

Fair Scheduler

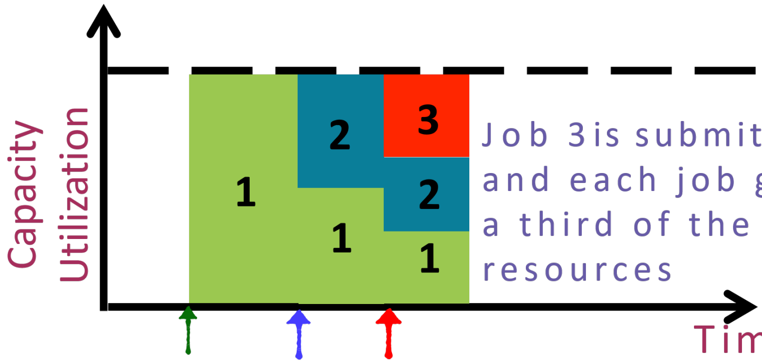


Fair Scheduler



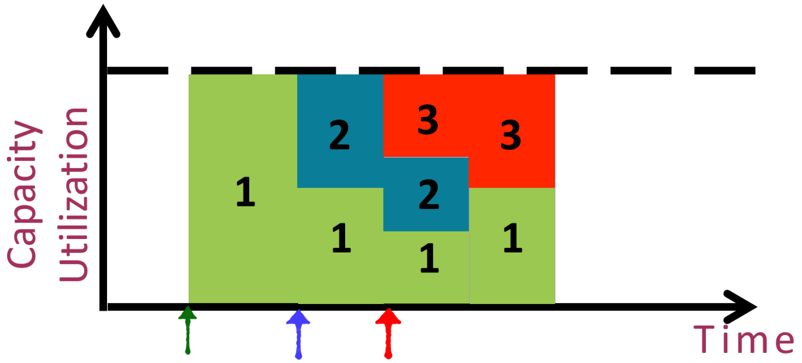
Job 2 is submitted and gets half of the resources

Fair Scheduler



Job 3is submitted and each job gets a third of the resources

Fair Scheduler



YARN Facts

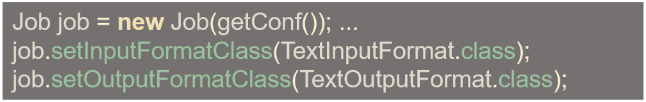
- Being called MapReduce2 is misleading: It does not change the MapReduce programming model,
- It can be used to manage resources for other frameworks, e.g., Spark, Flink, etc. as we will see in future lectures,
- Simply, such new frameworks, implement the Application Master/Manager differently,
- This means, we can have several frameworks installed and running on the same cluster where resources are managed solely by YARN

Limitations of MapReduce

- Usability limitations
  - No support for joins
  - No support for iterative processing No data
  - or process sharing
  - No utilization of indices
- Processing limitations
  - Intermediate steps are not persisted to HDFS
  - Efficient/Compact data storage formats
  - No persistence for intermediate results (Spark)

Processing Limitations: Storage Format

- HDFS like any other file system allows files of different format, up to the application.
- For analytics jobs, commonly input data are in textual formats, CSV, TSV, etc.
- Binary formats: AVRO, Parquet
- File formats affect the job runtime
  - Finding relevant data in the file
  - Time taken to read and write the data (compression)



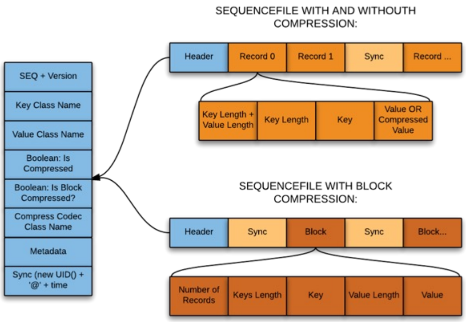
```
Job job = new Job(getConf()); ...
job.setInputFormatClass(TextInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);
```

Textual Files:

- Pros
  - Simple format,
  - Can be read by humans, if you need some debugging
  - Can encode most of analytics input data
  - Splittable
- Cons
  - Waste of space,
  - Sequential access

Sequence Files

- Default for intermediate results in a MapReduce job
- Pros
  - One of the natively supported file formats in HDFS
  - Allow for random access
  - Binary format, smaller than textual in size
  - Block-level compression
- Cons
  - Not backward compatible if you change the schema of your Writable class
  - Encodes the data as (key, value) pairs. You need extra logic in your mapper or record reader if you have a structure for value



SEQUENCEFILE WITH AND WITHOUTH COMPRESSION:
Header
Record 0
Record 1
Sync
Record ...
Key Length + Value Length
Key Length
Key
Value OR Compressed Value
SEQ + Version
Key Class Name
Value Class Name
Boolean: Is Compressed
Boolean: Is Block Compressed?
Compress Codec Class Name
Metadata
Sync (new UID() + '@' + time
SEQUENCEFILE WITH BLOCK COMPRESSION:
Header
Sync
Block
Sync
Block...
Number of Records
Keys Length
Key
Value Length
Value

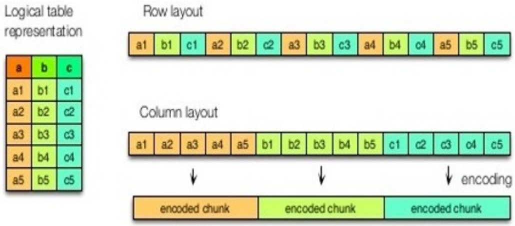
Apache AVRO

- You encode the data according to a schema provided as a JSON object
- Auto schema-based serialization/deserialization
- Allows schema evolution
- Allows compression
- Binary or textual format

```
{
"type": "record",
"name": "debs12",
"fields":
[
  {
    "name": "date",
    "type": "long",
    "format_as_time" : "unix_long"
  },
  {
    "name": "index",
    "type": "long"
  }
]
}
```

Apache Parquet

- Columnar storage
- Pros
  - Vertical partitioning in addition to horizontal
  - Useful if application needs access to subset of the columns, recall Hbase
  - Large community support
- Cons
  - Not suitable if you need to access the entire row


Logical table representation
a
b
c
a1
b1
c1
a2
b2
c2
a3
b3
c3
a4
b4
c4
a5
b5
c5
Row layout
a1 b1 c1 a2 b2 c2 a3 b3 c3 a4 b4 c4 a5 b5 c5
Column layout
a1 a2 a3 a4 a5 b1 b2 b3 b4 b5 c1 c2 c3 c4 c5
encoding
encoded chunk
encoded chunk
encoded chunk