SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Accredited "A" Grade by NAAC | 12B Status by UGC | Approved by AICTE

www.sathyabama.ac.in

SCHOOLOFCOMPUTING

DEPARTMENTOFCOMPUTERSCIENCEANDENGINEERING

UNITV Big Data Analytics – SCSA1603

UNIT 5 MAP REDUCE **9 Hrs**

Algorithms using map reduce - Matrix-Vector – Multiplication – Word Count - Understanding inputs and outputs of MapReduce, Data Serialization – Introduction to YARN – MapReduceVs YARN – YARN Architecture – Scheduling in YARN- – Fair Scheduler – Capacity Scheduler.

Map Reduce

MapReduce is a framework that is used by Hadoop to process the data residing with HDFS. HDFS store data in each block which size is 64 MB or 128 MB. And MapReduce can interaction with HDFS and operates the data in HDFS.The configuration which we set when we set the environment of hadoop.The below diagram will show the structure of hadoop.

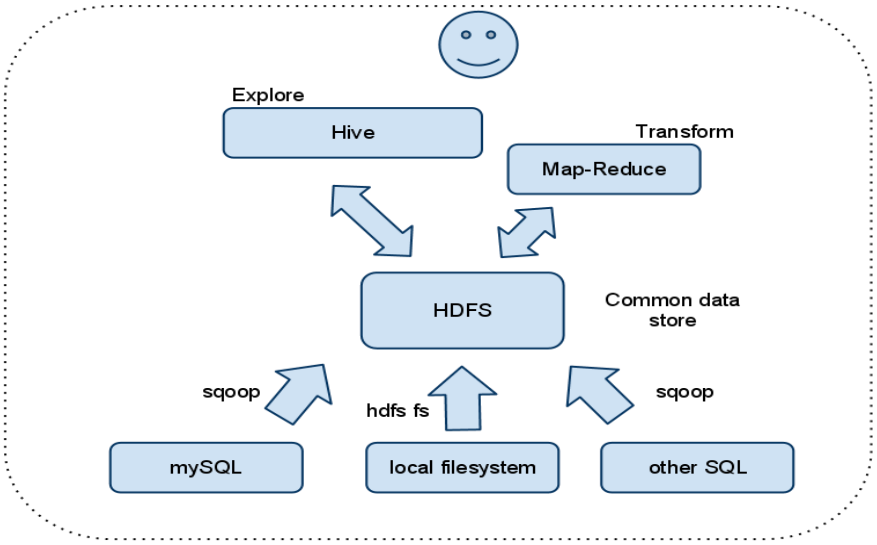


Figure 51.: Structure of Hadoop

MapTask(HDFS data localization):

The unit of input for a map task is an HDFS data block of the input file. The map task functions most efficientlyif the data block it has to process is available locally on the node on which the task is scheduled. This approachiscalled HDFS data localization.

An HDFS data locality miss occurs if the data needed by the map task is not available locally. In such a case, the map task will request the data from another node in the cluster: an operation that is expensive and timeconsuming,leadingto inefficiencies and, hence, delayin job completion.

[

Clients,DataNodes,andHDFSStorage:

Inputdataisuploaded totheHDFSfilesystemineitheroffollowingtwoways:

AHDFSclienthas alarge amountofdatatoplaceintoHDFS.AnHDFS clientis constantlystreamingdatainto HDFS.

Boththesescenarioshavethesameinteraction withHDFS,except thatinthe streamingcase,the clientwaitsforenough data to fill a data block before writing to HDFS. Data is stored in HDFS in large blocks, generally 64 to128MBor morein size.This storageapproach allows easyparallel processingofdata.

```
HDFS-SITE.XML
<property>
<name>dfs.block.size</name>
<value>134217728</value>ç128MB Block size
</property>
```

OR

```
<property>
<name>dfs.block.size</name>
<value>67108864</value>ç64MBBlocksize(Defaultis thisvalueisnot set)
</property>
```

BlockReplicationFactor:

During the process of writing to HDFS, the blocks are generally replicated to multiple data nodes forredundancy.Thenumber ofcopies, orthereplicationfactor, isset toadefaultof 3and canbemodifiedbytheclusteradministrator as below:

```
HDFS-SITE.XML
<property>
```

```
<name>dfs.replication</name>
<value>3</value>
</property>
<property>
```

Whenthereplicationfactor isthree,HDFS'splacement policyisto:

- Putone replicaon onenodeinthelocalrack,
- Anotheron anodeinadifferent (remote) rack,
- Lastonadifferent nodeinthesameremoterack.

Whenanew datablock isstored onadatanode, thedata nodeinitiates areplication processtoreplicate thedataonto a second data node. The second data node, in turn, replicates the block to a third data node, completing thereplicationofthe block.

With this policy, the replicas of a file do not evenly distribute across the racks. One third of replicas are on onenode, two thirds of replicas are on one rack, and the other third are evenly distributed across the remaining racks.Thispolicyimproves write performance without compromisingdata reliabilityorread performance.

5.1 Algorithms using Map Reduce

The MapReduce algorithm contains two important tasks, namely Map and Reduce.

- The map task is done by means of Mapper Class
- The reduce task is done by means of Reducer Class.

Mapper class takes the input, tokenizes it, maps and sorts it. The output of Mapper class is used as input by Reducer class, which in turn searches matching pairs and reduces them.

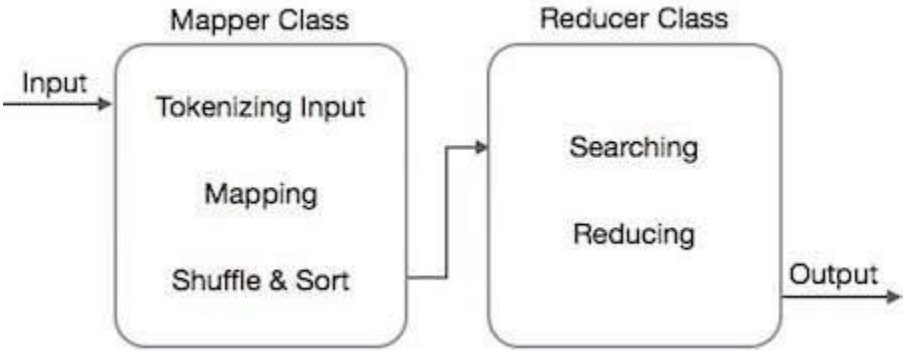


Figure 5.2: The Mapper class and the Reducer class

MapReduce implements various mathematical algorithms to divide a task into small parts and assign them to multiple systems. In technical terms, MapReduce algorithm helps in sending the Map & Reduce tasks to appropriate servers in a cluster.

These mathematical algorithms may include the following −

- Sorting
- Searching
- Indexing
- TF-IDF

Sorting

Sorting is one of the basic MapReduce algorithms to process and analyze data. MapReduce implements sorting algorithm to automatically sort the output key-value pairs from the mapper by their keys.

- Sorting methods are implemented in the mapper class itself.
- In the Shuffle and Sort phase, after tokenizing the values in the mapper class, the **Context** class (user-defined class) collects the matching valued keys as a collection.
- To collect similar key-value pairs (intermediate keys), the Mapper class takes the help of **RawComparator** class to sort the key-value pairs.
- The set of intermediate key-value pairs for a given Reducer is automatically sorted by Hadoop to form key-values (K2, {V2, V2, …}) before they are presented to the Reducer.

Searching

Searching plays an important role in MapReduce algorithm. It helps in the combiner phase (optional) and in the Reducer phase. Let us try to understand how Searching works with the help of an example.

Example

The following example shows how MapReduce employs Searching algorithm to find out the details of the employee who draws the highest salary in a given employee dataset.

- Let us assume we have employee data in four different files − A, B, C, and D. Let us also assume there are duplicate employee records in all four files because of importing the employee data from all database tables repeatedly. See the following illustration.

- **The Map phase** processes each input file and provides the employee data in key-value pairs (<k, v> : <emp name, salary>). See the following illustration.

- **The combiner phase** (searching technique) will accept the input from the Map phase as a key-value pair with employee name and salary. Using searching technique, the combiner will check all the employee salary to find the highest salaried employee in each file. See the following snippet.

```
<k: employee name, v: salary>
Max= the salary of an first employee. Treated as max salary

if(v(second employee).salary > Max){
  Max = v(salary);
}

else{
  Continue checking;
}
```

The expected result is as follows −

<satish, 26000>	<gopal, 50000>	<kiran, 45000>	<manisha, 45000>

- **Reducer phase** − Form each file, you will find the highest salaried employee. To avoid redundancy, check all the <k, v> pairs and eliminate duplicate entries, if any. The same algorithm is used in between the four <k, v> pairs, which are coming from four input files. The final output should be as follows −

<gopal, 50000>

Indexing

Normally indexing is used to point to a particular data and its address. It performs batch indexing on the input files for a particular Mapper.

The indexing technique that is normally used in MapReduce is known as **inverted index.** Search engines like Google and Bing use inverted indexing technique. Let us try to understand how Indexing works with the help of a simple example.

Example

The following text is the input for inverted indexing. Here T[0], T[1], and t[2] are the file names and their content are in double quotes.

T[0] = "it is what it is"
T[1] = "what is it"
T[2] = "it is a banana"

After applying the Indexing algorithm, we get the following output −

"a": {2}
"banana": {2}
"is": {0, 1, 2}
"it": {0, 1, 2}
"what": {0, 1}

Here "a": {2} implies the term "a" appears in the T[2] file. Similarly, "is": {0, 1, 2} implies the term "is" appears in the files T[0], T[1], and T[2].

TF-IDF

TF-IDF is a text processing algorithm which is short for Term Frequency − Inverse Document Frequency. It is one of the common web analysis algorithms. Here, the term 'frequency' refers to the number of times a term appears in a document.

Term Frequency (TF)

It measures how frequently a particular term occurs in a document. It is calculated by the number of times a word appears in a document divided by the total number of words in that document.

TF(the) = (Number of times term the ‘the’ appears in a document) / (Total number of terms in the document)

Inverse Document Frequency (IDF)

It measures the importance of a term. It is calculated by the number of documents in the text database divided by the number of documents where a specific term appears.

While computing TF, all the terms are considered equally important. That means, TF counts the term frequency for normal words like “is”, “a”, “what”, etc. Thus we need to know the frequent terms while scaling up the rare ones, by computing the following −

IDF(the) = log_e(Total number of documents / Number of documents with term ‘the’ in it).

The algorithm is explained below with the help of a small example.

Example

Consider a document containing 1000 words, wherein the word **hive** appears 50 times. The TF for **hive** is then (50 / 1000) = 0.05.

Now, assume we have 10 million documents and the word **hive** appears in 1000 of these. Then, the IDF is calculated as log(10,000,000 / 1,000) = 4.

The TF-IDF weight is the product of these quantities − 0.05 × 4 = 0.20.

5.2 Performing Matrix Vector Multiplication using MapReduce

Suppose we have an n×nn×n matrix M, whose element in row i and column j will be denoted mij . Suppose we also have a vector v of length n, whose jth element is vj . Then the matrix-vector product is the vector x of length n, whose ith element xi is given by xi=∑nj=1mij×vjxi=∑j=1nmij×vj. If n = 100, we do not want to use a DFS or MapReduce for this calculation. But this sort of calculation is at the heart of the ranking of Web pages that goes on at search engines, and there, n is in the tens of billions.3 Let us first assume that n is large, but not so large that vector v cannot fit in main memory and thus be available to every Map task. The matrix M and the vector v each will be stored in a file of the DFS. We assume that the row-column coordinates of each matrix element will be discoverable, either from its position in the file, or because it is stored with explicit coordinates, as a triple (i, j,mij). We also assume the position of element vj in the vector v will be discoverable in the analogous way.
The Map Function: The Map function is written to apply to one element of M. However, if v is not already read into main memory at the compute node executing a Map task, then v is first read, in its entirety, and subsequently will be available to all applications of the Map function performed at this Map task. Each Map task will operate on a chunk of the matrix M. From each matrix element mij it produces the key-value pair (i,mij* vj). Thus, all terms of the sum that make up the component xi of the matrix-vector product will get the same key, i.

The Reduce Function: The Reduce function simply sums all the values associated with a given key i. The result will be a pair (i, xi).

we can divide the matrix into vertical stripes of equal width and divide the vector into an equal number of horizontal stripes, of the same height. Our goal is to use enough stripes so that the portion of the vector in one stripe can fit conveniently into main memory at a compute node. Figure suggests what the partition looks like if the matrix and vector are each divided into five stripes.

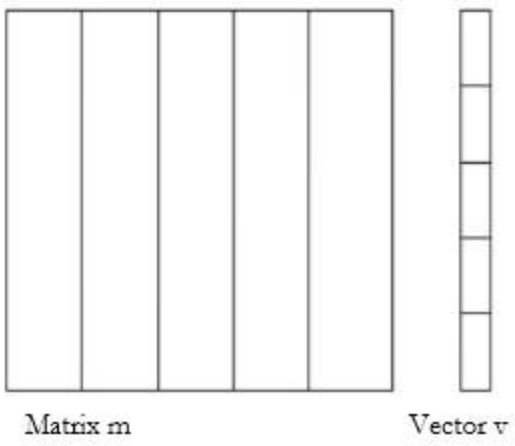


Figure 5.3: Matrix vector representation

Map : for input mij
Emit (i, ps = ∑ mij * vj)

Reduce : Compute Xi = ∑ ps

5.3 Matrix Multiplication With MapReduce

MapReduce is a technique in which a huge program is subdivided into small tasks and run parallelly to make computation faster, save time, and mostly used in distributed systems. It has 2 important parts:

- **Mapper:** It takes raw data input and organizes into key, value pairs. For example, In a dictionary, you search for the word "Data" and its associated meaning is "facts and statistics collected together for reference or analysis". Here the Key is *Data* and the **Value** associated with is *facts and statistics collected together for reference or analysis.*
- **Reducer:** It is responsible for processing data in parallel and produce final output.

Let us consider the matrix multiplication example to visualize MapReduce. Consider the following matrix:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}$$

Figure 5.4: 2×2 matrices A and B

Here matrix A is a 2×2 matrix which means the number of rows(i)=2 and the number of columns(j)=2. Matrix B is also a 2×2 matrix where number of rows(j)=2 and number of columns(k)=2. Each cell of the matrix is labelled as Aij and Bij. Ex. element 3 in matrix A is

called A21 i.e. 2nd-row 1st column. Now One step matrix multiplication has 1 mapper and 1 reducer. The Formula is:

Mapper for Matrix A (k, v)=((i, k), (A, j, Aij)) for all k
Mapper for Matrix B (k, v)=((i, k), (B, j, Bjk)) for all i

Therefore computing the mapper for Matrix A:

k, i, j computes the number of times it occurs.# Here all are 2, therefore when k=1, i can have # 2 values 1 & 2, each case can have 2 further# values of j=1 and j=2. Substituting all values # in formula

k=1 i=1 j=1 ((1, 1), (A, 1, 1))
j=2 ((1, 1), (A, 2, 2))
i=2 j=1 ((2, 1), (A, 1, 3))
j=2 ((2, 1), (A, 2, 4))

k=2 i=1 j=1 ((1, 2), (A, 1, 1))
j=2 ((1, 2), (A, 2, 2))
i=2 j=1 ((2, 2), (A, 1, 3))
j=2 ((2, 2), (A, 2, 4))

Computing the mapper for Matrix B

i=1 j=1 k=1 ((1, 1), (B, 1, 5))
k=2 ((1, 2), (B, 1, 6))
j=2 k=1 ((1, 1), (B, 2, 7))
j=2 ((1, 2), (B, 2, 8))

i=2 j=1 k=1 ((2, 1), (B, 1, 5))
k=2 ((2, 2), (B, 1, 6))
j=2 k=1 ((2, 1), (B, 2, 7))
k=2 ((2, 2), (B, 2, 8))

The formula for Reducer is:

Reducer(k, v)=(i, k)=>Make sorted Alist and Blist
(i, k) => Summation (Aij * Bjk)) for j
Output =>((i, k), sum)

Therefore computing the reducer:

We can observe from Mapper computation # that 4 pairs are common (1, 1), (1, 2),# (2, 1) and (2, 2)# Make a list separate for Matrix A & # B with adjoining values taken from # Mapper step above:

(1, 1) =>Alist={(A, 1, 1), (A, 2, 2)}

Blist={(B, 1, 5), (B, 2, 7)}

Now Aij x Bjk: [(1*5) + (2*7)] =19 -------(i)

(1, 2) =>Alist={(A, 1, 1), (A, 2, 2)}

Blist={(B, 1, 6), (B, 2, 8)}

Now Aij x Bjk: [(1*6) + (2*8)] =22 -------(ii)

(2, 1) =>Alist={(A, 1, 3), (A, 2, 4)}

Blist={(B, 1, 5), (B, 2, 7)}

Now Aij x Bjk: [(3*5) + (4*7)] =43 -------(iii)

(2, 2) =>Alist={(A, 1, 3), (A, 2, 4)}

Blist={(B, 1, 6), (B, 2, 8)}

Now Aij x Bjk: [(3*6) + (4*8)] =50 -------(iv)

From (i), (ii), (iii) and (iv) we conclude that

((1, 1), 19)

((1, 2), 22)

((2, 1), 43)

((2, 2), 50)

Therefore the Final Matrix is:

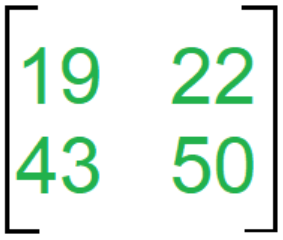


Figure 5.5.: Final output of Matrix multiplication.

5.4 Word Count Problem using Map reduce and Understanding inputs and outputs of MapReduce

5.4.1 Understanding the inputs and outputs

Input:

What do you mean by Object
What do you know about Java
What is Java Virtual Machine How Java enabled High Performance

Output:

What	3
Do	2
You	2
Mean	1
By	1
Object	1
Know	1
about	1
Java	3
Is	1
Virtual	1
Machine	1
How	
1	
enabled	1
High	1
Performance	1

5.4.2 Word Count Problem:

The word count problem undergoes two phases, a mapper phase and a reducer phase

Word count Mapper:

```
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;
public class WC_Mapper extends MapReduceBase implements Mapper<LongWritable,Text,Te
xt,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);
      }
   }

}
```

Word count Reducer

```
   import java.io.IOException;
   import java.util.Iterator;
   import org.apache.hadoop.io.IntWritable;
   import org.apache.hadoop.io.Text;
   import org.apache.hadoop.mapred.MapReduceBase;
   import org.apache.hadoop.mapred.OutputCollector;
   import org.apache.hadoop.mapred.Reducer;
```

```
import org.apache.hadoop.mapred.Reporter;

public class WC_Reducer  extends MapReduceBase implements Reducer<Text,IntWritable,T
ext,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));
}
}
```

Main method class

```
import java.io.IOException;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.TextInputFormat;
import org.apache.hadoop.mapred.TextOutputFormat;
public class WC_Runner {
   public static void main(String[] args) throws IOException{
      JobConf conf = new JobConf(WC_Runner.class);
      conf.setJobName("WordCount");
      conf.setOutputKeyClass(Text.class);
      conf.setOutputValueClass(IntWritable.class);
      conf.setMapperClass(WC_Mapper.class);
      conf.setCombinerClass(WC_Reducer.class);
```

```
        conf.setReducerClass(WC_Reducer.class);
        conf.setInputFormat(TextInputFormat.class);
        conf.setOutputFormat(TextOutputFormat.class);
        FileInputFormat.setInputPaths(conf,new Path(args[0]));
        FileOutputFormat.setOutputPath(conf,new Path(args[1]));
        JobClient.runJob(conf);
    }
}
```

MapReduce is a processing technique and a program model for distributed computing based on java. The MapReduce algorithm contains two important tasks, namely Map and Reduce.

- **Map** takes a set of data and converts it into another set of data, where individual elements are broken down into tuples (key/value pairs).
- **Reduce** task takes the output from a map as an input and combines those data tuples into a smaller set of tuples.

As the sequence of the name MapReduce implies, the reduce task is always performed after the map job.

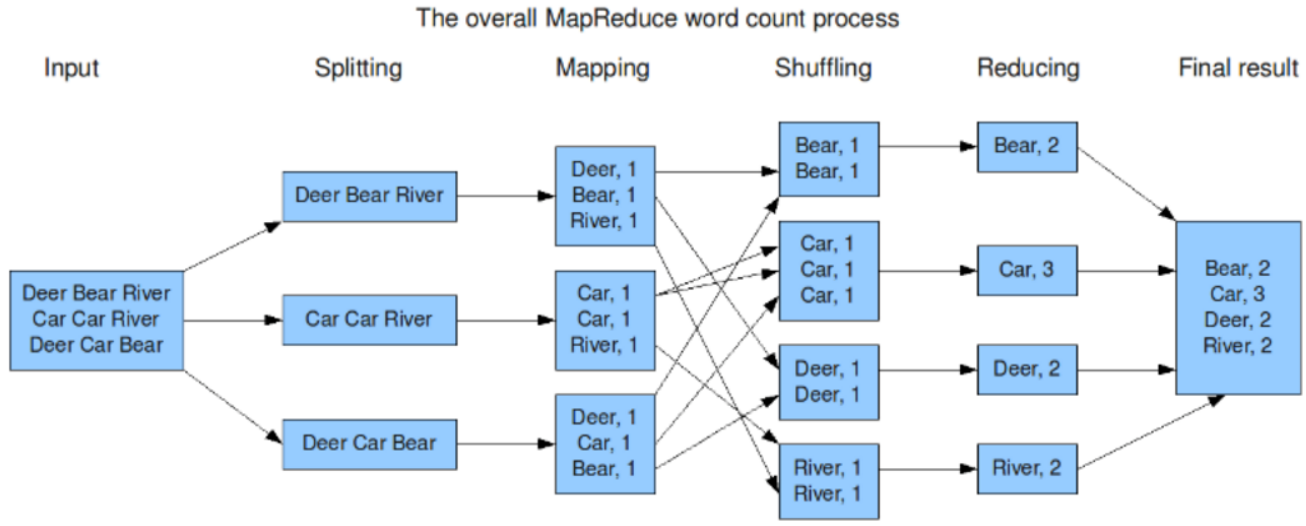


Figure 5.6: The word count problem using Map reduce

5.5 Data Serialization:

- Serialization is the process of converting object data into byte stream data for transmission over a network across different nodes in a cluster or for persistent data storage.

- Data serialization is a process that converts structure data manually back to the original form.
- Serialize to translate data structures into a stream of data. Transmit this stream of data over the network or store it in DB regardless of the system architecture.
- Isn't storing information in binary form or stream of bytes is the right approach.
- Serialization does the same but isn't dependent on architecture.

Consider CSV files contains a comma (,) in between data, so while Deserialization, wrong outputs may occur. Now, if metadata is stored in XML form, a self- architected form of data storage, data can easily deserialize.

Deserialization:

- Deserialization is the reverse process of serialization and converts byte stream data into object data for reading data from HDFS. Hadoop provides Writablesfor serialization and deserialization purpose.
- Writable and WritableComparable Interfaces to provide mechanisms for serialization anddeserialization of data, Hadoop provided two important interfaces Writable and WritableComparable
- **Hadoop File System (HDFS)?**
- Hadoop File System (HDFS) is a distributed file system. Store all types of files in the Hadoop file system. It supports all standard formats such as Gifs, text, CSV, tsv, xls, etc. Beginners of Hadoop can opt for tab_delimiter (data separated by tabs) files because it is -
- Easy to debug and readable
- The default format of Apache Hive
- **What is the Data Serialization Storage format?**
- Storage formats are a way to define how to store information in the file. Most of the time, assume this information is from the extension of the data. Both structured and unstructured data can store on HADOOP-enabled systems. Common Hdfs file formats are -
- Plain text storage
- Sequence files
- RC files
- AVRO
- Parquet

Need for storage formats in Hadoop

- File format must be handy to serve complex data structures

- HDFS enabled applications to find relevant data in a particular location and write back data to another location.
- Dataset is large
- Having schemas
- Having storage constraints
-

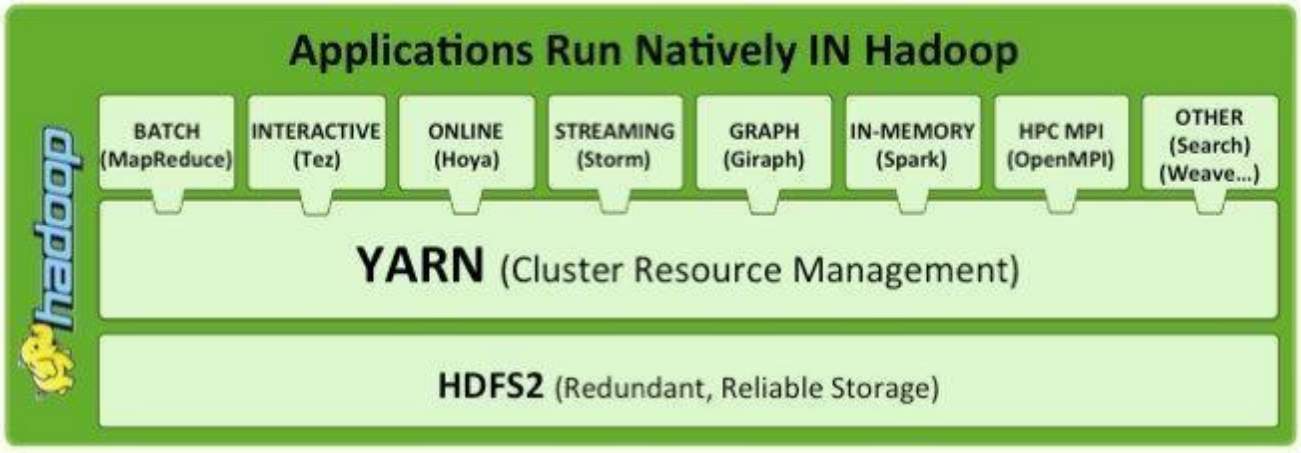
Serialization in Hadoop has two areas -

- **Interprocess communication**
- When a client calls a function or subroutine from one pc to the pc in-network or server, that calling is a remote procedure call.
-
- **Persistent storage**
- It is better than java's inbuilt serialization as java serialization isn't compact. Serialization and Deserialization of data help maintain and manage corporate decisions for effective use of resources and data available in Data warehouse or any other database -writable - language specific to java.

5.6 Introduction to YARN and Hadoop2.0

Apache Hadoop 2.0 represents a generational shift in the architecture of Apache Hadoop.With YARN, Apache Hadoop is recast as a significantly more powerful platform – one that takesHadoop beyond merely batch applications to taking its position as a _data operating system‘where HDFS is the file system and YARN is the operating system.YARN is a re-architecture ofHadoop that allows multiple applications to run on the same platform. With YARN, applicationsrun—in‖Hadoop,insteadof—on‖Hadoop:

Figure 5.7: Hadoop2.0 YARN



ThefundamentalideaofYARNistosplitupthetwomajorresponsibilitiesoftheJobTracker and TaskTracker into separate entities. In Hadoop 2.0, the JobTracker and TaskTracker no longerexistandhavebeenreplacedbythreecomponents:

- **ResourceManager:**ascheduler thatallocatesavailableresourcesinthecluster amongstthecompetingapplications.
- **NodeManager:** runs on each node in the cluster and takes direction from theResourceManager.Itisresponsibleformanagingresourcesavailableonasing lenode.
- **ApplicationMaster:** an instance of a framework-specific library, an ApplicationMaster runsa specific YARN job and is responsible for negotiating resources from theResourceManagerandalsoworkingwiththe NodeManagertoexecute andmonitorContainers.

The actual data processing occurs within the Containers executed by the ApplicationMaster.A Container grants rights to an application to use a specific amount of resources (memory,cpuetc.)onaspecifichost.

YARN is not the only new major feature of Hadoop 2.0. HDFS has undergone a majortransformationwithacollectionofnewfeaturesthatinclude:

- **NameNode HA:** automated failover with a hot standby and resiliency for

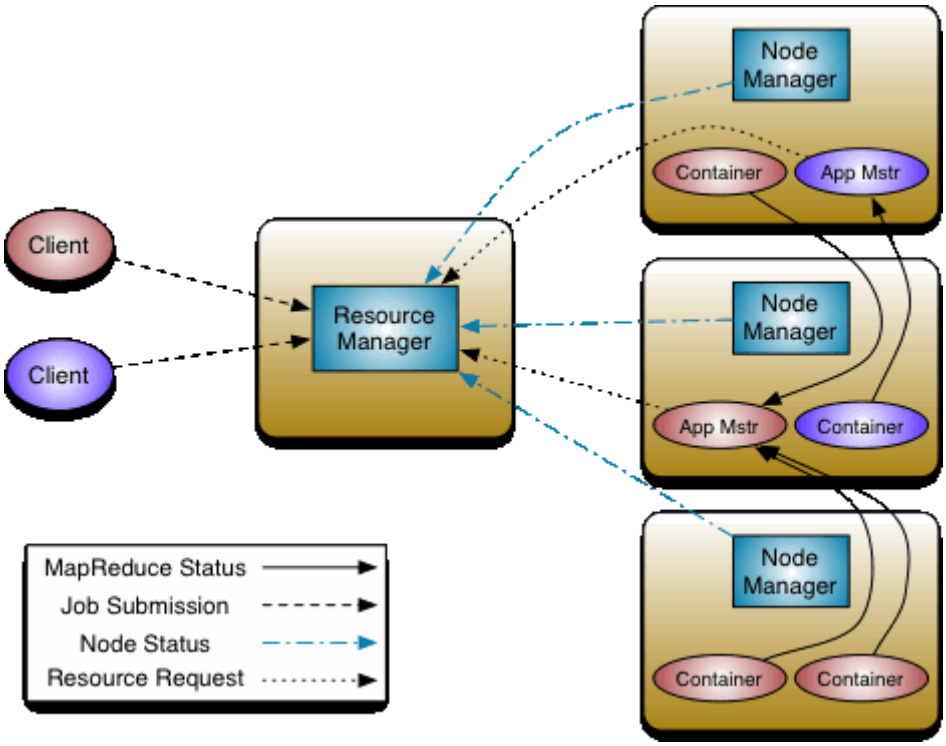
theNameNodemasterservice.

- **Snapshots:** point-in-timerecoveryforbackup, disasterrecoveryandprotectionagainstuseerrors.
- **Federation:** a clear separation of namespace and storage by enabling generic blockstoragelayer.

MRv2-YARN

Thenewarchitectureintroducedinhadoop-0.23,dividesthetwomajorfunctionsoftheJobTracker: resource management and job life-cycle management into separate components.Thenew ResourceManager manages the global assignment of compute resources to applications andtheper-applicationApplicationMaster managestheapplication‘sschedulingand coordination.

An application is either a single job in the sense of classic MapReduce jobs or a DAG of suchjobs.TheResourceManager and per-machine NodeManager daemon, which manages the



userprocessesonthatmachine,formthecomputationfabric.Theper-applicationApplicationMasteris, in effect, a framework specific library and is tasked with negotiating resources from theResourceManager and working with the NodeManager(s) to execute and monitor the tasks. Thefundamental idea of YARN is to split up the functionalities of resource management and jobscheduling/monitoring into separate daemons. The idea is to have a global ResourceManager(*RM*)and per-applicationApplicationMaster(*AM*).An applicationiseithera singlejob oraDAGofjobs.TheResourceManagerandtheNodeManagerformthedata-computationframework.TheResourceManageristheultimateauthoritythatarbitratesresourcesamongallthe applications in the system.TheNodeManager is the per-machine framework agent who isresponsibleforcontainers,monitoringtheirresourceusage(cpu,memory,disk,network)andreporting the same to the ResourceManager/Scheduler.The per-application

ApplicationMaster is,ineffect,aframeworkspecificlibraryandistaskedwithnegotiatingresourcesfromthe ResourceManagerand working withtheNodeManager(s)toexecuteandmonitorthetasks.

Figure 5.8YARN workflow

TheResourceManagerhastwomaincomponents:SchedulerandApplicationsManager. The Scheduler is responsible for allocating resources to the various running applications subjectto familiar constraints of capacities, queues etc. The Scheduler is pure scheduler in the sense thatit performs no monitoring or tracking of status for the application. Also, it offers no guaranteesabout restarting failed tasks either due to application failure or hardware failures.The Schedulerperforms its scheduling function based the resource requirements of the applications; it does sobasedontheabstractnotionofa resource*Container*whichincorporateselements suchasmemory,cpu,disk,networketc.

NameNodeHighAvailablity

Prior to Hadoop 2.0.0, the NameNode was a single point of failure (SPOF) in an HDFS cluster.EachclusterhadasingleNameNode,andifthatmachineorprocessbecameunava ilable,the cluster as a whole would be unavailable until the NameNode was either restarted or brought uponaseparatemachine.

Thisimpacted thetotal availabilityoftheHDFSclusterin twomajorways:

- In the case of an unplanned event such as a machine crash, the cluster would beunavailable untilanoperatorrestartedtheNameNode.
- Plannedmaintenanceeventssuch assoftwareorhardwareupgradeson theNameNodemachine wouldresultinwindowsofclusterdowntime.

The HDFS High Availability feature addresses the above problems by providing

the option ofrunning two redundant NameNodes in the same cluster in an Active/Passive configuration with ahot standby.This allows a fast failover to a new NameNode in the case that a machine crashes, ora gracefuladministrator-initiated failoverforthe purposeofplannedmaintenance.

5.7 YARN Architecture

In a typical HA cluster, two separate machines are configured as NameNodes. At anypoint in time, exactly one of the NameNodes is in an *Active* state, and the other is in a *Standby*state. The Active NameNode is responsible for all client operations in the cluster, while theStandbyissimply actingas aslave, maintainingenoughstateto provide a fastfailover ifnecessary.In order for the Standby node to keep its state synchronized with the Active node, thecurrent implementation requires that the two nodes both have access to a directory on a sharedstorage device (eg an NFS mount from a NAS). This restriction will likely be relaxed in futureversions.When any namespace modification is performed by the Active node, it durably logs arecord of the modification to an edit log file stored in the shared directory. The Standby node isconstantly watching this directory for edits, and as it sees the edits, it applies them to its ownnamespace.In the event of a failover, the Standby will ensure that it has read all of the edits fromthe shared storage before promoting itself to the Active state. This ensures that the namespacestate is fullysynchronizedbeforeafailoveroccurs.

In order to provide a fast failover, it is also necessary that the Standby node have up-to-date information regarding the location of blocks in the cluster. In order to achieve this, theDataNodesareconfiguredwiththelocationofbothNameNodes,andsendblocklocation informationandheartbeats toboth.

Hardwareresources

In ordertodeploy anHAcluster,youshouldprepare thefollowing:

- **NameNodemachines-**

  themachinesonwhichyouruntheActiveandStandbyNameNodesshouldhaveequivalenthardwaretoeachother,andequivalenthardware towhatwouldbeusedinanon-HAcluster.

- **Shared storage** - you will need to have a shared directory which both NameNode machinescan have read/write access to. Typically this is a remote filer whichsupportsNFS and ismountedoneachoftheNameNodemachines.Currentlyonlyasinglesharededitsdirectory is supported.Thus, the availability of the system is limited by the availability ofthis shared edits directory, and therefore in order to remove all single points of failurethere needs toberedundancyforthesharededitsdirectory.

HDFSFederation

HDFSFederationimprovestheexisting**HDFS**architecturethroughaclearseparationofnamespaceandstorage,enablinggenericblockstoragelayer.Itenables support for multiple namespaces in the cluster to improve scalability andisolation

This guide provides an overview of the HDFS Federation feature and how toconfigure andmanagethefederatedcluster.

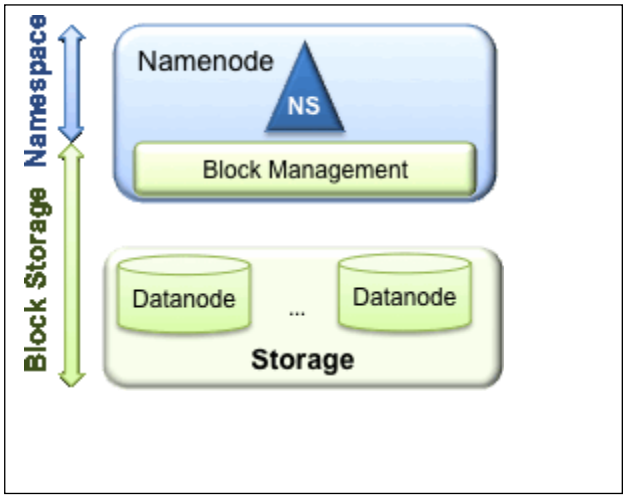


Figure 5.9 Hadoop Storage Representation

HDFS hastwomainlayers:

Namespace

- Consistsof directories,filesandblocks.

It supportsallthenamespacerelated filesystem operationssuch ascreate,delete,modifyandlistfilesanddirectories.

BlockStorageService,which hastwoparts:

- BlockManagement (performedintheNamenode)

Provides Datanode cluster membership by handling registrations,andperiodic heartbeats.

Processesblockreportsandmaintainslocationofblocks.

Supportsblockrelated operationssuch ascreate,delete,modifyandgetblocklocation.

Managesreplicaplacement,blockreplicationforunder replicatedblocks,anddeletesblocksthatareoverreplicated.

Storage - is provided by Datanodes by storing blocks on the local filesystemandallowingread/writeaccess.

The prior HDFS architecture allows only a single namespace for the entire cluster. In thatconfiguration, a single Namenode manages the namespace. HDFS Federation addresses thislimitationbyaddingsupportformultipleNamenodes/namespacestoHDFS.

KeyBenefits

- NamespaceScalability-Federationaddsnamespacehorizontalscaling.Largedeployments or deployments using lot of small files benefit from namespace scaling byallowingmoreNamenodestobeaddedtothecluster.
- Performance - File system throughput is not limited by a single Namenode. Adding moreNamenodestotheclusterscalesthe filesystemread/writethroughput.
- Isolation-

AsingleNamenodeoffersnoisolationinamultiuserenvironment.Forexample,anexperimentalapplicationcanoverloadtheNamenodeandslowdownproduction critical applications. By using

multiple Namenodes, different categories ofapplicationsanduserscanbe isolatedtodifferentnamespaces.

Federationconfigurationis**backwardcompatible**andallowsexistingsingleNamenod econfigurationstoworkwithoutanychange.Thenewconfigurationisdesigned such that all the nodes in the cluster have the same configuration without theneed fordeployingdifferent configurationsbasedonthe typeofthenode inthe cluster.

Configuration:

Step1:Addthedfs.nameservicesparameter to your configurationand configureitwithalist ofcommaseparatedNameServiceIDs.Thiswillbeusedby the Datanodestodetermine theNamenodesinthecluster.

Step 2: For each Namenode and Secondary Namenode/BackupNode/Checkpointer add thefollowingconfigurationparameterssuffixedwiththecorrespondingNameServiceIDinto thecommonconfigurationfile:

FormattingNamenodes

Step 1:Formata Namenodeusingthefollowingcommand:

[hdfs]$$HADOOP_PREFIX/bin/hdfsnamenode-format[-clusterId<cluster_id>]

Chooseauniquecluster_id which willnotconflictotherclustersin your environment.Ifacluster_idis notprovided, thenauniqueone is autogenerated.

Step 2:FormatadditionalNamenodesusing thefollowing command:

[hdfs]$$HADOOP_PREFIX/bin/hdfsnamenode-format-clusterId<cluster_id>

Upgradingfromanolderreleaseandconfiguringfederation

Older releasesonlysupportasingleNamenode.Upgradethecluster

tonewerreleaseinordertoenable federationDuringupgrade youcanprovide aClusterID asfollows:

[hdfs]$ $HADOOP_PREFIX/bin/hdfs start namenode --config $HADOOP_CONF_DIR -upgrade-clusterId<cluster_ID>

Ifcluster_idisnot provided,itisauto generated.

Adding anew NamenodetoanexistingHDFScluster Performthefollowingsteps:

Adddfs.nameservicestotheconfiguration.
Updatetheconfigurationwith theNameServiceIDsuffix.Configurationkeynameschanged post release 0.20. You must use the new configuration parameter namesinordertousefederation.
AddthenewNamenoderelatedconfigtotheconfigurationfile.
Propagate the configurationfiletothe allthe nodesin thecluster.
Start thenewNamenodeandSecondary/Backup.
Refresh the Datanodes to pickup the newly added Namenode by runningthefollowingcommandagainstallthe Datanodesinthecluster:

[hdfs]$$HADOOP_PREFIX/bin/hdfsdfsadmin-refreshNameNodes <datanode_host_name>:<datanode_rpc_port>

Managing a cluster

Tostarttheclusterrunthefollowingcommand:

[hdfs]$ $HADOOP_PREFIX/sbin/start-dfs.sh To stop the cluster run thefollowingcommand:

[hdfs]$$HADOOP_PREFIX/sbin/stop-dfs.sh

Balancer

TheBalancerhasbeen changed toworkwithmultipleNamenodes.TheBalancer can berunusingthecommand:

[hdfs]$$HADOOP_PREFIX/sbin/hadoop-daemon.shstartbalancer[-policy<policy>]

Decommissioning

Decommissioning is similar to prior releases. The nodes that need to be decomissioned areadded totheexcludefile atall ofthe Namenodes.EachNamenodedecommissionsitsBlock Pool.When all the Namenodes finish decommissioning a Datanode, the Datanode is considereddecommissioned.

Step 1:Todistributeanexclude filetoalltheNamenodes,use thefollowingcommand:[hdfs]$$HADOOP_PREFIX/sbin/distribute-exclude.sh<exclude_file>

Step 2: Refresh all the Namenodes to pick up the new exclude file:[hdfs]$ $HADOOP_PREFIX/sbin/refresh-namenodes.sh

Theabovecommand usesHDFS configurationtodeterminetheconfigured Namenodesinthe clusterandrefreshesthemtopickupthenewexcludefile.

ClusterWebConsole

Similar to the Namenode status web page, when using federation a Cluster Web Console isavailabletomonitorthefederatedcluster athttp://<any_nn_host:port>/dfsclusterhealth.jsp.AnyNamenodeinthe clustercanbeusedtoaccessthiswebpage.

MigratingfromMapReduce1(MRv1)to MapReduce2

MapReduce from Hadoop 1 (MapReduce MRv1) has been split into two components. Theclusterresourcemanagementcapabilitieshavebecome YARN (YetAnotherResourceNegotiator), while the MapReduce-specific capabilities remain MapReduce. In the MapReduce MRv1 architecture, the cluster was managed by a service called the JobTracker. TaskTrackerserviceslivedoneachhostandwouldlaunchtasksonbehalfofjobs. TheJobTrackerwouldserve informationaboutcompletedjobs.

InMapReduceMRv2,thefunctionsoftheJobTrackerhavebeensplitbetweenthreeservices. The ResourceManager is a persistent YARN service that receives and runs applications (aMapReducejobisanapplication)onthecluster. Itcontainsthescheduler,which,aspreviously,ispluggable. TheMapReduce-specificcapabilities oftheJobTrackerhavebeenmovedintotheMapReduceApplicationMaster,oneofwhichisstartedtomanageeachMapReducejobandterminatedwhenthejobcompletes. TheJobTrackerfunctionofservinginformationaboutcompletedjobshasbeenmovedtotheJobHistoryServer. TheTaskTracker

hasbeenreplacedwiththeNodeManager,aYARNservicethatmanagesresourcesanddeployment on a host. It is responsible for launching containers, each of which can house a mapor reduce task.Nearly all jobs written for MRv1 will be able to run without any modifications onanMRv2cluster.

The new architecture has its advantages. First, by breaking up the JobTracker into a few differentservices,itavoidsmanyofthescalingissuesfacedbyMapReduceinHadoop1.Moreimportantly, it makes it possible to run frameworks other than MapReduce on a Hadoop cluster.Forexample,Impala canalsorunonYARN andshare resourceswithMapReduce.

ConfigurationMigration

Since MapReduce 1 functionality has been split into two components, MapReduce clusterconfigurationoptions havebeensplit intoYARNconfiguration options,whichgo inyarn-site.xml, and MapReduce configuration options, which go in mapred-site.xml. Many have beengiven new names to reflect the shift. As JobTrackers and TaskTrackers no longer exist in MRv2,all configuration options pertaining to them no longer exist, although many have correspondingoptionsfortheResourceManager,NodeManager,andJobHistoryServer.

A minimal configuration required to run MRv2jobson YARNis:yarn-site.xmlconfiguration

```
<?xmlversion="1.0"encoding="UTF-8"?>
<configuration>
<property>
<name>yarn.resourcemanager.hostname</name>
<value>you.hostname.com</value>
</property>

<property>
<name>yarn.nodemanager.aux-services</name>
<value>mapreduce_shuffle</value>
</property>
</configuration>
```

mapred-site.xmlconfiguration

```
<?xmlversion="1.0"encoding="UTF-8"?>
<configuration>
<property>
<name>mapreduce.framework.name</name>
<value>yarn</value>
</property>
</configuration>
```

Below is a table with HA-related configurations used in MRv1 and their equivalents inYARN:

MRv1	**YARN/ MRv2**	**Comment**
mapred.jobtrackers.*name*	yarn.resourcemanager.ha.rm-ids	
mapred.ha.jobtracker.id	yarn.resourcemanager.ha.id	Unlike in MRv1, thismust be configured inYARN.
mapred.jobtracker.*rpc-address.name.id*	yarn.resourcemanager.*rpc-address.id*	YARN/ MRv2hasdifferent RPC ports fordifferentfunctionalities.Eachport-relatedconfigurationmustbesuffixed withanid. Note thatthereis no <name>in YARN.
mapred.ha.jobtracker.*rpc-address.name.id*	yarn.resourcemanager.ha.admin.address	
mapred.ha.fencing.methods	yarn.resourcemanager.ha.fencer	Not required to bespecified
mapred.client.failover.*	None	Notrequired
	yarn.resourcemanager.ha.enabled	Enable HA
mapred.jobtracker.restart.recover	yarn.resourcemanager.recovery.enabled	Enable recovery of jobsafterfailover
	yarn.resourcemanager.store.class	org.apache .hadoop.yarn
		.server.resourcemanager.recovery .ZKRMStateStore
mapred.ha.automatic-failover.enabled	yarn.resourcemanager.ha.automatic-failover.enabled	Enable automaticfailover
mapred.ha.zkfc.port	yarn.resourcemanager.ha.automatic-failover.port	

mapred.job.tracker	yarn.resourcemanager.cluster.i	

ProgramminginYARNFramework

Under Hadoop 2.0, MapReduce is but one instance of a YARN application, where YARN hastakencenterstageasthe‐operatingsystem‖ofHadoop. BecauseYARNallows*any*applicationto run on equal footing with MapReduce, it opened the floodgates for a new generation of softwareapplicationswiththesekinds offeatures:

More programming models. Because YARN supports any application that can divide itself intoparalleltasks,theyarenolongershoehornedintothepaletteof‐mappers,‖‐combiners,‖and ‐reducers.‖This in turn supports complex data-flow applicationslike ETL and ELT, and iterative programslikemassively-parallelmachinelearningandmodeling.

Integration of native libraries. Because YARN has robust support for *any* executable – notlimited to MapReduce, and not even limited to Java – application vendors with a large maturecode basehaveaclearpathtoHadoopintegration.

Supportforlargereferencedata.YARNautomatically‐localizes‖andcacheslargereference datasets,makingthemavailabletoallnodesfor‐datalocal‖processing. Thissupportslegacy functions like address standardization, which require large reference data sets that cannot beaccessed from the Hadoop Distributed File System (HDFS) by the legacy libraries. Of course,MapReduceisn‘t the only option for processing data at scale using Hadoop. Toolslike Pig (alarge scale query and analysis system), Hive (a data warehousing application) and others havebeen available for some time. These tools can express transforms and analysis using moreaccessible constructs:HiveusesHQL,a languagesimilartoSQL.Pig provides a script language (Pig Latin) to create MapReduce jobs. Business analystsfamiliar with conventional tools like SQL and SAS should be able to use these tools towrite programstosolvelargedataproblems onHadoopclusters

TheValueofVisualApplicationDevelopmentTools

Anewgenerationof‐visualdesign‖applicationdevelopmenttoolscouldhelpsolve thesecoding problems. By running as native YARN applications and side-stepping

theneed for MapReduce, some of these programs eliminate coding altogether.Other toolsreduce coding by generating MapReduce code or by generating scripts like Pig.

Visualdesignersarepowerfulforseveralreasons:

- Increasedlevelofabstraction:Insteadof thinkingaboutclassesandmethods,usersseeoperations,data,andoutcomes.
- Fast—what-if‖:Thedrag-and-connectinterfacesupportsquicktry/observe/adjustcycles.
- Automaticoptimization:Scalingandefficiencyarebuilt-in.
- High-levelpalette:High-levelconstructslike–standardizeaddress‖,–deduplicateconsumers‖, or–parsenames‖areoftendirectlyonthedesignerpalette.

Howdoesthislookinpractice?Here‘sanillustrationthatshowshowthreecompetingapproachesdiffer:

- MapReducewritteninJava
- Pigscriptsdevelopedfromscratch
- A visually-designed process running a native YARN ETL application. The applicationisfrom RedPointGlobal,but comparableapproachescan beseen inTalendandActian.

Usingthesethreeapproaches,weconducteda–WordCount‖teston30,000files(20gigabytes) of Project Gutenberg books. This test reads lines of text, breaks them into words, and creates aconcordance(listofwords andthenumberof times eachoccurs).OurHadoopclusterwassmall—onlyfournodes—butwaslargeenoughtodemonstratethe conceptsandtradeoffs

MapReduce:

Set-up time: While flexible, MapReduce had the longest learning curve and required significantcoding skills—bothasa Java programmeranda MapReducespecialist—toprepare the test.

Performance:Ittook3hours20minutestoruntheteststinitiallyduetothe–smallfilesproblem‖ that is familiar to seasoned MapReduceprogrammers.This problem occurs when reading largecollections of small files, because MapReduce‘s default behavior is to assign a mapper task toeach file. This results in a huge number of tasks.To address this issue, we created a

customInputFormat class to read multiple files at once. This reduced our run time to 58 minutes. Thenwe tuned the split sizes and mapper task limit appropriately, which dropped the run time to aboutsixminutes.

Comments:Eachperformanceimprovementcameatacost.Overall,nearlyafulldayofprogrammertimewasspentoptimizingtheoriginalcode.

Pig:

Set-up time: Learning Pig was fairly easy. It was pretty natural to create the coding forthis test. However to make a common adjustment in the code—changing the set ofwhitespaceseparatorstoincludepunctuation—requiredtheadditionofa-Userdefined function‖orUDFwhichhadtobe written inJava.

Pigisgenerallyeasyenoughtousebypeoplewhoaren‘tprofessionalprogrammersbut who know how to write scripting languages like JavaScript or Visual Basic.Performance:Theresultswerenotstellar:run timewascloseto 15 minutes.

Comments:Whilecodingtooklesstime,Javaprogrammingwasultimatelyrequiredtomeetthetestrequirements.

Sample Pig script without the User Defined Function:

```
SET pig.maxCombinedSplitSize 67108864
SET pig.splitCombination true
A = LOAD '/testdata/pg/*/*/*';
B = FOREACH A GENERATE FLATTEN(TOKENIZE((chararray)$0)) AS word;
C = FOREACH B GENERATE UPPER(word) AS word;
D = GROUP C BY word;
E = FOREACH D GENERATE COUNT(C) AS occurrences, group;
F = ORDER E BY occurrences DESC;
STORE F INTO '/user/cleonardi/pg/pig-count';
```

YARN-enabledETL/ELTdesigner:

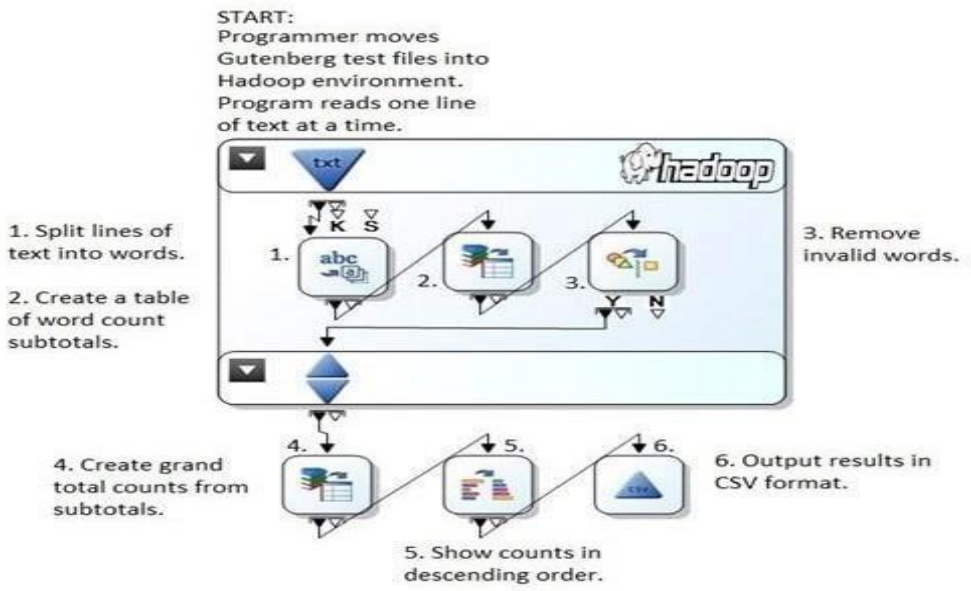
Set-up time: The tool is designed to have a shorter learning curve than even Pig scripting.Dragging tools like ‑Delimited Input|, ‑Summarize|and ‑Tokenize|from the palette and configuring them is designed to be discoverable and intuitive, and the resulting diagram has aone-to-onecorrespondencebetween iconsand operations.There‘sno needforcoding orlearninga languagelikeJavaorPig.

The visual design covers the input file format, tokenizing and counting steps. The resulting dataflow graphcontainsseveniconsalongwith a grouping constructthat showswhat executes‑inside|Hadoop.Eachiconrepresents astepinthe datatransformation.

Performance: The run time for this data flow is just over three minutes with no tuning.*Comments:*Becausethereis nocodetomanage, andeditingis donevisually,running‑whatif|scenariosis quickfornon-programmers.

Once the data flow is designed, it can be stored and saved for later use. In addition, the logic canbecapturedintoa‑macr0|forsharingandreusebetweenmultipledataflows

Figure 5.10: Sharing and reusing between multiple data flows



5.8 Scheduling in YARN

5.8.1 Hadoop Fair Scheduler:

FairScheduler, a pluggable scheduler for Hadoop that allows YARN applications to share resources in large clusters fairly.

Fair scheduling is a method of assigning resources to applications such that all apps get, on average, an equal share of resources over time. HadoopNextGen is capable of scheduling multiple resource types. By default, the Fair Scheduler bases scheduling fairness decisions only on memory. It can be configured to schedule with both memory and CPU, using the notion of Dominant Resource Fairness developed by Ghodsi et al. When there is a single app running, that app uses the entire cluster. When other apps are submitted, resources that free up are assigned to the new apps, so that each app eventually on gets roughly the same amount of resources. Unlike the default Hadoop scheduler, which forms a queue of apps, this lets short apps finish in reasonable time while not starving long-lived apps. It is also a reasonable way to share a cluster between a number of users. Finally, fair sharing can also work with app priorities - the priorities are used as weights to determine the fraction of total resources that each app should get.

The scheduler organizes apps further into "queues", and shares resources fairly between these queues. By default, all users share a single queue, named "default". If an app specifically lists a queue in a container resource request, the request is submitted to that queue. It is also possible to assign queues based on the user name included with the request through configuration. Within each queue, a scheduling policy is used to share resources between the running apps. The default is memory-based fair sharing, but FIFO and multi-resource with Dominant Resource Fairness can also be configured. Queues can be arranged in a hierarchy to divide resources and configured with weights to share the cluster in specific proportions.

In addition to providing fair sharing, the Fair Scheduler allows assigning guaranteed minimum shares to queues, which is useful for ensuring that certain users, groups or production applications always get sufficient resources. When a queue contains apps, it gets at least its minimum share, but when the queue does not need its full guaranteed share, the excess is split between other running apps. This lets the scheduler guarantee capacity for queues while utilizing resources efficiently when these queues don't contain applications.

The Fair Scheduler lets all apps run by default, but it is also possible to limit the number of running apps per user and per queue through the config file. This can be useful when a user must submit hundreds of apps at once, or in general to improve performance if running too many apps at once would cause too much intermediate data to be created or too much context-switching. Limiting the apps does not cause any subsequently submitted apps to fail, only to wait in the scheduler's queue until some of the user's earlier apps finish.

Two things to note about Fair Scheduler in YARN are-

1. By default, the Fair Scheduler bases scheduling fairness decisions only on memory. It can be configured to schedule with both memory and CPU.
2. The scheduler organizes apps further into "queues", and shares resources fairly between these queues.

As example– If there are two queues **sales** and **finance**. A job is submitted to sales queue, being a sole running job it will get all the resources. Now a job is submitted to finance queue which will result in the new job gradually getting half of the resources. So jobs in both of the queues will have 50% of the resources each. Now another job is submitted to finance queue that will result in half of the resources allocated to finance queue allocated to this new job. So two jobs in the finance queue will now share resources allocated to finance queue (50% of the total resources) in equal proportions where as job in sales queue will use whole 50% of the resources allocated to sales queue.

Hierarchical Queues support:
The fair scheduler in YARN supports hierarchical queues which means an organization can create sub-queues with in its dedicated queue.
All queues descend from a queue named "root". Available resources are distributed among the children of the root queue in the typical fair scheduling fashion. Then, the children distribute the resources assigned to them to their children in the same fashion.

Fair Scheduler Configuration
To use the Fair Scheduler in YARN first assign the appropriate scheduler class in **yarn-site.xml**:

```
<property>
<name>yarn.resourcemanager.scheduler.class</name>
<value>org.apache.hadoop.yarn.server.resourcemanager.scheduler.fair.FairScheduler</value>
</property>
```

Setting up queues
Properties for setting up queues are as follows. These changes are done in the configuration file **etc/hadoop/fair-scheduler.xml**.
<queue> element– Represent queues. Some of the important properties of queue element are as following.

- **minResources**: minimum resources the queue is entitled to, in the form "X mb, Y vcores". If a queue's minimum share is not satisfied, it will be offered available resources before any other queue under the same parent.
- **maxResources**: maximum resources a queue is allocated, expressed either in absolute values (X mb, Y vcores) or as a percentage of the cluster resources (X% memory, Y% CPU).
- **weight**: to share the cluster non-proportionally with other queues. Weights default to 1, and a queue with weight 2 should receive approximately twice as many resources as a queue with the default weight.
- **schedulingPolicy**: to set the scheduling policy of any queue. The allowed values are "fifo", "fair", "drf" or any class that extends org.apache.hadoop.yarn.server.resourcemanager.scheduler.fair.SchedulingPolicy. Defaults to "fair".

<defaultQueueSchedulingPolicy> element- Which sets the default scheduling policy for queues; overridden by the schedulingPolicy element in each queue if specified. Defaults to "fair".

<queueMaxAppsDefault> element- Which sets the default running app limit for queues; overridden by maxRunningApps element in each queue.
<queuePlacementPolicy> element– This element contains a list of rule elements that tell the scheduler how to place incoming apps into queues. Rules are applied in the order that they are listed. All rules accept the "create" argument, which indicates whether the rule can create a new queue. "Create" defaults to true; if set to false and the rule would place the app in a queue that is not configured in the allocations file, we continue on to the next rule.
Valid rules are as follows:

- **specified**: The application is placed into the queue it requested.
- **user**: The application is placed into a queue with the name of the user who submitted it.
- **primaryGroup**: The application is placed into a queue with the name of the primary group of the user who submitted it.
- **secondaryGroupExistingQueue**: The application is placed into a queue with a name that matches a secondary group of the user who submitted it.
- **nestedUserQueue**: The application is placed into a queue with the name of the user under the queue suggested by the nested rule.
- **default**: The application is placed into the queue specified in the 'queue' attribute of the default rule. If 'queue' attribute is not specified, the app is placed into 'root.default' queue.
- **reject**: The application is rejected.

Queue configuration example:
If there are two top level child queues **sales** and **finance** (Descending from root). With in sales queues there are two sub-queues **apac** and **emea** then the queues can be set up to use fair scheduler as given below-

```
<allocations>
<queuename="sales">
<minResources>10000 mb,0vcores</minResources>
<maxResources>50000 mb,0vcores</maxResources>
<weight>2.0</weight>
<schedulingPolicy>fifo</schedulingPolicy>
<queuename="emea"/>
<queuename="apac"/>
</queue>
<queuename="finance">
<minResources>10000 mb,0vcores</minResources>
<maxResources>70000 mb,0vcores</maxResources>
<weight>3.0</weight>
<schedulingPolicy>fair</schedulingPolicy>
</queue>
<queuePlacementPolicy>
<rulename="specified"/>
<rulename="primaryGroup"create="false"/>
<rulename="default"queue="finance"/>
</queuePlacementPolicy>
</allocations>
```

5.8.2 Capacity Scheduler

d YARN defines a minimum allocation and a maximum allocation for the resources it is scheduling for: Memory and/or Cores today. Each server running a worker for YARN has a NodeManager that is providing an allocation of resources which could be memory and/or cores that can be used for scheduling. This aggregate of resources from all Node Managers is provided as the 'root' of all resources the capacity scheduler has available.

The fundamental basics of the Capacity Scheduler are around how queues are laid out and resources are allocated to them. Queues are laid out in a hierarchical design with the topmost parent being the 'root' of the cluster queues, from here leaf (child) queues can be assigned from the root, or branches which can have leafs on themselves. Capacity is assigned to these queues as min and max percentages of the parent in the hierarchy. The minimum capacity is the amount of resources the queue should expect to have available to it if everything is running maxed out on the cluster. The maximum capacity is an elastic like capacity that allows queues to make use of resources which are not being used to fill minimum capacity demand in other queues.

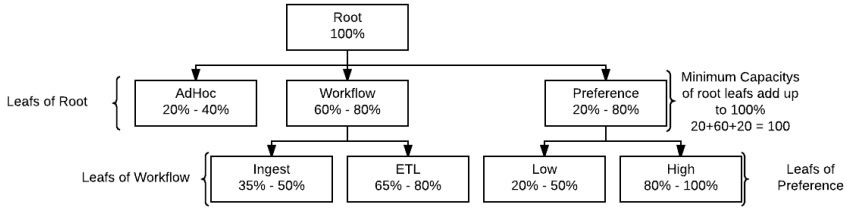


Fig 5.11Capacity Scheduler

Children Queues like in the figure above inherit the resources of their parent queue. For example, with the Preference branch the Low leaf queue gets 20% of the Preference 20% minimum capacity while the High lead gets 80% of the 20% minimum capacity. Minimum Capacity always has to add up to 100% for all the leafs under a parent.

Minimum User Percentage and User Limit Factor

Minimum User Percentage and User Limit Factor are ways to control how resources get assigned to users within the queues they are utilizing. The Min User Percentage is a soft limit on the smallest amount of resources a single user should get access to if they are requesting it. For example a min user percentage of 10% means that 10 users would each be getting 10% assuming they are all asking for it; this value is soft in the sense that if one of the users is asking for less we may place more users in queue.

User Limit Factor is a way to control the max amount of resources that a single user can consume. User Limit Factor is set as a multiple of the queues minimum capacity where a user limit factor of 1 means the user can consume the entire minimum capacity of the queue. If the user limit factor is greater than 1 it's possible for the user to grow into

maximum capacity and if the value is set to less than 1, such as 0.5, a user will only be able to obtain half of the minimum capacity of the queue. Should you want a user to be able to also grow into the maximum capacity of a queue setting the value greater than 1 will allow the minimum capacity to be overtaken by a user that many times.

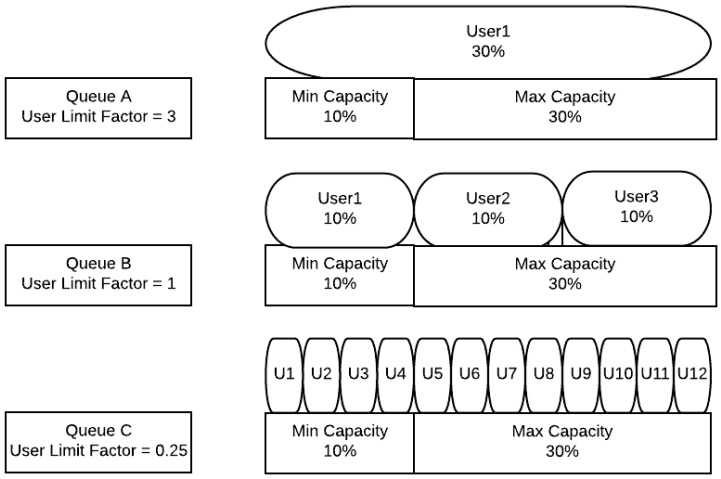


Fig 5.12 Queining in capcityscheduler

A common design point that may initially be non-intuitive is creation of queues by workloads and not by applications and then using the user-limit-factor to prevent individual takeover of queues by a single user by using a value of less than 1.0. This model supports simpler operations by not allowing queue creation to spiral out of control by creating one per LoB but by creating queues by workloads to create predictable queue behaviors. Once a cluster is at full use and has applications waiting in line to run the lower user-limit-factors will be key to controlling resource sharing between tenants.

Initially this may not provide the cluster utilization at the start of their Hadoop platform journey due to using a smaller user limit to limit user resources, there are many approaches but one to consider is that initially it may be justified to allow a single tenant to take over a tiny cluster (say 10 nodes) but when expanding the platform footprint lower the user limit factor so that each tenant keeps the same amount of allocatable resources as before you added new nodes. This lets the user keep his initial experience, but be capped off from taking over the entire cluster as it grows making room for new users easily to get allocated capacity while not degrading the original user's experience.

Archetypes

Designing queue archetypes to describe effective behavior of the tenant in the queue provides a means to measure changes against to see if they align or deviate from the expectation. While by no means a complete list of workload behaviors the below is a good list to start from. Create your own definitions of the types of queues your organization needs as the types of patterns applications and end users are performing.

- **AD-HOC**
  - This is where random user queries, unknown, and new workloads may be ran, there is no expectation as to resource allocation behaviors but can work as a good place to initially run applications to get a feel for per application tuning needs.
- **PREFERENCE**
  - These are applications that should get resources before and retain them longer. This could be for many reasons such as catch up applications, emergency runs, or other operational needs.
- **MACHINE LEARNING**
  - Machine Learning applications can be typically characterized by their long run times and large or intensive resource requirements. Termination of a task for some machine learning workloads can have long running impacts by greatly extending the duration
- **DASHBOARDING**
  - Low concurrency (refresh rate) but high number of queries per day. Dashboards need to be refreshed in an expected time but are very predictable workloads
- **EXPLORATION**
  - Exploration users have a need for low latency queries and need throughput to churn through very large datasets. Resources will likely be held for use the entire time the user is exploring to provide an interactive experience
- **BATCH WORKFLOW**
  - Designed to provide general computing needs for transformation and batch workloads. Setup is concerned most frequently with throughput of applications not individual application latency
- **ALWAYS ON**
  - Applications that are always running with no concept of completion. Applications that keep resources provisioned while waiting for new work to arrive. Slider deployed instances such as LLAP

Container Churn

Churn within a queue is best described as a constant existing and starting of new containers. This behavior is very important to have a well behaving cluster were queues are quickly rebalanced to their minimum capacity and to balance capacity of the queue between its users fairly. The anti-pattern to churn is the long lived container that allocates itself and never releases as it can prevent proper rebalancing of resources in some cases it can completely block other applications from launching or queues from getting their capacity back. When not using preemption if a queue was to grow into elastic space but never release

its containers the elastic space will never be given back to queues that have been guaranteed it. If a application running in your queues has long lived containers make special note and consider ways to mitigate its impact to other users with features like user-limit-factors, preemption, or even dedicated queues without elastic capacity.

Features &Behaviors of the Capacity Scheduler

CPU Scheduling (Dominant Resource Fairness)

CPU Scheduling is not enabled by default and allows for the oversubscription of cores with no enforcement of use or preferred allocation taken into account. Many batch driven workloads may experience a throughput reduction if CPU scheduling is used, but may be required for strict SLA guarantees. A method called Dominant Resource Fairness (DRF) is used to decide what resource type to weight the utilization as: DRF takes the most used resource and treats you as using the highest percent for scheduling.

There are 2 primary parts to CPU Scheduling

1. Allocation and Placement
2. Enforcement

Allocation and placement is solved simply by enabling CPU Scheduling so that the Dominant Resource Fairness algorithm and VCores Node Manager's report begin getting used by the scheduler. This solves for some novel problems like an end user that used to schedule his Spark applications with 1 or 2 executors but 8 cores per, and then all other tasks that ran on these nodes due to the excess of available memory were impacted. By enabling simple CPU Scheduling other tasks would no longer be provisioned onto the servers were all the cores are being utilized and find other prefered locations for task placement in the cluster.

Enforcement is solved by utilizing CGroups. This allows for YARN to ensure that if a task asked for a single vcore that they cannot utilize more than what was requested. Sharing of vcores when not used can be enabled or a strict enforcement of only what was scheduled can be done. The Node Manager's can also be configured as to what the max amount of CPU use on the server they will allow all tasks to sum up to, this allows for cores to be guaranteed to OS functions.

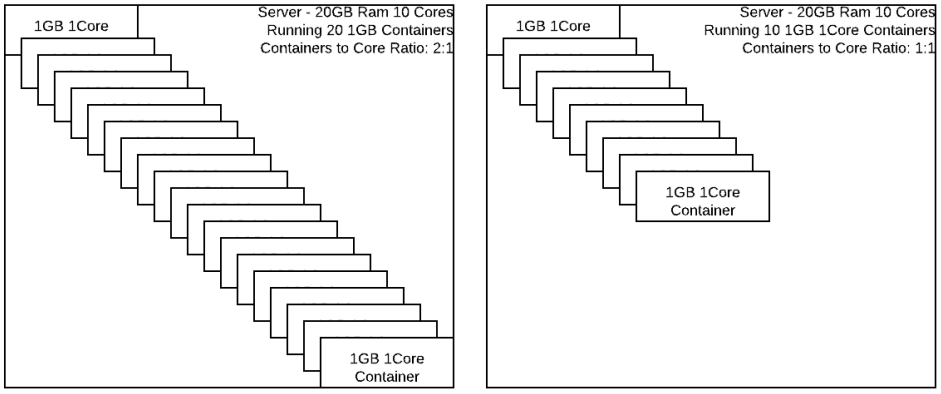


Fig 5.13 Resource allocation to CPU cores

The above figure gives an idea of how many concurrent containers can change if limited to the smallest resource (typically CPU cores.) It's unlikely a true 1:1 Core to Container ratio is what will be required but this aspect of tuning is best left to monitoring historical system metrics and then increasing or decrease the number of NodeManagerVCores available for scheduler to allow more or less containers on a given server group.

Preemption

When applications are making use of elastic capacity in their queues and another application comes along to demand back their minimum capacity (which is being used in another queue as elastic) traditionally the application would have to wait for the task to finish to get its resource allocations. With Preemption enabled resources in other queues can be reclaimed to provide the minimum capacity back to the queue that needs it. Preemption will try not to outright kill a application, and will take reducers last as they have to repeat more work then mappers if they have to re-run. From an ordering perspective Preemption looks at the youngest application and most over-subscribed ones first for task reclamation.

Preemption has some very specific behaviors and some of them don't function as expected for users. The most commonly expected behavior is for the queues to preempt within themselves to balance the resources over all users. That assumption is wrong as preemption only works across queues today, resources that are mis-balanced within a queue between users needs to look into other ways to control this such as User Limit Factors, Improved Queue Churned, and Queue FIFO/FAIR Policies.

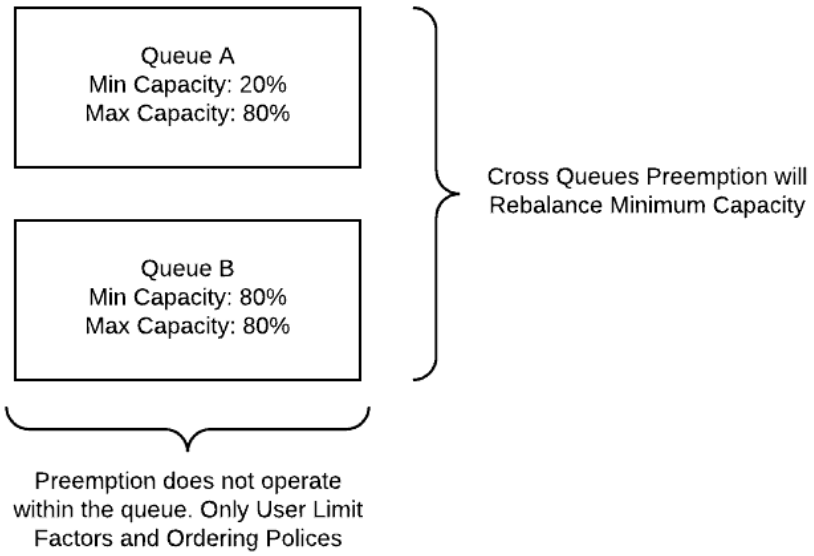


Figure 5.14: FIFO fair policy

Another behavior is that Preemption will not preempt resources if it cannot provide enough to fulfill another resource allocation request. While typically this is not an issue with large cluster for small clusters with large maximum container sizes could run into a scenario were Preemption is not configured to reclaim a container of the largest possible size and therefore will do nothing at all. The main properties used to control this behavior is the Total PreemptionPer Round and the Natural Termination Factor. Total PreemptionPer Round is the percentage of resources on the cluster that can be preempted at once, the Natural Termination Factor is the percent of resources out of the total cluster (100%) requested that will be preempted up to the Total Preemption Per Round.

The last misconception is that Preemption will rebalance max (elastic) capacity use for between queues that wish to grow into it. Maximum Capacity is given on a first come first serve basis only. If a single queue has taken over all the of the clusters capacity, and another application start in a queue that needs its minimum capacity back, only the minimum capacity will be preempted and all of the maximum capacity being used by the other queue will remain until the containers churn naturally.

Queue Ordering Policies

There are currently two types of ordering policies the Capacity Scheduler supports today: FIFO and FAIR. The default that queues start with is FIFO which in my experience is not the behavior that clients expect from their queues. By configuring between FIFO and FAIR at the Queue Leaf level you can create behaviors that lead to throughput driven processing or shared fair processing between applications running. One important thing to know about

ordering policies is that they operated at the application level in the queue and do not care about which user owns the applications.

With the FIFO policy applications are evaluated in order of oldest to youngest for resource allocation. If an application has outstanding resource requests they are immediately fulfilled first come first serve. The result of this is that should an application have enough outstanding requests that they would consume the entire queue multiple times before completing they will block out other applications from starting while they are first allocated resources as the oldest application. It is this very behavior that comes most unexpected to users and creates the most discontent as a users can even block their own applications with their own applications!

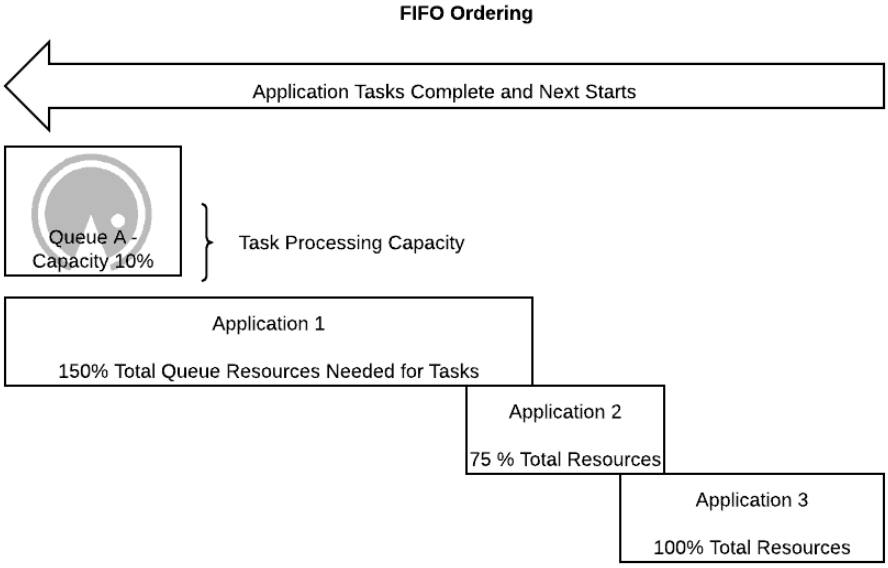


Figure 5.15 FIFO Ordering

That behavior is easily solved for today by using the FAIR ordering policy. When using the FAIR ordering policy on a leaf queue applications are evaluated for resource allocation requests by first the application using the least resources first and most last. This way new applications entering the queue who have no resources for processing will be first asked for their required allocations to get started. Once all applications in the queue have resources they get balanced fairly between all users asking for them.

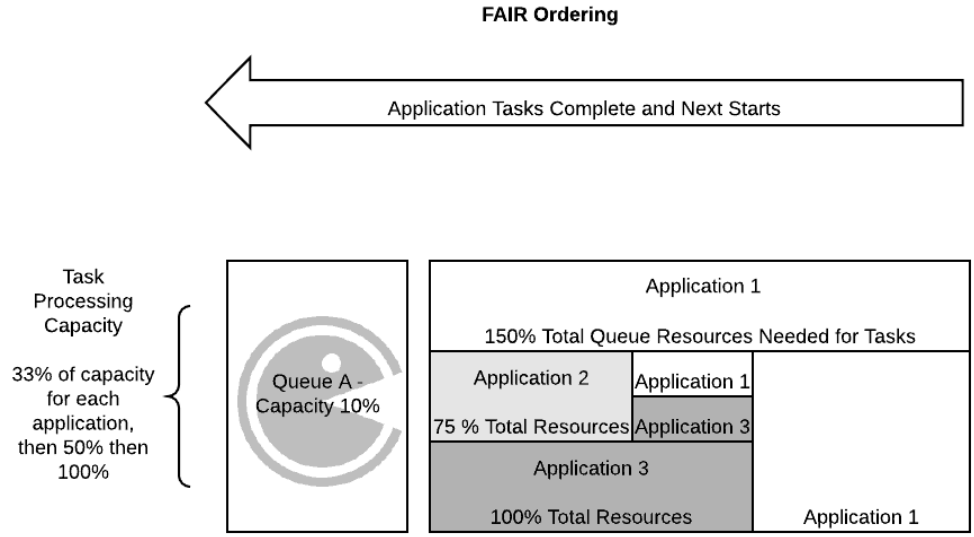


Figure 5.16 Fair ordering

It's important to note that this behavior only occurs if you have good container churn in your queue. Because Preemption does not exist inside of a queue resources cannot be forcefully redistributed within it and the FAIR ordering policy only is concerned with new allocations of resources not current ones; what does this mean? That if the queue is currently utilized with tasks which never complete or run for long periods of time without allowing for container churn to occur in the queue will hold the resources and still prevent applications from executing.

Username and Application Driven Calculations

Calculations in the Capacity Scheduler look at two primary attributes when attempting to provide allocations: User Name and Application ID. When it comes to sharing resources between users in a queue aspects like Minimum User Percentage and User Limit Factor both look at the User Name itself; this can obviously cause some conflicting problems if you're using a service account for multiple users to run jobs as only 1 user will appear to the Capacity Scheduler. Within a Queue which application gets resource allocations are driven by the leaf queues ordering policies: FIFO or FAIR which only care about the application and not which user is running it. In FIFO resources are allocated first to the oldest application in the queue and only when it no longer requires any will the next application gets an allocation. With FAIR applications that are using the least about of resources are first asking if there are pending allocations for them and fulfilled if so, if not the next application with least resources is checked; this helps to evenly share queues with applications that would normally consume them.

Default Queue Mapping

Normally to target a specific queue the user provides some configuration information telling the client tools what queue to request. But commonly users utilize tools that have a hard time passing configurations downstream to target specific queues. With Default Queue Mappings we can route an entity by its username or groups it belongs to into specific queues. Take note that the default queue routing configuration matches whichever routing attribute comes first. So if a group mapping is provided before the user mapping that matches the user he is routed to the queue for the group.

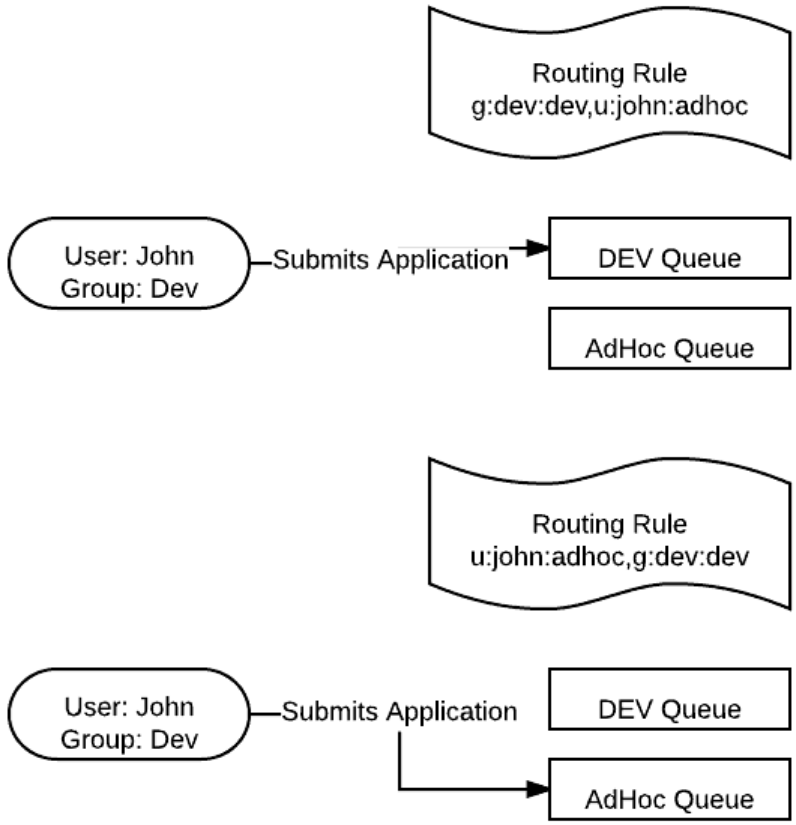


Figure 5.17 Default queue mapping

Priority

When resources are allocated over multiple queues the one with the lowest relative capacity gets resources first. If you're in a scenario where you want to have a high priority queue that receives resources before others then changing to a higher priority is a simple way to do

this. Today using queue priorities with LLAP and Tez allows for more interactive workloads as these queues can be assigned resources at a higher priority to reduce the end latency that an end user may experience.

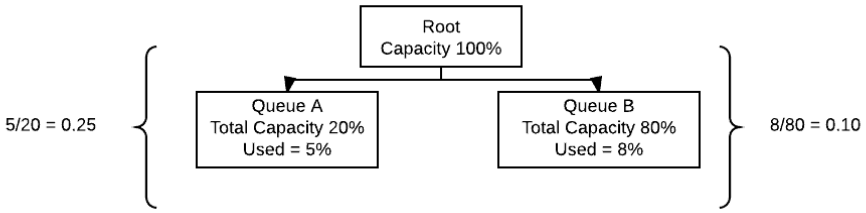


Figure 5.18 Priority queueing

Above the two queues provide an example of how relative capacity is used unmodified by queue priorities. In this case even while Queue A is smaller than Queue B and while Queue B is using more absolute resources it is elected to continue receiving them first because its relative capacity is lower than that of Queue A. If the desired behavior required Queue A to always receive resource allocations first the Queue priority should be increased above that of Queue B. When assigning priority a higher value represents higher priority.

Labels

Labels are better describe as partitions of the cluster. All clusters start with a default label or partition that is exclusive in the sense that as new labeled partitions are added to the cluster they are not shared with the original default cluster partition. Labels be defined as exclusive or shared when created and a node can only have a single label assigned to it. More common uses of labels is for the targeting of GPU hardware in the cluster or to deploy licensed software against only a specific subset of the cluster. Today LLAP also uses Labels to leverage dedicated hosts for long running processes.

Shared labels allow applications in other labeled partitions such as the default cluster to grow into them and utilize the hardware if no specific applications are asking for the label. If an application comes along that targets the label specifically the other applications that were utilizing it will be preempted off the labeled node so the application needing it can make use of it. Exclusive labels are just that, exclusive and will not be shared by any others; Only applications specifically targeting labels will run solely on them. Access to labeled partitions is provided to leaf queues so users able to submit to them are able to target the label. If you wish to autoroute users to labels say creating a GPU queue that automatically uses the GPU label this is possible.

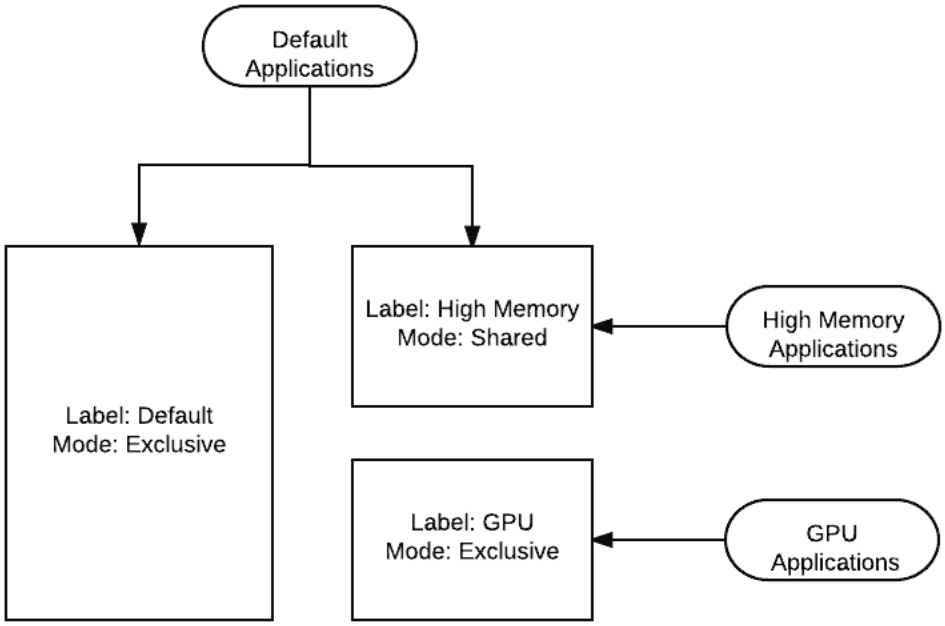


Figure 5.19 Labelling

Queue Names

Queue leaf names have to be unique with the Capacity Scheduler. For example if you created a queue in the capacity scheduler as root.adhoc.dev dev will have to be unique as a leaf over all queue names and you cannot have a root.workflow.dev queue as it would no longer be unique. This is in line with how queues are specified for submission by only using the leaf name and not the entire composite queue name. Parents of leafs are never submitted to directly and have no need to be unique so you can have root.adhoc.dev and root.adhoc.qa without issue as both dev and qa are unique leaf names.

Limiting Applications per Queue

Spawning of a queue by launching many applications into it so that none can effectively complete can create bottlenecks and impact SLAs. At the very worst the entire queue becomes deadlocked and nothing is able to process without and admin physically killing jobs to free up resources for compute tasks. This is easily prevented by placing a limit on the total number of applications allowed to run in the leaf queue, alternatively it's possible to control the % of resources of the leaf that can be used by Application Masters. By default

this value is typically rather large over 10,000 applications (or 20% of leaf resources) and can be configured per leaf if needed otherwise the value is inherited from prior parent queues of the leaf.

Container Sizing

An unknown to many people utilizing the Capacity Scheduler is that containers are multiples in size of the minimum allocation. For example if your minimum scheduler mb of memory per container was 1gb and you requested a 4.5gb sized container the scheduler will round this request up to 5gb. With very high minimums this can create large resource wastage problems for example with a 4gb minimum if we requested 5gb we would be served 8gb providing us with 3 extra GB that we never even planned on using! When configuring and minimum and maximum container size the maximum should be evenly divisible by the minimum.

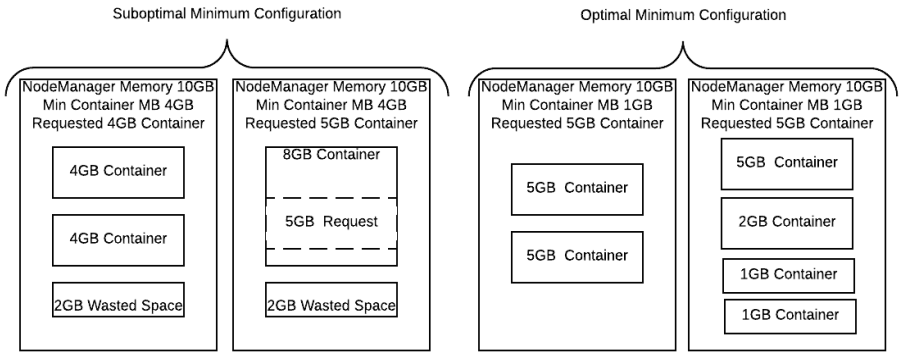


Figure 5.20 Container Seizing

References:

1. Matrix Multiplication https://www.tutorialspoint.com/map_reduce/map_reduce_algorithm.htm
2. Fair Scheduler: https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/FairScheduler.html
3. https://knpcode.com/hadoop/yarn/fair-scheduler-yarn/
4. Hadoop Capacity scheduler: https://blog.cloudera.com/yarn-capacity-scheduler/
5. https://hortonworks.com/blog/better-slas-via-resource-preemption-in-yarns-capacityscheduler/
6. Data serialization: https://www.xenonstack.com/blog/data-serialization-hadoop#:~:text=Architecture%2C%20Tools%2C%20Challenges-,Guide%20to%20Data%20Serialization%20in%20Hadoop,regardless%20of%20the%20system%20architecture.