

Lesson 2

MapReduce Basics:

Spontaneous fulfillment of desires

Wholeness of the Lesson

The input to a MapReduce job is at the DFS. Output key-value pairs from each reducer are written back onto the DFS (whereas intermediate key-value pairs are transient and not preserved). *TM is a simple, effortless mental technique that can be used by anyone, no matter what their lifestyle. It promotes non-procedural (spontaneous) fulfillment of desires, by bringing the desires of the individual into accord with Natural Law, without the individual having to know the underlying mechanism.*

MapReduce

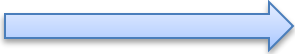
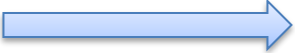
1. Divide and Conquer
2. Functional Programming
 1. Map
 2. Reduce

MapReduce

Key-value pairs form the basic data structure in MapReduce.

Keys and values may be primitives (such as integers, floating point values, strings, and raw bytes) or objects (such as lists, tuples, associative arrays, etc.). Programmers typically need to define their own classes.

In MapReduce, the programmer defines a mapper and a reducer with the following signatures:

- map: (k1; v1)  [(k2; v2)]
- reduce: (k2; [v2])  [(k3; v3)]

Mappers

The input to a MapReduce job is the data stored on the underlying distributed file system.

A **Mapper** object is created for each input-split. Thus the number of **map tasks** is determined by the number of input-splits.

Very Important

In Hadoop 1.x, a task tracker is created for each input-split. Each task tracker has a map task to perform.

In order to perform the map task, the user created Mapper object is sent to the datanode (with input-split).

Map Task

Mapper class has a **map method**.

An input-split has many records.

The “map” method is invoked for each record in the input-split.

Very Important

- (1) Do not write a “loop” to iterate over all records in the input-split. That is taken care of by the framework.
- (2) The map method process just one record at a time and it must just extract “absolutely essential data” and emit it as a key-value pair.

Main Point 1

A Mapper object is created for each input-split and thus the number of “map tasks” is determined by the number of input-splits. An input-split has many records and the “map method” is called for each record in the input-split. *Nature is capable of harmoniously organizing the entire universe from an unmanifest level.*

Mappers and Reducers

Framework receives all key-value pairs emitted by all the Mappers.

Framework then performs Shuffle and Sort.

Shuffle and Sort

Framework uses a “getPartition” method to partition the intermediate key space. Framework then “Shuffles” the intermediate key-value pairs. (That means, the partitioned key-value pairs are sent to their destination.)

At the destination, key-value pairs arrive from all map tasks.

All those key-value pairs are “group by key” followed by “Sort by key”.

Reducers

Number of “**reduce tasks**” is specified by the user in the job configuration. That many **Reducer Objects** are created by the framework and each reducer object is responsible for a partition of the intermediate key space.

Reducer class has a **reduce** method. The reduce method is called for each **key-value** pair in the partition.

Reducers

Very Important.

(1) Reducer input is sorted by the key

(2) However, no ordering relationship is guaranteed for keys across different reducers.

Reducers

Output key-value pairs from each reducer are written persistently back onto the distributed file system (whereas intermediate key-value pairs are transient and not preserved).

The output ends up in r files on the distributed file system, where r is the number of reducers.

For the most part, there is no need to consolidate reducer output, since the r files often serve as input to yet another MapReduce job.

Word count algorithm

```
class Mapper
```

```
    method Map(docid a, doc d)
```

```
        for all term t in the record r do        // r is a record of d
```

```
            Emit(t, 1)
```

```
class Reducer
```

```
    method Reduce(term t, integer [c1, c2, ...])
```

```
        sum = 0
```

```
        for all c in [c1, c2,...] do
```

```
            sum = sum + c
```

```
        Emit(t, sum)
```

Hadoop vs. Google

In Hadoop, the reducer is presented with a key and an iterator over all values associated with the particular key. The values are arbitrarily ordered.

Google's implementation allows the programmer to specify a secondary sort key for ordering the values (if desired) in which case values associated with each key would be presented to the developer's reduce code in sorted order.

In Google's implementation the programmer is not allowed to change the key in the reducer. That is, the reducer output key must be exactly the same as the reducer input key.

In Hadoop, there is no such restriction, and the reducer can emit an arbitrary number of output key-value pairs (with different keys).

Notes

1. Mappers and reducers can perform arbitrary computations over their inputs.
2. Be careful about use of external resources since multiple mappers or reducers may be contending for those resources. For example, it may be unwise for a mapper to query an external SQL database.
3. Mappers can emit any number of intermediate key-value pairs, and they need not be of the same type as the input key-value pairs.
4. Reducers can emit an arbitrary number of final key-value pairs, and they can differ in type from the intermediate key-value pairs.

Special cases

Case 1. MapReduce programs can contain no reducers, in which case mapper output is directly written to disk (one file per mapper). Example problems: parse a large text collection or independently analyze a large number of images.

Case 2. MapReduce program with no mappers is not possible, although in some cases it is useful for the mapper to implement the identity function and simply pass input key-value pairs to the reducers. This has the effect of sorting and regrouping the input for reduce-side processing.

Special cases

Case 3. Similarly, in some cases it is useful for the reducer to implement the identity function, in which case the program simply sorts and groups mapper output.

Case 4. Running identity mappers and reducers has the effect of regrouping and resorting the input data (which is sometimes useful).

PARTITIONERS AND COMBINERS

Partitioners are responsible for dividing up the intermediate key space. In other words, the partitioner specifies the task to which an intermediate key-value pair must be copied.

Within each reducer, keys are processed in sorted order (which is how the “group by” is implemented).

The simplest partitioner involves computing the hash value of the key and then taking the mod of that value with the number of reducers. This assigns approximately the same number of keys to each reducer.

PARTITIONERS AND COMBINERS

The partitioner only considers the key and ignores the value.

User has the ability to override the method

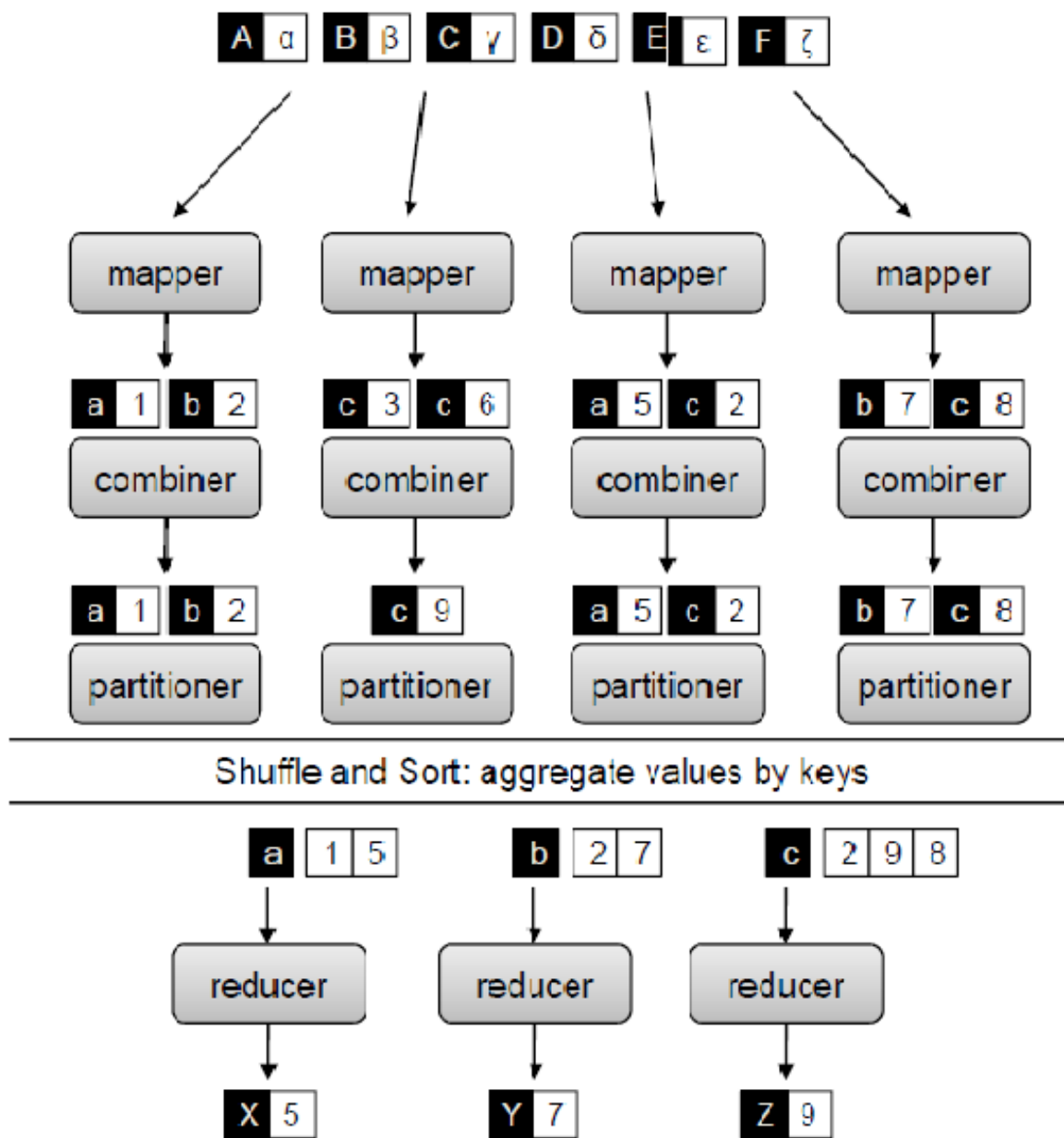
`getPartition(Key k, int numReducers)`

which has the following default implementation.

```
int getPartition(Key k, int numReducers)  
    return Math.abs(k.hashCode()) % numReducers;
```

Combiners

1. Combiners are an optimization in MapReduce that allow for local aggregation before the shuffle and sort phase.
2. In this class, we will not study combiners. Rather, we will study a more efficient technique to achieve the same goal in a much more reliable manner. It is called “In-Mapper Combining Technique”. We will study that in our next lecture.



Job

A complete MapReduce job consists of

1. Mapper class,
2. Reducer class,
3. (optional. We will not) Combiner class
4. (optional) Partitioner class
5. Job configuration parameters.

MapReduce Workflow

Datanode (of a Mapper)

MAP

Your Mapper class
It contains a map
method.

SHUFFLE

Framework partitions
the mapper output
and sends to Reducers

Datanode (of a Reducer)

SORT

Framework receives key
value pairs from all
Mappers
Sort them by key
Group them by key
Prepares reducer input

REDUCE

Your Reducer class
It contains a reduce
method. It outputs to HDFS

Main Point 2

Intermediate data arrive at each reducer in order, sorted by the key without any ordering relationship on keys across different reducers. The reduce method is applied to all values associated with the same intermediate key to generate output key-value pairs. *Sometimes it is necessary to step back from the points to see the wholeness. TM promotes the ability to see the wholeness as well as the point value, the larger picture as well as the details.*

.

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

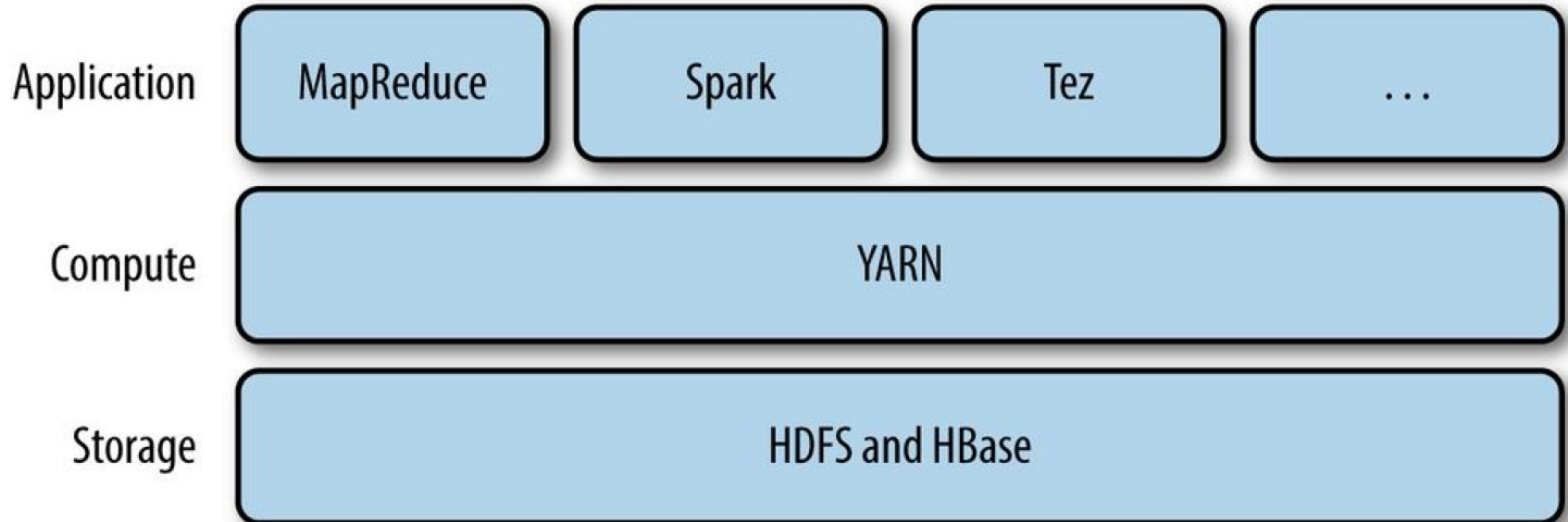
1. MapReduce is a simple but elegant programming paradigm.
 2. MapReduce paradigm can be used to solve a wide variety of problems.
-
3. ***Transcendental consciousness:*** *is a silent field of all possibilities, the basis of any desired outcome.*
 4. ***Impulses within the Transcendental Field:*** *Transcendental consciousness has infinite energy, infinite creativity, and infinite intelligence, which allows the impulses within the transcendental field to create anything, giving it the qualities of infinite flexibility and infinite power.*
 5. ***Wholeness moving within itself:*** *In unity consciousness, any desire is projected from the field of pure consciousness and therefore is fulfilled immediately*



Appendix

Hadoop 2.x and Yarn

- YARN(Yet Another Resource Negotiator) is one of the key features in Hadoop 2
- Originally described by Apache as a redesigned resource manager, YARN is now characterized as a large-scale, distributed operating system for big data applications.



YARN

- YARN is a software rewrite that decouples **MapReduce's resource management and scheduling capabilities** from the **data processing component**
 - enabling Hadoop to support more varied processing approaches and a broader array of applications.
- For example, Hadoop cluster can now run interactive querying and streaming data applications simultaneously with MapReduce batch jobs.
- The original incarnation of Hadoop closely paired the Hadoop Distributed File System (HDFS) with the batch-oriented MapReduce programming framework, which **handles resource management and job scheduling on Hadoop systems.**

YARN

- YARN combines a central **resource manager** that reconciles the way applications use Hadoop system resources with **node manager** agents that monitor the processing operations of individual cluster nodes.
- Running on commodity hardware clusters, Hadoop has attracted particular interest as a staging area and data store for large volumes of structured and unstructured data intended for use in analytics applications.
- Separating HDFS from MapReduce with YARN makes the Hadoop environment more suitable for operational applications that can't wait for batch jobs to finish.

Anatomy of a YARN Application Run

YARN provides its core services via two daemons:

- a **resource manager** (one per cluster) to manage the use of resources across the cluster
- **node managers** on all the nodes in the cluster to launch and monitor **containers**. A container executes application-specific process with constrained set of resources (RAM, CPU, etc.).

STEP 1: To run an application, a client contacts the resource manager and asks it to run an **application master process**.

STEP 2: The resource manager then finds a node manager that can launch an application master in a container

Anatomy of a YARN Application Run

Precisely what the application master does once it is running depends on the application. It could simply run a computation in the container it is running in and return the result to the client. Or, it could request more containers from the resource manager and use them to run a distributed computation (MapReduce does this).

<http://ercoppa.github.io/HadoopInternals/AnatomyMapReduceJob.html>

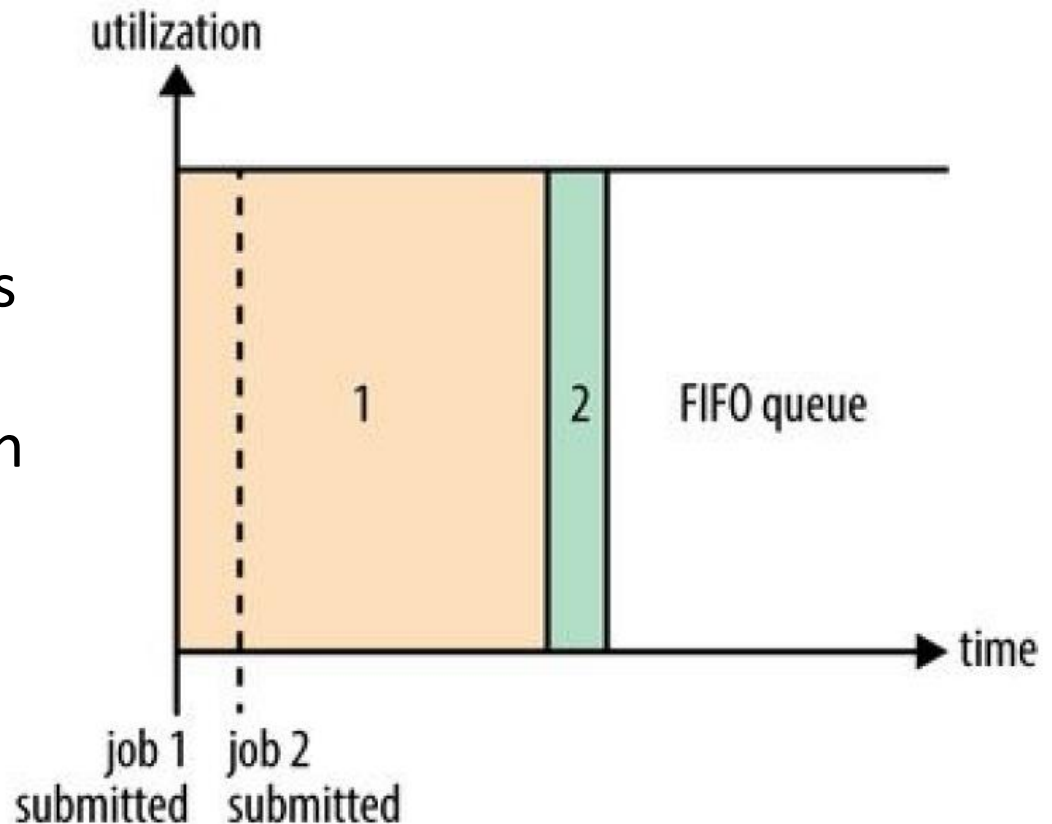
Scheduling in YARN

- In an ideal world, the requests that a YARN application makes would be granted immediately.
- In the real world, however, resources are limited, and on a busy cluster, an application will often need to wait to have some of its requests fulfilled.
- Scheduling in general is a difficult problem and there is no one “best” policy, which is why YARN provides a choice of schedulers and configurable policies

Scheduling in YARN

FIFO (first in, first out)

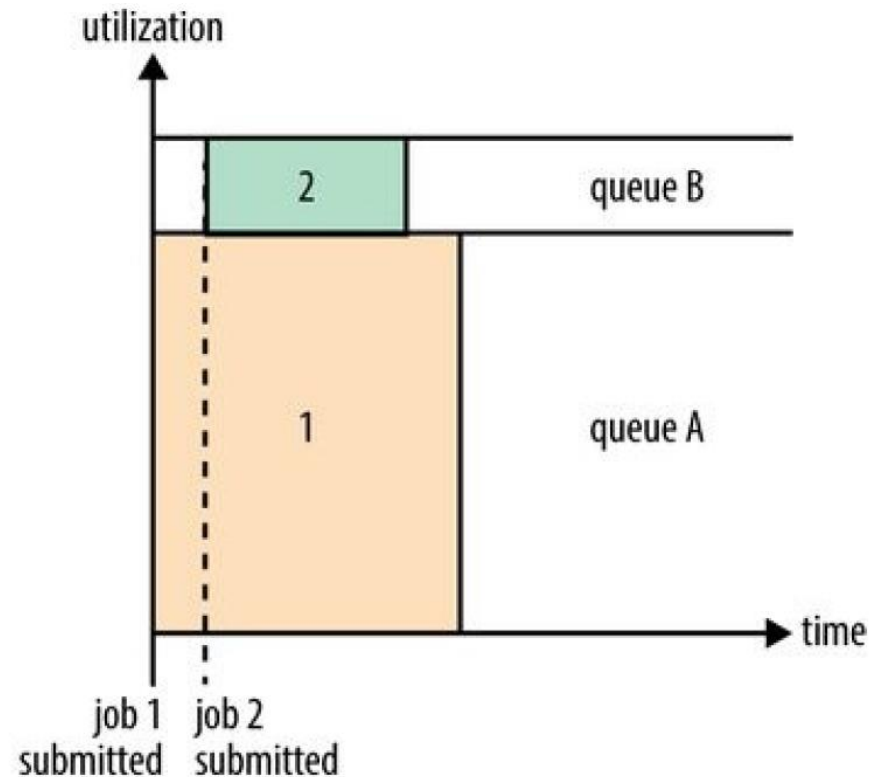
- The FIFO Scheduler places applications in a queue and runs them in the order of submission.
- Requests for the first application in the queue are allocated first; once its requests have been satisfied, the next application in the queue is served, and so on.



Scheduling in YARN

Capacity Scheduler

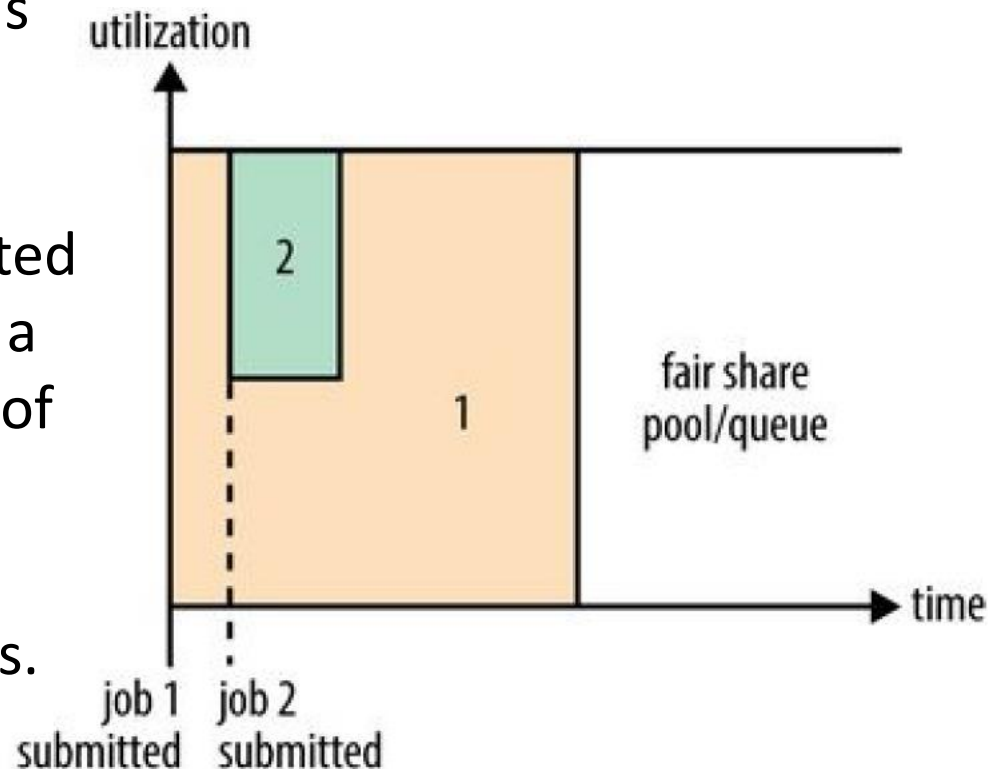
- The Capacity Scheduler allows sharing of a Hadoop cluster along organizational lines, whereby each organization is allocated a certain capacity of the overall cluster.
- Queues may be further divided in hierarchical fashion allowing each organization to share its cluster allowance between different groups of users.
- Within a queue, applications are scheduled using FIFO scheduling.



Scheduling in YARN

Fair Scheduler

- The Fair Scheduler attempts to allocate resources so that all running applications get the same share of resources.
- When job 1 is submitted, it is allocated all resources available.
- Assume that job 2 is submitted while job 1 is running. After a while, each job is using half of the resources.
- Job 1 is allocated full resources after job 2 finishes.



Hadoop 2

- Hadoop 2 adds support for running non-batch applications through the introduction of YARN.
- YARN puts resource management and job scheduling functions in a separate layer beneath the data processing one
 - enabling Hadoop 2 to run a variety of applications (big data analytics and other enterprise applications).
- For example, it is now possible to run event processing as well as streaming, real-time and operational applications.
- The capability to support programming frameworks other than MapReduce also means that Hadoop can serve as a platform for a wider variety of big data analytical applications.

Hadoop 2

- Hadoop 2 also includes new features designed to improve system availability and scalability.
- For example, it introduced an Hadoop Distributed File System (HDFS) **high-availability** (HA) feature that brings a new NameNode architecture to Hadoop:
 - allows users to configure clusters with redundant NameNodes, removing the chance that a lone NameNode will become a single point of failure (SPoF) within a cluster.
 - a new HDFS federation capability lets clusters be built out horizontally with multiple NameNodes that work independently but share a common data storage pool, offering better compute scaling as compared to Apache Hadoop 1.x.