Hadoop Fundamentals

zzak00

February 3, 2022

Abstract

Grace Hopper -: In pioneer days they used oxen for heavy pulling, and when one ox couldn't budge a log, they didn't try to grow a larger ox. We shouldn't be trying for bigger computers, but for more systems of computers.

I. Hadoop Fundamentals

Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.

i. Hadoop Ecosystem

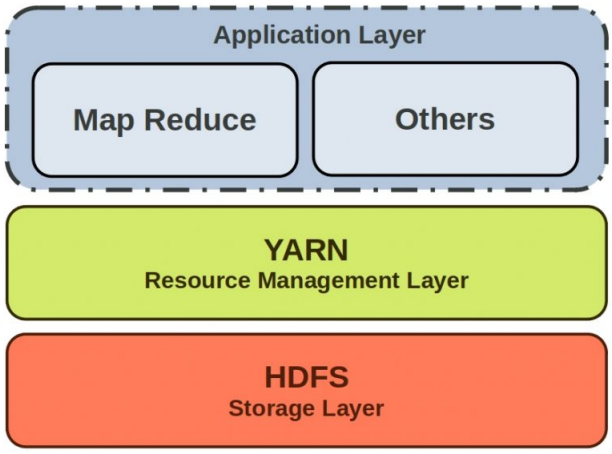

Figure 1: *Hadoop EcoSystem.*

II. HDFS

i. HDFS Design

HDFS, which stands for *Hadoop Distributed Filesystem*, is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware. In details:

- *Very large files* : Very large in this context means files that are hundreds of megabytes, gigabytes, or terabytes in size.
- *Streaming Data access* : HDFS is built around the idea that the most efficient data processing pattern is a write-once, read-many-times pattern. Hadoop HDFS is mainly designed for batch processing rather than interactive use by users. The force is on high throughput of data access rather than low latency of data access. It focuses on how to retrieve data at the fastest possible speed while analyzing logs.
- *Commodity hardware / Hardware failure* : Hadoop doesn't require expensive, highly reliable hardware. It's designed to run on of commodity hardware (commonly available hardware that can be obtained from multiple vendors) for which the chance of node failure across the cluster is high, at least for large clusters. HDFS is designed to carry on working without a noticeable interruption to the user in the face of such failure.
- *Moving computation is cheaper than moving data* : If an application does the computation near the data it operates on, it is much more efficient than done far of. This fact becomes stronger while dealing with large data set. The main advantage of this is that it increases the overall throughput of the system. It also minimizes network congestion. The assumption is that it is better to move computation closer to data instead of moving data to computation.

ii. HDFS Concepts

ii.1 NameNode

An HDFS cluster has two types of nodes operating in a masterworker pattern: a namenode (the master) and a number of datanodes (workers). The namenode manages the filesystem namespace. It maintains the filesystem tree and the metadata for all the files and directories in the tree. This information is stored persistently on the local disk in the form of two files: the fsimage and the edit log. The namenode also knows the datanodes on which all the blocks for a given file are located; however, it does not store block locations persistently, because this information is reconstructed from datanodes when the system starts.

- **Fsimage**: Fsimage stands for File System image. It contains the complete namespace of the Hadoop file system since the NameNode creation.
- **Edit log**: It contains all the recent changes performed to the file system namespace to the most recent Fsimage.

ii.2 DataNode

Datanodes are the workhorses of the filesystem. They store and retrieve blocks when they are told to (by clients or the namenode), and they report back to the namenode periodically with lists of blocks that they are storing.

ii.3 SecondaryNameNode

Apart from DataNode and NameNode, there is another daemon called the secondary NameNode. Secondary NameNode works as a helper node to primary NameNode but doesn't replace primary NameNode.When the NameNode starts, the NameNode merges the Fsimage and edit logs file to restore the current file system namespace. Since the NameNode runs continuously for a long time without any restart, the size of edit logs becomes too large. This will result in a long restart time for NameNode.
Secondary NameNode solves this issue. Secondary NameNode downloads the Fsimage file and edit logs file from NameNode.It periodically applies edit logs to Fsimage and refreshes the edit logs. The updated Fsimage is then sent to the NameNode so that NameNode doesn't have to re-apply the edit log records during its restart. This keeps the edit log size small and reduces the NameNode restart time.
If the NameNode fails, the last save Fsimage on the secondary NameNode can be used to recover file system metadata. The secondary NameNode performs regular checkpoints in HDFS.

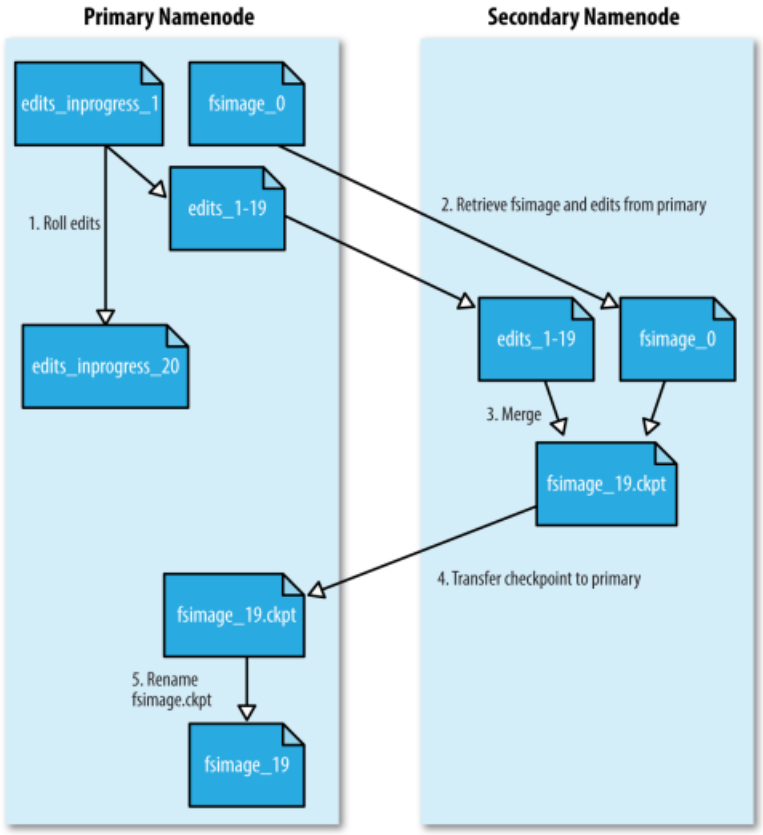

Figure 2: *The checkpointing process.*

ii.4 Blocks

A disk has a block size, which is the minimum amount of data that it can read or write. Filesystems for a single disk build on this by dealing with data in blocks, which are an integral multiple of the

disk block size. Filesystem blocks are typically a few kilobytes in size, whereas disk blocks are normally 512 bytes. This is generally transparent to the filesystem user who is simply reading or writing a file of whatever length. However, there are tools to perform filesystem maintenance, such as df and fsck, that operate on the filesystem block level.
HDFS, too, has the concept of a block, but it is a much larger unit—128 MB by default. Like in a filesystem for a single disk, files in HDFS are broken into block-sized chunks, which are stored as independent units. Unlike a filesystem for a single disk, a file in HDFS that is smaller than a single block does not occupy a full block's worth of under- lying storage. (For example, a 1 MB file stored with a block size of 128 MB uses 1 MB of disk space, not 128 MB.)

Why Is a Block in HDFS So Large?

HDFS blocks are large compared to disk blocks, and the reason is to minimize the cost of seeks. If the block is large enough, the time it takes to transfer the data from the disk can be significantly longer than the time to seek to the start of the block. Thus, transferring a large file made of multiple blocks operates at the disk transfer rate.

A quick calculation shows that if the seek time is around 10 ms and the transfer rate is 100 MB/s, to make the seek time 1% of the transfer time, we need to make the block size around 100 MB. The default is actually 128 MB, although many HDFS installations use larger block sizes. This figure will continue to be revised upward as transfer speeds grow with new generations of disk drives.

This argument shouldn't be taken too far, however. Map tasks in MapReduce normally operate on one block at a time, so if you have too few tasks (fewer than nodes in the cluster), your jobs will run slower than they could otherwise.

ii.5 Replication Management

For a distributed system, the data must be redundant to multiple places so that if one machine fails, the data is accessible from other machines (Fault tolerance). In Hadoop, HDFS stores replicas of a block on multiple DataNodes based on the replication factor which is the number of copies to be created for blocks of a file in HDFS architecture.

ii.6 RackAwareness in HDFS

Rack is the collection of around 40-50 machines (DataNodes) connected using the same network switch. If the network goes down, the whole rack will be unavailable.
Hadoop's default strategy is to place the first replica on the same node as the client (for clients running outside the cluster, a node is chosen at random, although the system tries not to pick nodes that are too full or too busy). The second replica is placed on a different rack from the first (off-rack), chosen at random. The third replica is placed on the same rack as the second, but on a different node chosen at random. Further replicas are placed on random nodes in the cluster, although the system tries to avoid placing too many replicas on the same rack.

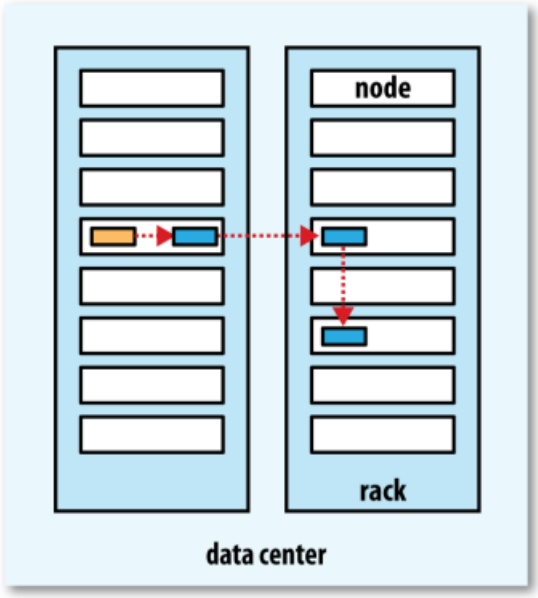

Figure 3: *The checkpointing process.*

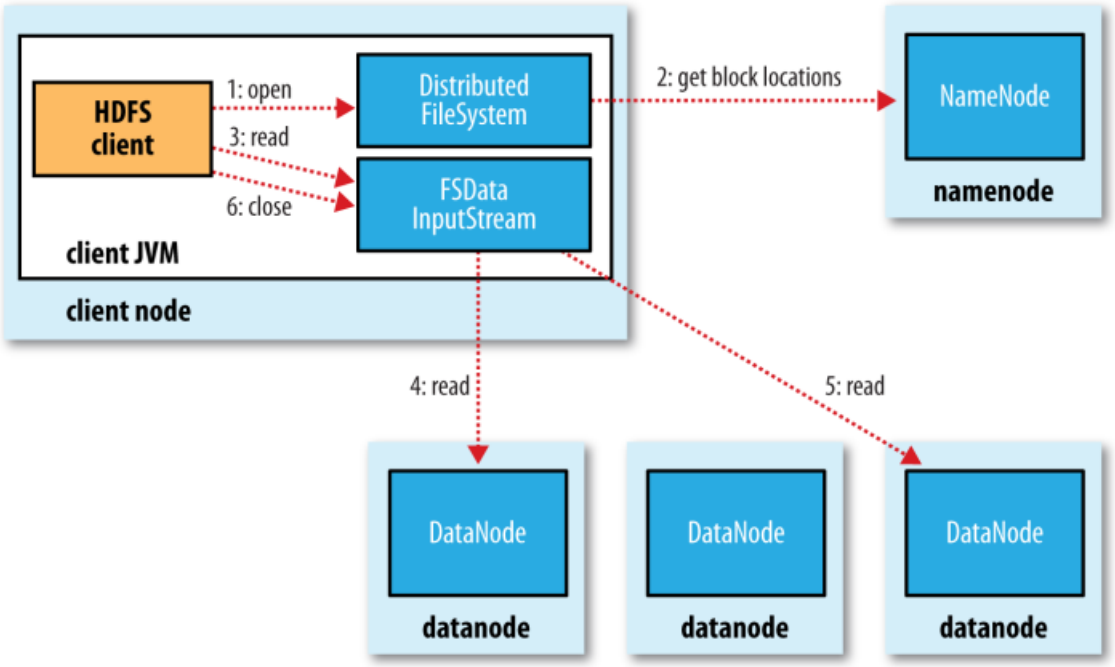

Figure 4: *A client reading data from HDFS.png.*

ii.7 Anatomy of a File Read

The client opens the file it wishes to read by calling open() on the FileSystem object, which for HDFS is an instance of DistributedFileSystem (step 1). DistributedFileSystem calls the namenode,

using remote procedure calls (RPCs), to determine the locations of the first few blocks in the file (step 2). For each block, the namenode returns the addresses of the datanodes that have a copy of that block. Furthermore, the datanodes are sorted according to their proximity to the client. If the client is itself a datanode (in the case of a MapReduce task, for instance), the client will read from the local datanode if that datanode hosts a copy of the block.

The DistributedFileSystem returns an FSDataInputStream (an input stream that supports file seeks) to the client for it to read data from. FSDataInputStream in turn wraps a DFSInputStream , which manages the datanode and namenode I/O.

The client then calls read() on the stream (step 3). DFSInputStream , which has stored the datanode addresses for the first few blocks in the file, then connects to the first (closest) datanode for the first block in the file. Data is streamed from the datanode back to the client, which calls read() repeatedly on the stream (step 4). When the end of the block is reached, DFSInputStream will close the connection to the datanode, then find the best datanode for the next block (step 5). This happens transparently to the client, which from its point of view is just reading a continuous stream.

Blocks are read in order, with the DFSInputStream opening new connections to datanodes as the client reads through the stream. It will also call the namenode to retrieve the datanode locations for the next batch of blocks as needed. When the client has finished reading, it calls close() on the FSDataInputStream (step 6).

During reading, if the DFSInputStream encounters an error while communicating with a datanode, it will try the next closest one for that block. It will also remember datanodes that have failed so that it doesn't needlessly retry them for later blocks. The DFSInput Stream also verifies checksums for the data transferred to it from the datanode. If a corrupted block is found, the DFSInputStream attempts to read a replica of the block from another datanode; it also reports the corrupted block to the namenode.

One important aspect of this design is that the client contacts datanodes directly to retrieve data and is guided by the namenode to the best datanode for each block. This design allows HDFS to scale to a large number of concurrent clients because the data traffic is spread across all the datanodes in the cluster. Meanwhile, the namenode merely has to service block location requests (which it stores in memory, making them very efficient) and does not, for example, serve data, which would quickly become a bottle- neck as the number of clients grew.

ii.8 Anatomy of a File Write

The client creates the file by calling create() on DistributedFileSystem (step 1). DistributedFileSystem makes an RPC call to the namenode to create a new file in the filesystem's namespace, with no blocks associated with it (step 2). The namenode performs various checks to make sure the file doesn't already exist and that the client has the right permissions to create the file. If these checks pass,the namenode makes a record of the new file; otherwise, file creation fails and the client is thrown an IOException . The DistributedFileSystem returns an FSDataOutputStream for the client to start writing data to. Just as in the read case, FSDataOutputStream wraps a DFSOutputStream , which handles communication with the datanodes and namenode.

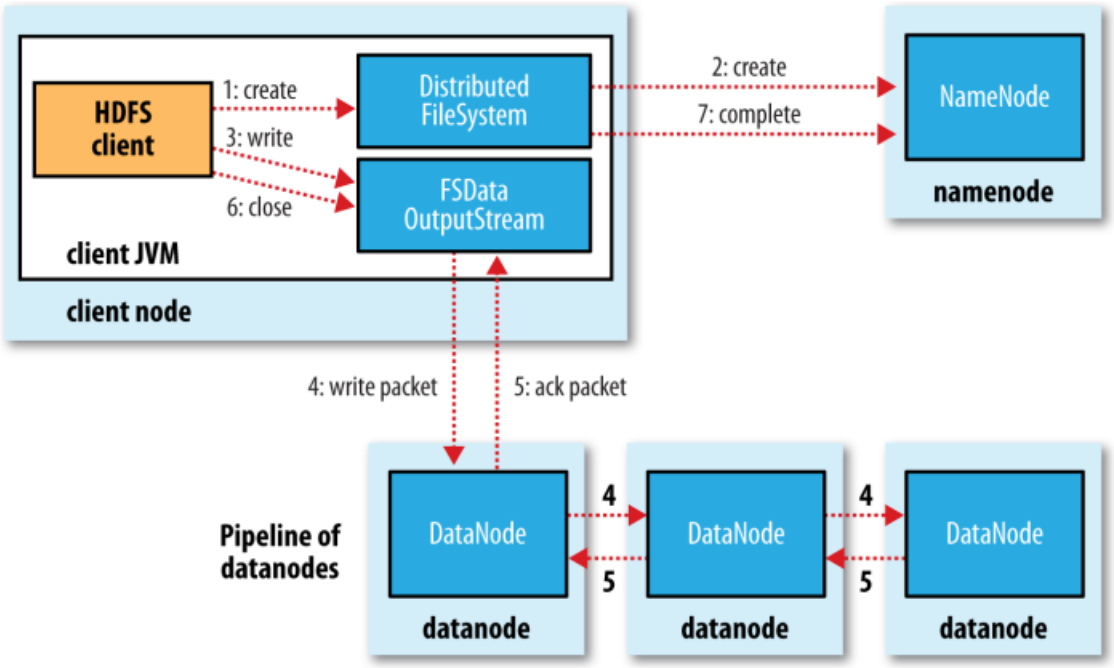

Figure 5: *A client writing data from HDFS.png.*

As the client writes data (step 3), the DFSOutputStream splits it into packets, which it writes to an internal queue called the data queue. The data queue is consumed by the DataStreamer , which is responsible for asking the namenode to allocate new blocks by picking a list of suitable datanodes to store the replicas. The list of datanodes forms a pipeline, and here we'll assume the replication level is three, so there are three nodes in the pipeline. The DataStreamer streams the packets to the first datanode in the pipeline, which stores each packet and forwards it to the second datanode in the pipeline. Similarly, the second datanode stores the packet and forwards it to the third (and last) datanode in the pipeline (step 4).

The DFSOutputStream also maintains an internal queue of packets that are waiting to be acknowledged by datanodes, called the ack queue. A packet is removed from the ack queue only when it has been acknowledged by all the datanodes in the pipeline (step 5).

If any datanode fails while data is being written to it, then the following actions are taken, which are transparent to the client writing the data. First, the pipeline is closed, and any packets in the ack queue are added to the front of the data queue so that datanodes that are downstream from the failed node will not miss any packets. The current block on the good datanodes is given a new identity, which is communicated to the namenode, so that the partial block on the failed datanode will be deleted if the failed datanode recovers later on. The failed datanode is removed from the pipeline, and a new pipeline is constructed from the two good datanodes. The remainder of the block's data is written to the good datanodes in the pipeline. The namenode notices that the block is under-replicated, and it arranges for a further replica to be created on another node. Subsequent blocks are then treated as normal.
It's possible, but unlikely, for multiple datanodes to fail while a block is being written. As long as

dfs.namenode.replication.min replicas (which defaults to 1) are written, the write will succeed, and the block will be asynchronously replicated across the cluster until its target replication factor is reached (dfs.replication , which defaults to 3).

When the client has finished writing data, it calls close() on the stream (step 6). This action flushes all the remaining packets to the datanode pipeline and waits for ac- knowledgments before contacting the namenode to signal that the file is complete (step 7). The namenode already knows which blocks the file is made up of (because Data Streamer asks for block allocations), so it only has to wait for blocks to be minimally replicated before returning successfully.

III. YARN

Apache YARN (Yet Another Resource Negotiator) is Hadoop's cluster resource management system.
YARN provides APIs for requesting and working with cluster resources, but these APIs are not typically used directly by user code. Instead, users write to higher-level APIs provided by distributed computing frameworks, which themselves are built on YARN and hide the resource management details from the user.

i. Anatomy of a YARN Application Run

YARN provides its core services via two types of long-running daemon: a resource manager (one per cluster) to manage the use of resources across the cluster, and node managers running on all the nodes in the cluster to launch and monitor containers. A container executes an application-specific process with a constrained set of resources (memory, CPU, and so on).

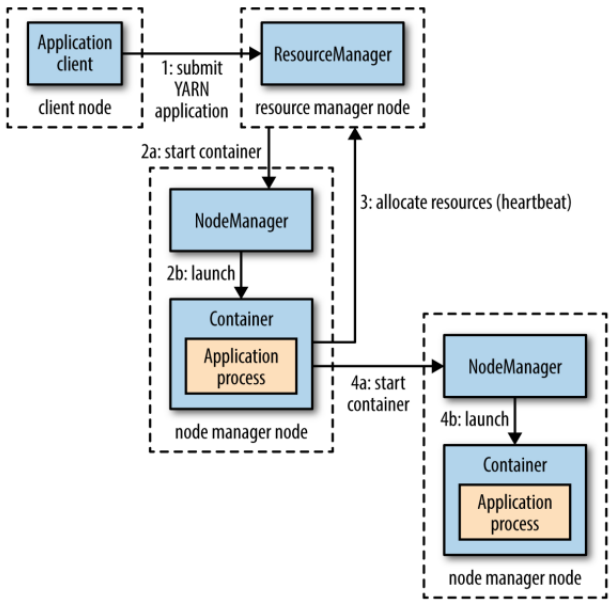

Figure 6: *How YARN runs an application.*

To run an application on YARN, a client contacts the resource manager and asks it to run an application master process (step 1). The resource manager then finds a node manager that can

launch the application master in a container (steps and 2b). 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 managers (step 3), and use them to run a distributed computation (steps 4a and 4b).

ii. 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. It is the job of the YARN scheduler to allocate resources to applications according to some defined policy. 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.

ii.1 Scheduler Options

Three schedulers are available in YARN: the FIFO, Capacity, and Fair Schedulers. The FIFO Scheduler places applications in a queue and runs them in the order of submission (first in, first out). 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.
The FIFO Scheduler has the merit of being simple to understand and not needing any configuration, but it's not suitable for shared clusters. Large applications will use all the resources in a cluster, so each application has to wait its turn. On a shared cluster it is better to use the Capacity Scheduler or the Fair Scheduler. Both of these allow long running jobs to complete in a timely manner, while still allowing users who are running concurrent smaller ad hoc queries to get results back in a reasonable time.
The difference between schedulers, which shows that under the FIFO Scheduler (i) the small job is blocked until the large job completes.
With the Capacity Scheduler (ii), a separate dedicated queue allows the small job to start as soon as it is submitted, although this is at the cost of overall cluster utilization since the queue capacity is reserved for jobs in that queue. This means that the large job finishes later than when using the FIFO Scheduler.
With the Fair Scheduler (iii), there is no need to reserve a set amount of capacity, since it will dynamically balance resources between all running jobs. Just after the first (large) job starts, it is the only job running, so it gets all the resources in the cluster. When the second (small) job starts, it is allocated half of the cluster resources so that each job is using its fair share of resources.
Note that there is a lag between the time the second job starts and when it receives its fair share, since it has to wait for resources to free up as containers used by the first job complete. After the small job completes and no longer requires resources, the large job goes back to using the full cluster capacity again. The overall effect is both high cluster utilization and timely small job completion.

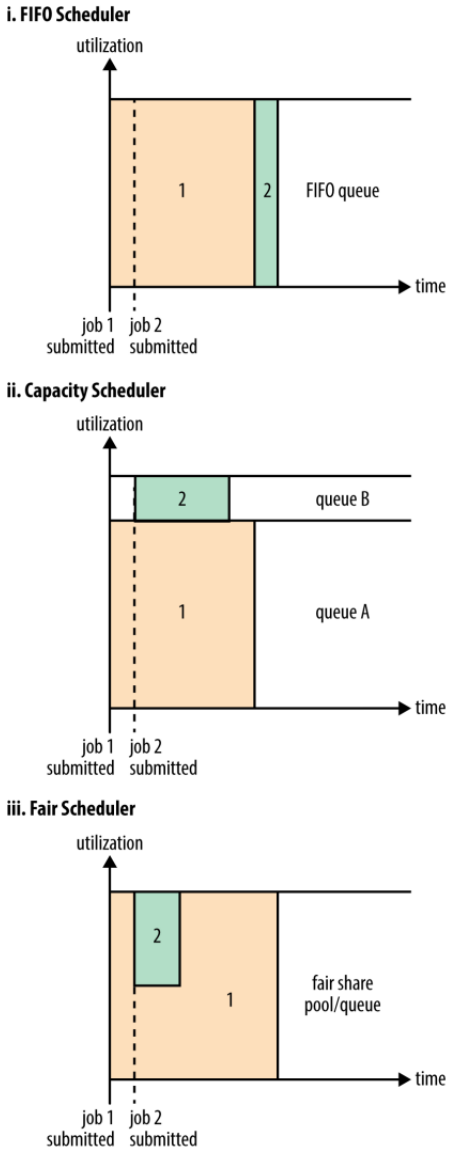

Figure 7: *Cluster utilization over time when running a large job and a small job under the FIFO Scheduler (i), Capacity Scheduler (ii), and Fair Scheduler (iii).*

IV. MAP REDUCE

i. Anatomy of a MapReduce Job Run

MapReduce job is called with a single method call: submit() on a Job object (you can also call waitForCompletion() , which submits the job if it hasn't been submitted already, then waits for it to finish). 1 This method call conceals a great deal of processing behind the scenes. This section uncovers the steps Hadoop takes to run a job. At the highest level, there are five independent entities:

- The client, which submits the MapReduce job.

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

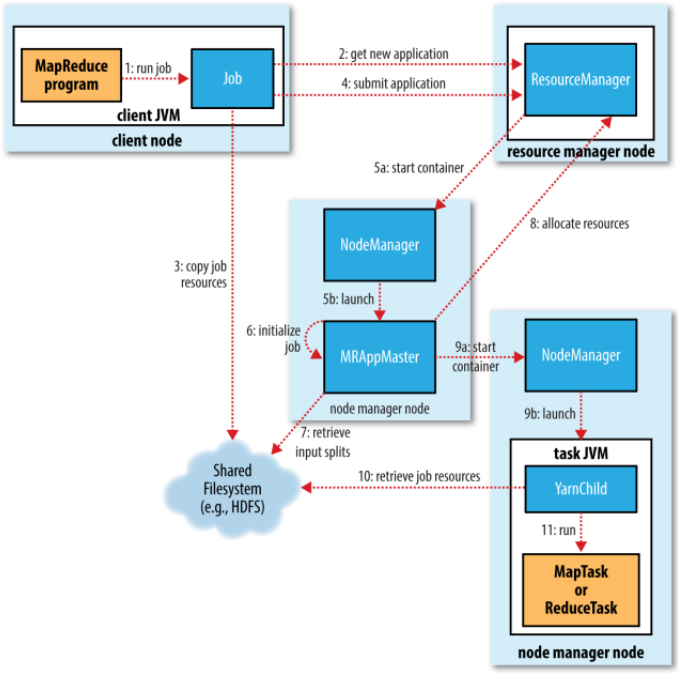

Figure 8: *How Hadoop runs a MapReduce job.*

References

[Figueredo and Wolf, 2009] Figueredo, A. J. and Wolf, P. S. A. (2009). Assortative pairing and life history strategy - a cross-cultural study. *Human Nature*, 20:317–330.