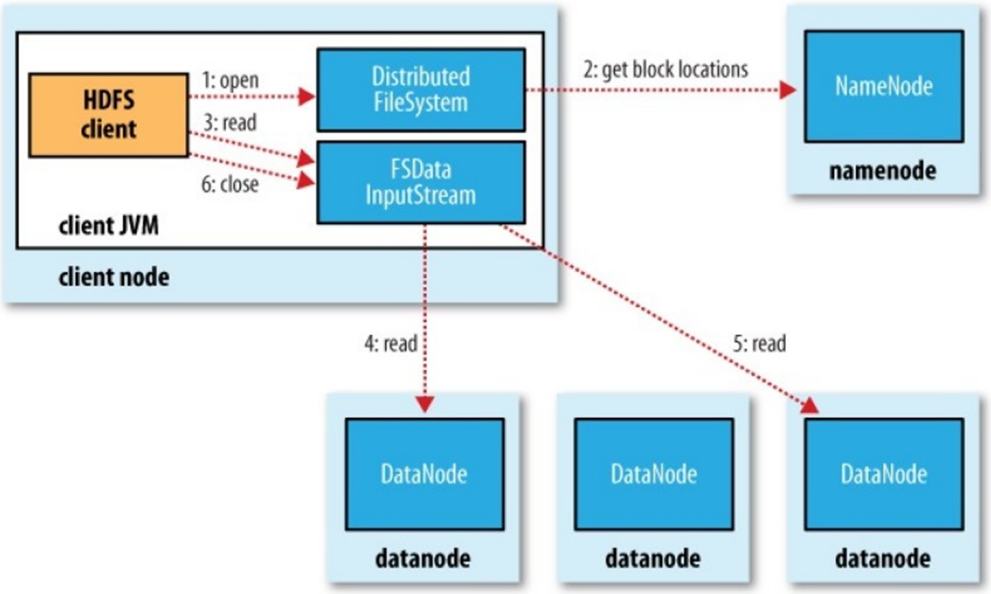
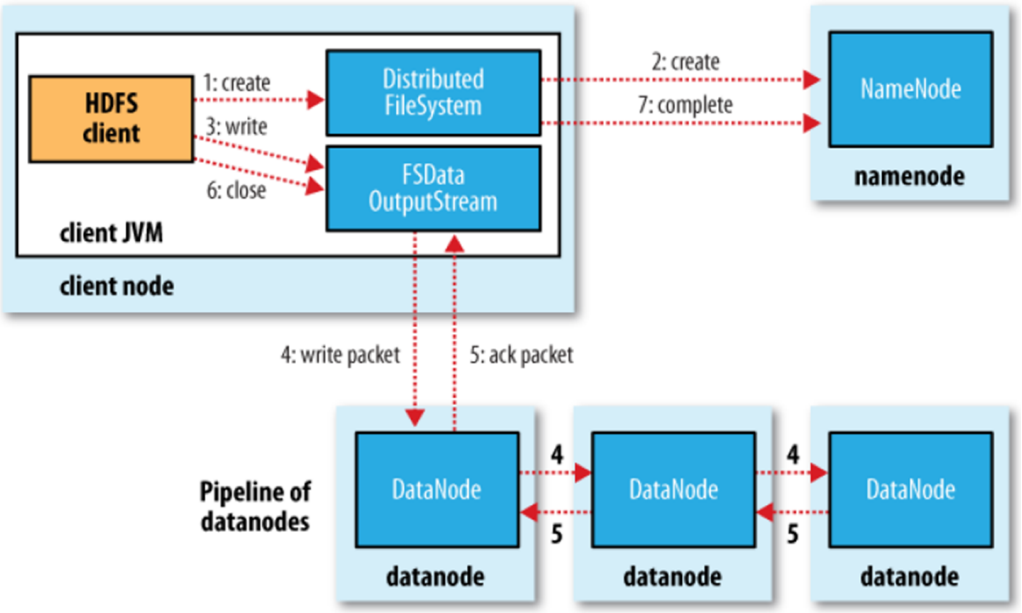
Anatomy of a File Read Request



- The client starts with calling *open()* on the HDFS.
- HDFS calls the namenode, using remote procedure calls (RPCs), to determine the locations of the first few blocks.
- For each block, the namenode returns the addresses of the datanodes that have a copy of that block. Datanodes are sorted according to their proximity to the client.
- The HDFS returns a *DFSInputStream* that supports file seeks to the client for it to read data.
- The client then calls read() on the *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 *DFSInputStream*.
- When the end of the block is reached, *DFSInputStream* will close the connection to the datanode and then find the best datanode for the next block.
- This happens transparently to the client (from the client point of view, it is reading a continuous stream.).
- Blocks are read in order. It also calls the namenode to retrieve datanode locations for next batch of blocks as needed.
- When the client has finished reading, it calls *close()* on the *DFSInputStream*.
- During reading, if an error occurs, it will try the next closest datanode for the same block. It will remember the failed datanode so that it won't retry the same datanode for a different block.
- The *DFSInputStream* also verifies checksums for the data transferred to it from the datanode. If a corrupted block is found, it attempts to read a replica of the block from another datanode; it also reports the corrupted block to the namenode.
- **Important aspect:** Client contacts datanode 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 bottleneck as the number of clients grew.

Anatomy of File Write Request



- The client creates the file by calling *create()* on HDFS.
- HDFS makes an RPC call to the namenode to create a new file in the filesystem's namespace with no blocks associated with it.
- The namenode checks to make sure file doesn't already exist and client has the permission to create a new file. HDFS returns a *DFSOutputStream* to the client so that client can start writing data.
- As the client writes data, the *DFSOutputStream* splits it into packets and they get written into an internal queue called *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 form a pipeline and if the replication level is 3, there will be 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.
- The second datanode in the pipeline stores each packet and forwards it to the third (and the last) datanode in the pipeline.
- The *DFSOutputStream* also maintains an internal queue, called *ack queue*, of packets that are waiting to be acknowledged by datanodes.
- A packet is removed from *ack queue* only when it has been acknowledged by all the datanodes in the pipeline.
- 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 *DFSOutputStream*.

File Delete Operation in HDFS

- When a file is deleted by a client, HDFS renames that file to a file in the */trash* directory for a configurable amount of time.
- A client can request for an undelete in this allowed time.
- After the specified time the file is deleted and the space is reclaimed.
- When the replication factor is reduced, the NameNode selects excess replicas that can be deleted.
- Next heartbeat transfers this information to the DataNode that clears the blocks for use.

Actions of JobTracker in MR1

- Client applications submit jobs to the Job tracker.
- The JobTracker talks to the NN to determine the location of the data.
- The JobTracker locates TaskTracker nodes with available slots at or near the data.
- The JobTracker submits the work to the chosen TaskTracker nodes.
- The TaskTracker nodes are monitored. If they do not submit heartbeat signals often enough, they are deemed to have failed and the work is scheduled on a different TaskTracker.
- A TaskTracker will notify the JobTracker when a task fails. The JobTracker decides what to do then: it may resubmit the job elsewhere, it may mark that specific record as something to avoid, and it may even blacklist the TaskTracker as unreliable.
- When the work is completed, the JobTracker updates its status.

Fensing in Hadoop 2

- It is vital for the correct operation of an HA cluster that only one of the NameNodes be Active at a time. Otherwise, the namespace state would quickly diverge between the two, risking data loss or other incorrect results.
- In order to ensure this property and prevent the so-called "split-brain scenario," the administrator must configure at least one fencing method for the shared storage.
- During a failover, if it cannot be verified that the previous Active node has relinquished its Active state, the fencing process is responsible for cutting off the previous Active's access to the shared edits storage.
- This prevents it from making any further edits to the namespace, allowing the new Active to safely proceed with failover.

Job History

Job history refers to the events and configuration for a completed MapReduce job. It is retained regardless of whether the job was successful, in an attempt to provide useful information for the user running a job.

Job history files are stored in HDFS by the MapReduce application master, in a directory set by the *mapreduce.jobhistory.done-dir* property.

Job history files are kept for one week before being deleted by the system.

The history log includes job, task, and attempt events, all of which are stored in a file in JSON format.

The history for a particular job may be viewed through the web UI for the job history server (which is linked to from the resource manager page) or via the command line using *mapred job -history* (which points at the job history file).

Scheduling in YARN

In an ideal world, the requests that a YARN application makes would be granted immediately.

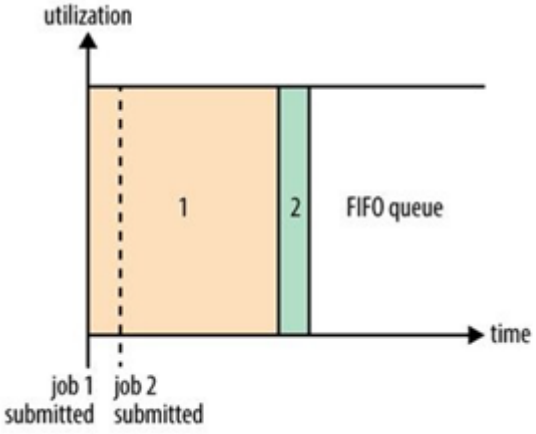
In the real world, however, resources are limited, and on a busy cluster, an application will often need to wait to have some of its requests fulfilled.

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.

FIFO (first in, first out)

The FIFO Scheduler places applications in a queue and runs them in the order of submission.

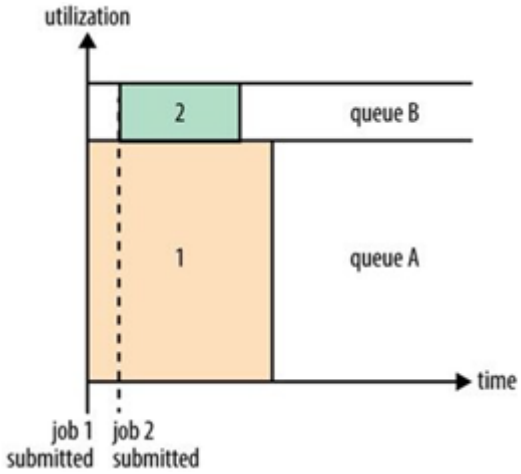
Requests for the first application in the queue are allocated first; once its requests have been satisfied, the next application in the queue is served, and so on.



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.



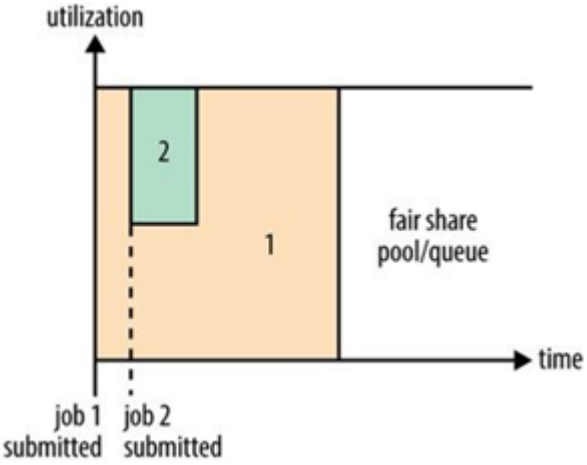
Fair Scheduler

— 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.



Uber Jobs

What the application master does is to decide how to run the tasks that make up the MapReduce job. If the job is small, the application master may choose to run the tasks in the same JVM as itself. This happens when it judges the overhead of allocating and running tasks in new containers outweighs the gain to be had in running them in parallel, compared to running them sequentially on one node.
Now, the questions could be raised as "What qualifies as a small job?"
By default, a small job is one that has less than 10 mappers, only one reducer, or an input size that is less than the size of one HDFS block.

YARN Configuration File

— The YARN configuration file is an XML file that contains properties.

— This file is placed in a well-known location on each host in the cluster and is used to configure the ResourceManager and NodeManager.

— By default, this file is named *yarn-site.xml*.

— It is required by the ResourceManager and NodeManager to run properly.

— YARN keeps track of two resources on the cluster, vcores and memory. The NodeManager on each host keeps track of the local host's resources, and the ResourceManager keeps track of the cluster's total.