CS4650 Topic 3:
HDFS - Hadoop Distributed Filesystem

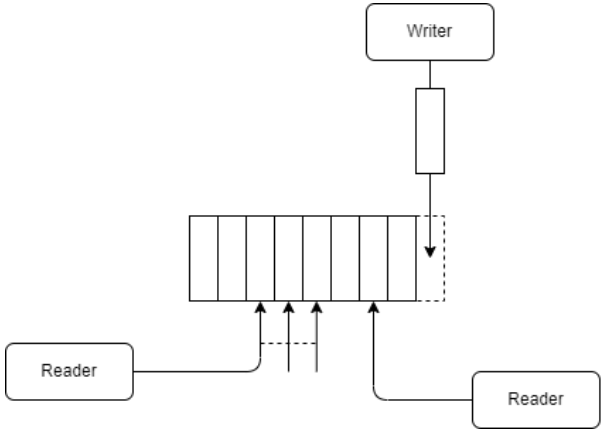
Hadoop Distributed Filesystem: HDFS

- HDFS is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware.
- Let's unpack this…

Very Large Files

- We typically mean files that are hundreds of megabytes, gigabytes or terabytes. Some are even petabytes.
- Not all files are that size, but files of this size can be handled by the system.

Streaming Data Access



- For many data processing operations, the most efficient pattern is write-once, read-many-times.
- Typically one source writes data to the file, appending data to the end of the file.
- Many tools can access this data to perform analysis. A usual analysis reads all of the data in the file, in order.
- Time to read the whole file (throughput) is more important than the latency to read the first record.

Commodity Hardware

- Hadoop doesn't require expensive, reliable hardware. It is designed to run on a cluster of commonly available hardware.
- The chance for failure of a node in the cluster is high, so HDFS is designed to work without noticeable interruption when such failures occur.

What About Small Data?

- While HDFS is designed to handle these large configurations (big data, cloud computing), it can also be run on your personal computer with more modest file sizes.
- In fact, you will be doing this for some assignments in this class!

Application Areas

- There are some application areas where HDFS is not a good fit:
  - Low-latency data access, where quick access to some piece of data is required.
  - Lots of small files. Each file in HDFS has some overhead, so if there are too many files, there is too much overhead. Millions of files is feasible, but billions are not. Hence, in many cases, multiple small files are merged into fewer larger files before being stored in HDFS.
  - Multiple writers. HDFS is designed to support a single writer of the data.
  - Arbitrary file modifications. The usual pattern is that data is only written to the file, it is not later modified or deleted.

Blocks

- HDFS, like every other file system, stores data in blocks of a fixed size. An HDFS file is composed of a sequence of blocks.
  - Typical disk block sizes are 512 bytes.
  - File system block sizes are typically a few kilobytes.
  - HDFS typically uses block sizes around 64 MB in size!
- HDFS blocks are not the same as disk blocks, however. These blocks can be on different computers in a cluster. So each HDFS 'block' is actually a 'file' on some disk drive.

Why Large Blocks?

- Disk seek time is around 10ms. This is time required for the disk drive to find the start of the file.
- Disk transfer rates are 100MB/s.
- Consequently, the disk could transfer 1 MB of data in the same time it takes to perform one seek.
- The typical use of an HDFS file is to scan the whole file. So scanning the file will involve transfer of all of the data, but also includes all of the seeks to find the blocks.
- The seek time is essentially wasted overhead, so to minimize the number of seeks, HDFS maximizes the size of the blocks.

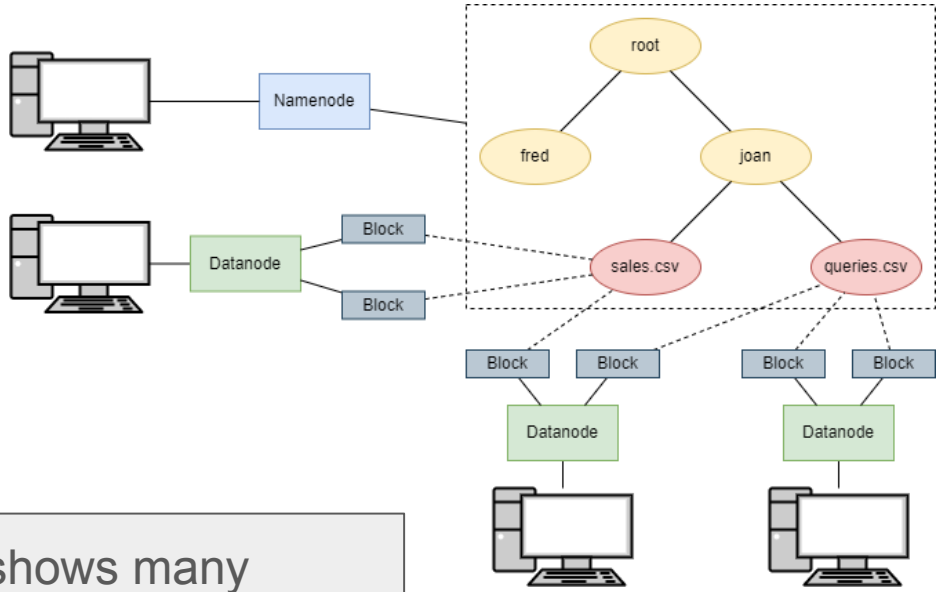
Block Benefits

- Because of the block abstraction, HDFS has some benefits:
  - Blocks in a file can be on different computers. Hence a complete file can be much larger than any one computer could hold.
  - Also, many data processing tasks can be performed in parallel, as we will see in the next topic. With a cluster of computers, each computer can process its block of the file, all at the same time. The results of the analysis can then be combined to produce the final result.
  - Blocks can be replicated, so the same data can exist on multiple computers. This gives redundancy, so if any computer is now, or if the block gets corrupted, other computers in the cluster can perform the correct calculation.

Distributed File System

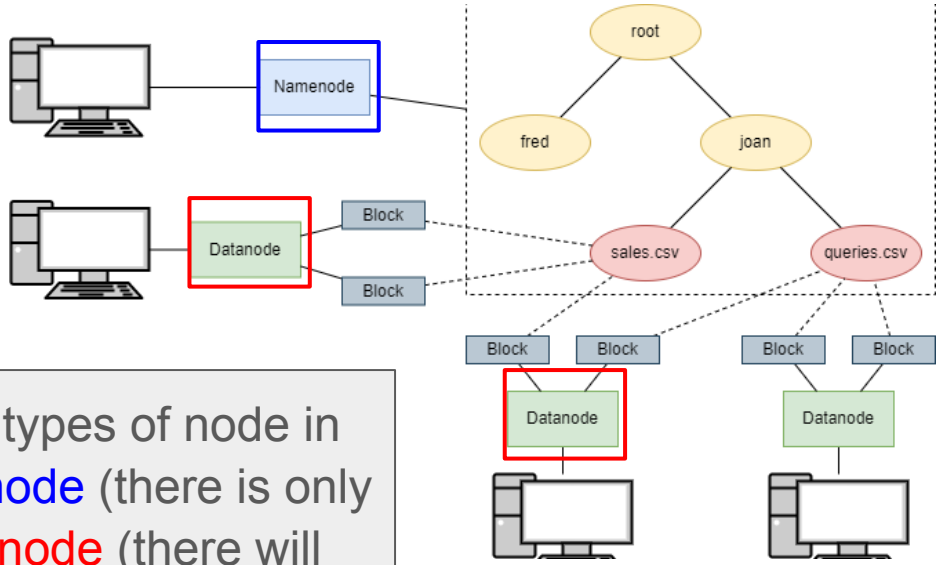
- Being a distributed file system, HDFS can run on one or more computers in a cluster.
- One of the computers runs the *Namenode* of the HDFS. The Namenode holds the information about the file system itself, the directory tree and the list of files.
- The other computers run a *Datanode* of the HDFS. A Datanode manages the HDFS Blocks that exist on that computer.
- For a smaller HDFS installation, one computer might run both the Namenode and one of the Datanodes.

HDFS Diagram



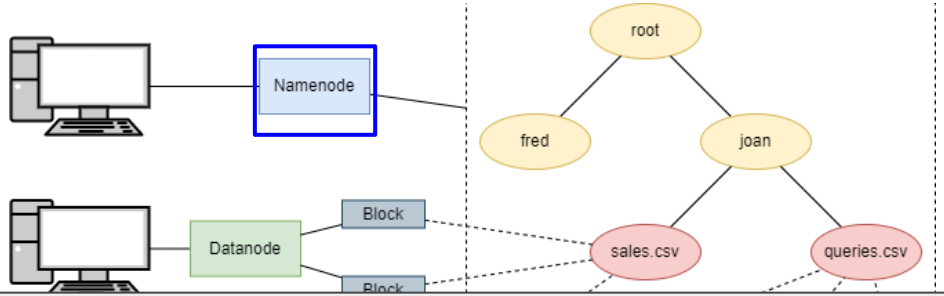
- This diagram shows many aspects of the HDFS.

HDFS Diagram

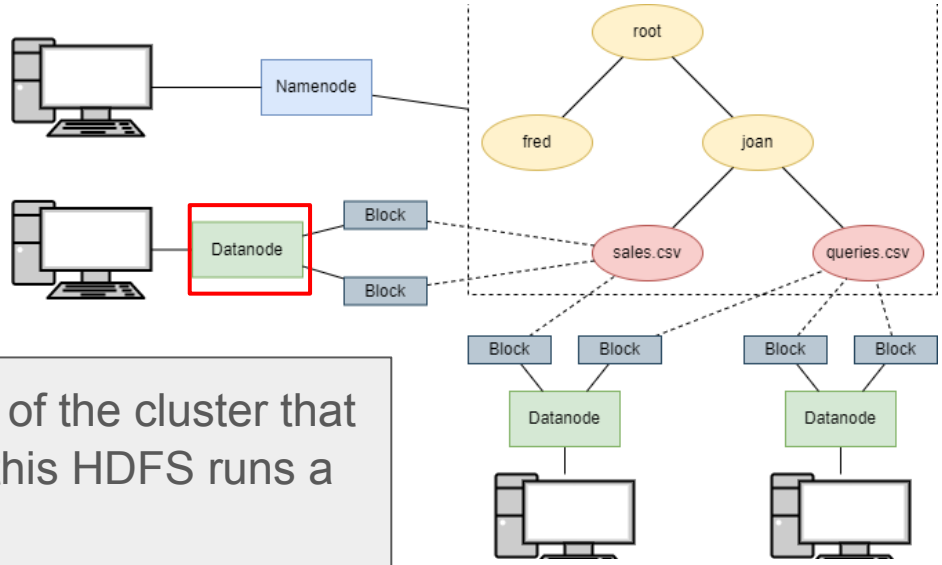


- There are two types of node in HDFS: Namenode (there is only one) and Datanode (there will probably be many)

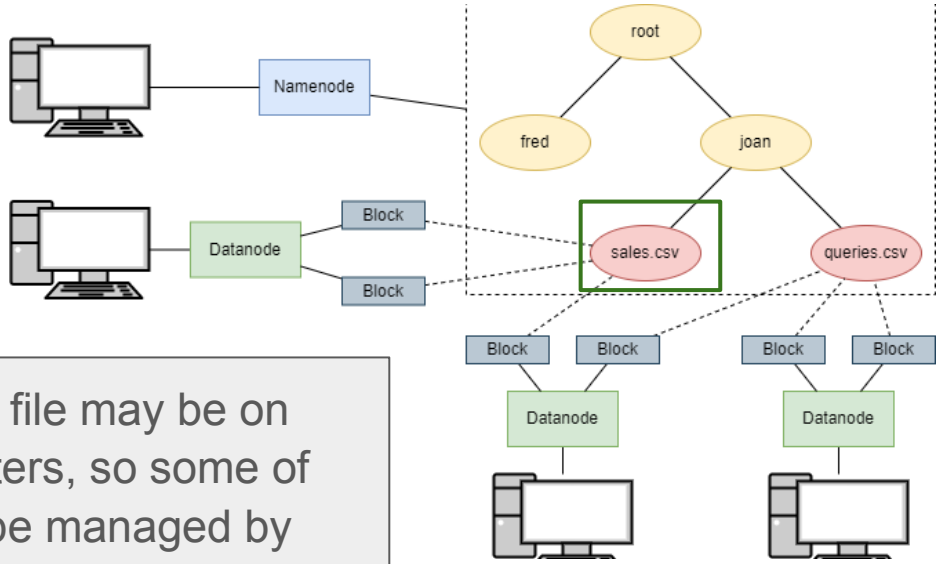
HDFS Diagram



- The Namenode contains the file system for HDFS. It contains a tree of directories, along with entries for each file. It also has a list of all of the Datanodes that make up this HDFS.
- The entry for a file lists all of the Blocks that make up the file.
- If the Namenode becomes lost or corrupted, the whole HDFS becomes unusable!



- Each computer of the cluster that has Blocks for this HDFS runs a Datanode.
- The Datanode manages all of the Blocks that are on this computer.



- The Blocks of a file may be on multiple computers, so some of the Blocks will be managed by one Datanode and other blocks may be managed by other Datanodes.

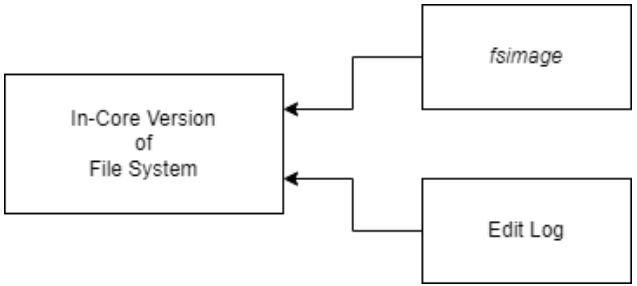
Under the Hood

- What we have talked about so far gives an overview of the architecture of HDFS.
- Next, we will be looking into more of the internal details of HDFS.
  - This peek is good experience, showing a Computer Science perspective on a complex, working system.
  - For many people who use HDFS, these details are not important -- Just use the system.
- Later, we will start discussing in the opposite direction, seeing the 'outside view' of HDFS. What commands can you run, how can you interface through an API, and so on.
- We won't talk too deeply about the management and tuning of an HDFS system.

Under the Hood

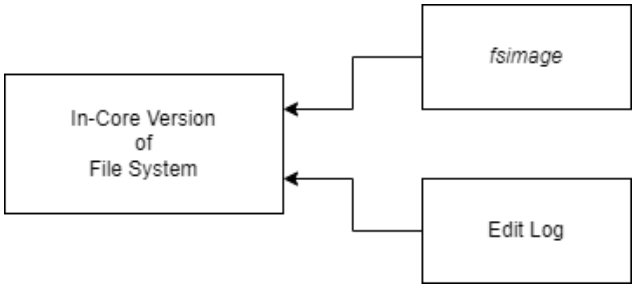
- Internal details:
  - Namenode Operation
  - Namenode Failure
  - Duplication of Nodes

Namenode Operation



- As the Namenode runs, it keeps an in-core view of the HDFS file system: what are the directories and subdirectories, what are the files, what blocks comprise the files, the name, permissions, and so on.
- All of this is in-core, so that the system runs fast.
- When changes are made, such as files are added, moved, renamed, deleted, and so on, or when new blocks are added to the file, this in-core structure is updated.

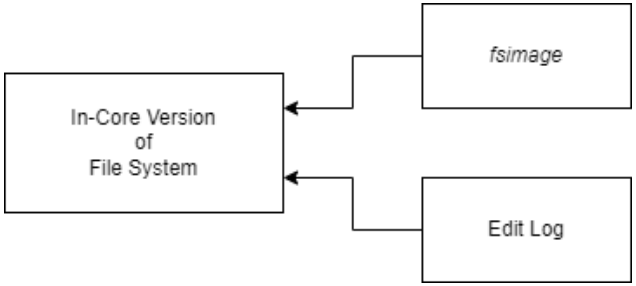
Namenode Operation



- However, what happens when this computer is shut down, then restarted? The Namenode will restart, but all of the dynamic data is lost!
- When the Namenode starts, it loads an image of the Namenode from a file known as the *fsimage* (the image of the file system). Initially this is a basic blank system.
- As edits are made to the file system, the editing operations are written to an *Edit Log* file. In fact, the information is written and flushed to the edit log *before* the changes are made to the in-core structure, and before the response code is given to the client application.

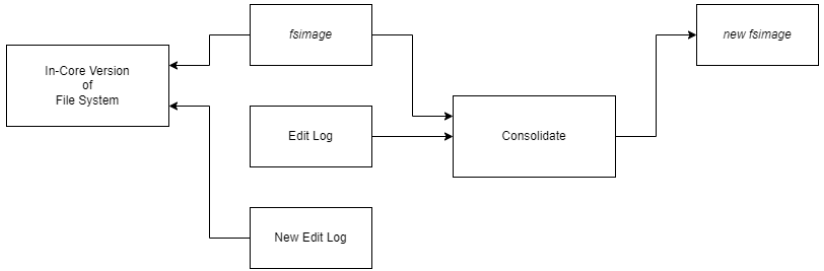
Namenode Operation

- On a restart, the Namenode loads the *fsimage*, then it runs through all of the entries in the edit log, making the changes to the file system image.
- Once that process is complete, the HDFS is back to where it was before the system was shut down.
- Note that the edit log only shows changes to the file system structure, it does not record changes to the files. As entries are written to the end of a file in HDFS, that data is written directly to the Block. The only time these edits would cause a change in the file system is if a new Block is added to a file.



Namenode Consolidation

- As time goes on, the edit log might get prohibitively large, so restarting the system could take a long time.
- HDFS has a mechanism to handle this situation. The steps are these:
  - The Namenode is told to 'roll over the edit log' (stop using the current edit log and start using a new one).
  - A new 'Namenode' is started that loads the original *fsimage*, then applies all of the changes from the old edit log.
  - This program then creates a new *fsimage* file.
  - The original Namenode is then informed of the new *fsimage*. It then deletes the original *fsimage* and edit log, and from now on uses the new *fsimage* and new edit log.



Namenode Failure

We now can consider what happens if there is a failure with the Namenode:

- The default approach is to start a new Namenode on a new server, with the *fsimage* and Edit Log from the old Namenode. This new Namenode rebuilds the file system, then contacts all of the Datanodes (to inform them that *it* is the new Namenode and to get all of the Block information). This can take 30 minutes or more.
- Another approach is to have a stand-by Namenode, which starts from the same *fsimage* and Edit Log. Every operation performed by the primary Namenode is also performed by the secondary. If the primary fails, the secondary is ready to step in. This can take a few seconds to a minute.

Duplication of Nodes

- To provide fault tolerance and to increase availability, HDFS typically *replicates* the blocks of a file, and the replication factor is usually 3.
- This means that the HDFS stores 3 identical copies of a file. So if a file takes 5 blocks, HDFS allocates 15 blocks for the file, 3 copies of the first block, 3 of the second, and so on.
- Whenever a write happens to a file, the write will occur to all three blocks at the end of the file.
- For a read operation, only one block needs to be read.

Duplication of Nodes

Usually the HDFS will store the three copies of the files on different Datanodes, for a couple of reasons:

- If the copies were all on the same Datanode, and that Datanode fails or is down temporarily, then none of those Blocks are reachable. By spreading the Blocks out, at least one copy of a Block should be available.
- Since write operations happen to all three Blocks, if these are all on the same machine, that write operation will take three times as long. However, by being on different machines, all three machines can be running in parallel.
- As we will see in a couple of Topics, by spreading this out, we also have redundancy in performing analysis.

Duplication of Nodes

- If a block becomes unavailable due to corruption or failure of a computer, HDFS can notice that essentially the replication count for that block is now two.

- It can then allocate a new Block on a different Datanode, then copy the data from one of the two existent copies.
- Once this is done, the replication count is again back up to three.
- Some applications may want a higher replication count for popular data. Requests can be performed on a subset of the Datanodes, which would increase the throughput.
- When we work on our laptops, we can set the replication count to 1, to limit the space taken on our computer.

More Info

- The book contains additional information about the internals of HDFS.
- This is interesting stuff… but maybe you are on overload.
- Hence, we will move on. But for those who are interested, you can dig deeper!

External View

We will briefly touch on two interfaces to HDFS:

- Command line interface
- Java interface

Command-Line Interface

- There are two usage models for the HDFS command-line interface:
  - Pseudo-Distributed mode (actually just running on one computer)
  - Cluster mode (running across multiple machines)
- When running on our own computers, we will use pseudo-distributed mode.
- There are two key settings for this mode:
  - fs.default.name = hdfs://localhost/
    This setting is a URI indicating where to find the filesystem. This value indicates that it is on your computer (using port 8020).
  - dfs.replication = 1
    This means to override the default 3-copies, where each block is replicated for redundancy. On your one computer, you just one one copy of each block.
- These values are described further in the book appendix.

Command-Line Interface

- The command line interface looks like this:

  % hadoop fs <command>
- There are commands to copy files from your local computer into HDFS, to copy files from HDFS to your local computer, to create directories, copy files, delete files, rename and move files.

Java Interface

- HDFS also has a FileSystem class (as well as a couple of more specialized classes). Using FileSystem, the Java code will be portable across file systems (so it can be run locally for tests or distributed for actual computations).
- We will look into the Java Interface in a couple of lectures, when we use HDFS and MapReduce to analyze some data!

Anatomy of a Read

1. The client opens the file by calling open() on the FileSystem object.
2. The FileSystem calls the namenode (using RPC) to determine the locations of the first few blocks of the file.
3. For info of each block, the namenode returns the address of the 'closest' datanode holding a copy of that block.
4. The FileSystem returns a FSDataInputStream, from which the client can read data.
5. The client then calls read() on this stream. That datanode returns the next values to the client.
6. When one block is exhausted, the FileSystem closes the connection to that datanode and opens a connection to the next datanode.

Anatomy of a Read

7. When the list of known blocks is exhausted, the namenode is asked for the next few blocks of the file.
8. If, during the reading, an error is encountered, the FileSystem will record that this block is not working, and will then try the next closest datanode that has a duplicate of this block.

- Because of this architecture, the network traffic is spread out.
  - All clients need to talk to the namenode, but they only need a little information.
  - The bulk of the traffic is with the datanodes, but these are spread out and are replicated.

Anatomy of a Write

- For this example, we will create a new file, write data to it, then close the file.

1. The client calls create() on the FileSystem object.
2. The FileSystem calls the namenode (using RPC) to create a new file in the HDFS directory tree (the in-core structure, while writing this information to the Edit Log). There are no blocks yet, but a new file is created.
3. The FileSystem returns a FSDataOutputStream, to which the client can write data.
4. As the client writes data, the FSDataOutputStream splits it into packets, which are written to the *data queue.*

Anatomy of a Write

5. Packets are removed from the data queue and written to the HDFS as follows:
    a. The namenode is asked for new blocks to store the data (typically 3 for clusters, 1 for local).
    b. The namenode picks three 'best' datanodes, which will form a pipeline.
    c. The packet is sent to the first of the datanodes, which stores the data in a new block, but then sends the packet to the second of the datanodes.
    d. The second datanode does the same, and the third datanode also stores the data.
    e. The client's program meanwhile moves the packet to the *ack queue*, where it waits for acknowledgements.
    f. As each datanode successfully processes the packet, it sends an *ack* (acknowledge) signal back to the source.
    g. When the source receives all three acks, it then discards the packet.

Anatomy of a Write

6. If any of the datanodes fail while processing the packet, the following actions are performed:
    a. The pipeline is closed.
    b. Any packets in the *ack queue* are moved back to the *data queue* (they need to be sent again).
    c. The current block on any good datanode are given a new identity, which is communicated to the namenode (so that the namenode can identify the failed datanodes).
    d. The failed datanode is removed from the pipeline.
    e. The remainder of the data packets are sent to the two remaining datanodes.
    f. The namenode will realize that this block is under-replicated, so it will arrange for the block to be duplicated from one of the good datanodes.
7. When the client is done, it calls close(). This then flushes all of the buffers and waits for the *ack*s.

Coherency Model

- Due to the way HDFS is written, data written to the last block of a file will not necessarily be seen when the file is read.
- When a file is created, it is immediately visible, even before any data is written to the file.
- Blocks are not visible until:
  - The block is completely full and the next block is started, or
  - hflush() is called by the writer, which forces all buffers to be synchronized to the datanodes.
- There is a performance overhead to call hflush().
- Consequently, there is a tradeoff of security/speed when considering how often to call hflush().
- Up to a block's worth of data will be lost if the client or system fails, unless hflush() is used.