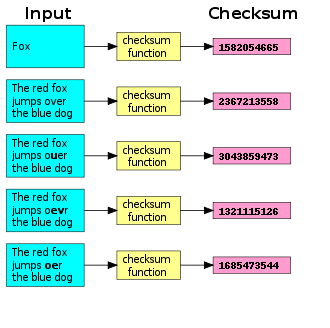
**Assignment 4.5**

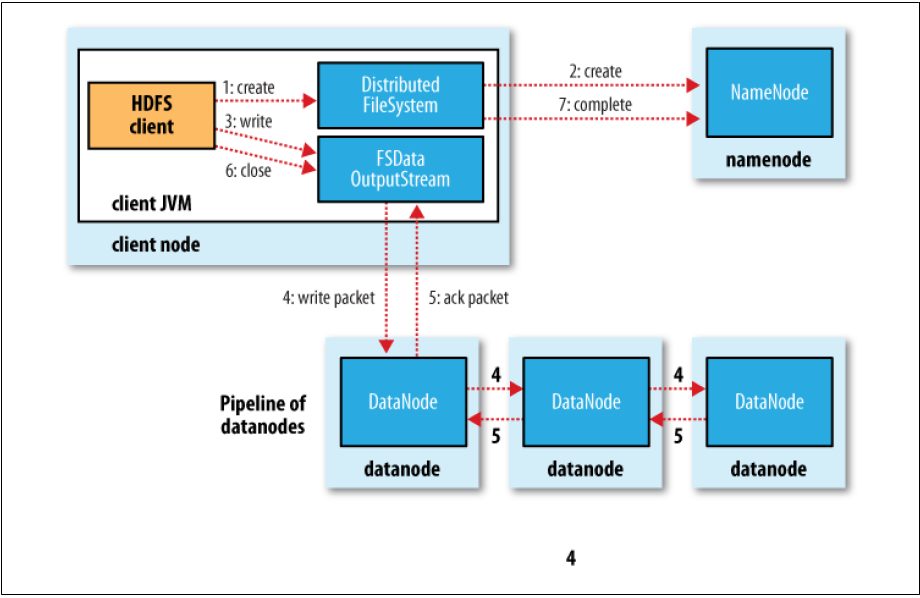
1. **Explain what is checksum and the importance of checksum and** **how Hadoop performs checksum.**

**Checksum:** A***checksum*** is a small-sized datum derived from a block of [digital data](https://en.wikipedia.org/wiki/Digital_data) for the purpose of [detecting errors](https://en.wikipedia.org/wiki/Error_detection) which may have been introduced during its [transmission](https://en.wikipedia.org/wiki/Telecommunication) or [storage](https://en.wikipedia.org/wiki/Computer_storage). It is usually applied to an installation file after it is received from the download server. By themselves, checksums are often used to verify data integrity but are not relied upon to verify data [authenticity](https://en.wikipedia.org/wiki/Authentication).



In Hadoop checksum works in following way:

1. The HDFS client software implements checksum checker. When a client creates an HDFS file, it computes a checksum of each block of the file and stores these checksums in a separate hidden file in the same HDFS namespace.
2. When a client retrieves file contents, it verifies that the data it received from each DataNode matches the checksum stored in the associated checksum file.
3. If not, then the client can opt to retrieve that block from another DataNode that has a replica of that block.
4. If checksum of another Data node block matches with checksum of hidden file, system will serve these data blocks.
5. **Explain the anatomy of file write to HDFS**



1. Client creates, calls create() on DistributedFileSystem.

2. DistributedFileSystem contacts namenode to create a new file in the filesystem’s namespace, with no blocks associated with it. 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 (in edits)

3. DistributedFileSystem returns an FSDataOutputStream for the client to start writing data. FSDataOutputStream uses DFSOutputStream, which handles communication with the datanodes and namenode.

4. Client signals write() method on FSDataOutputStream.

5. DFSOutputStream splits data into packets and writes it to an internal queue called the data queue.

• The data queue is consumed by the DataStreamer, which asks the namenode to give a location for the datanodes where blocks will be stored.

• The list of datanodes form a pipeline with a number of datanodes equals replication factor.

• The DataStreamer streams the packets to the first datanode in the pipeline.

• First datanode 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 datanode in the pipeline.

• The DFSOutputStream also maintains an internal queue 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.

• So there are two queues: data queue (containing packets that are to be written) and ack queue (containing packets that are yet to be acknowledged)

6. When the client has finished writing data, it calls close() on the stream. This flushes all the remaining packets to the datanode pipeline and waits for acknowledgments.

7. DistributedFileSystem contacts the namenode to signal that the file write activity is complete.

• The namenode already knows which blocks the file is made up of (because DataStreamer had asked for block allocations), so it only has to wait for blocks to be minimally replicated before returning successfully.

• Closing a file in HDFS performs an implicit hflush().

• After a successful return from hflush(), HDFS guarantees that the data written up to that point in the file has reached all the datanodes in the write pipeline and is visible to all new readers.

1. **Explain how HDFS handles failures during file write.**

1. The pipeline is closed and any packets in the ack queue are added to the front of the data queue.

2. The current block on the good datanodes is given a new identity, which is communicated to the namenode

3. The failed datanode is removed from the pipeline, and a new pipeline is constructed from the two good datanodes.

4. The remainder of the block’s data is written to the good datanodes in the pipeline.

5. The namenode notices that the block is under-replicated, and it arranges for a further replica to be created on another node.

6. As long as dfs.namenode.replication.min replicas (which defaults to 1) are written, the write will succeed.

7. The block will be asynchronously replicated across the cluster until its target replication factor is reached (dfs.replication, which defaults to 3).