Q1. Explain what is checksum and the importance of checksum and how Hadoop performs checksum?

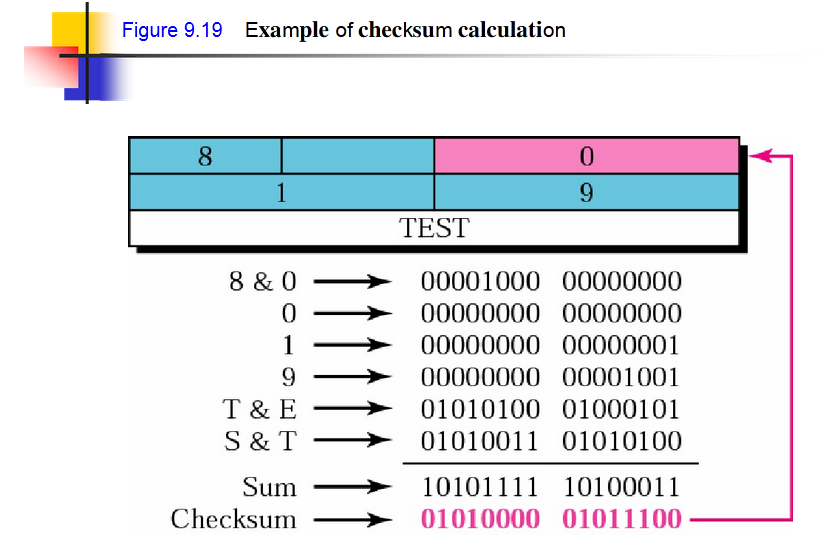
A1.

1) The checksum is a small-size datum from a block of digital data for the purpose of detecting errors which may have been introduced during its transmission or storage.

2) HDFS checksums all data written to it and by default verifies checksums when reading data.

3) A separate checksum is created for every dfs.bytes-per-checksum bytes of data.

4) The default is 512 bytes, and because a CRC-32C checksum is 4 bytes long. The storage overhead is less than 1%.



5) Datanodes are responsible for verifying the data they receive before storing the data and its checksum.

6) When the clients read data from datanodes, they verify checksums as well, comparing them with the ones stored at the datanodes.

7) -get command does the checksum verification during the data read.

8) -copyFromLocal doesn’t perform checksum during data read.

9)-ignoreCrcoption with the -get is equivalent to –copyToLocal command.

10) Disabling checksum verification is useful if we have a corrupt file that we want to inspect so that we can decide what to do with it.

Q2. Explain the anatomy of file write to HDFS?

A2

Anatomy of File Write to HDFS



Writing File to HDFS: Steps

1. The client sends create request on Distributed File System APIs. Distributed File System contacts name node to create a new file in the file system’s namespace, with no blocks associated with it. The name node 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 name node makes a record of the new file (in edits)

2. Distributed File System returns an FSDataOutputStream for the client to start writing data. FSDataOutputStream uses DFSOutputStream, which handles communication with the data nodes and name node.

3. Client signals write() method on FSDataOutputStream.

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

1. The data queue is consumed by the DataStreamer, which asks the name node to give a location for the data nodes where blocks will be stored.
2. The list of data nodes form a pipeline with a number of data nodes equals replication factor.
3. The DataStreamerstreams the packets to the first data node in the pipeline.
4. First data node stores each packet and forwards it to the second data node in the pipeline.
5. Similarly, the second data node stores the packet and forwards it to the third data node in the pipeline.
6. The DFSOutputStream also maintains an internal queue called the ackqueue.
7. A packet is removed from the ackqueue only when it has been acknowledged by all the data nodes in the pipeline.
8. So there are two queues: data queue (containing packets that are to be written) and ackqueue (containing packets that are yet to be acknowledged)

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

6. DistributedFileSystem contacts the name node to signal that the file write activity is complete.

1. The name node 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.
2. Closing a file in HDFS performs an implicit hflush().
3. After a successful return from hflush(), HDFS guarantees that the data written up to that point in the file has reached all the Data nodes in the write pipeline and is visible to all new readers.

Q3. Explain how HDFS handles failures during file write?

A3

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

2. The current block on the good data nodes is given a new identity, which is communicated to the name node

3. The failed data node 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 data nodes in the pipeline.

5. The name node 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.minreplicas (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).