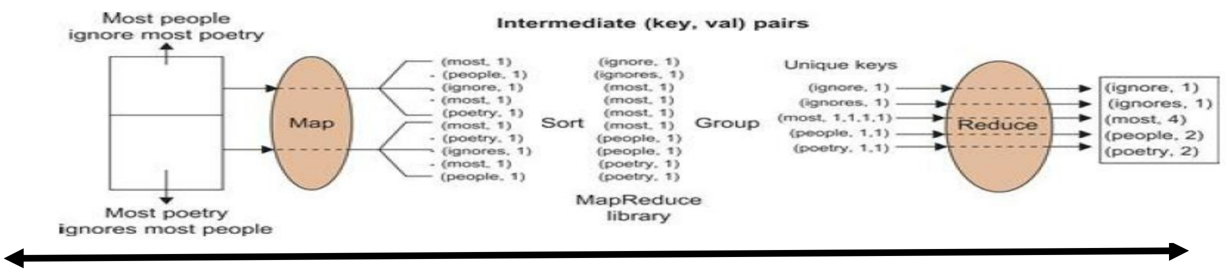
- In this case, the *Map* function simultaneously produces a number ofintermediate (key, value) pairs for each line of content so that each word is theintermediate key with *1* as its intermediate value; for example, *(ignore, 1)*.
- Then theMapReduce library collects all the generated intermediate (key, value) pairs and sortsthem to group the *1*'s for identical words; for example, *(people, [1,1])*.
- Groups are thensent to the *Reduce* function in parallel so that it can sum up the *1* values for each wordand generate the actual number of occurrence for each word in the file; for example,*(people, 2)*.

THE DATA FLOW OF THE WORD COUNT PROBLEM



4.7 DESIGN OF HADOOP FILE SYSTEM, HDFS CONCEPTS, COMMAND LINE AND JAVA INTERFACE

Part-B	**1.Elaborate HDFS concepts with Suitable Illustrations(AU/April/May 2017)** **2.HDFS is fault Tolerant.Is it True?Justify Answer (AU/Nov/Dec 2017)**

- A disk has a block size, which is the minimum amount of data that it can read or write.
- File system blocks are typically a few kilobytes in size, while disk blocks are normally 512 bytes.
- HDFS has the concept of a block, but it is a much larger unit—64 MB by default.File0sin HDFS are broken into block-sized chunks, which are stored as independent units.
- Unlike a file system for a single disk, a file in HDFS that is smaller than a single block does not occupy a full block's worth of underlying storage.
- The storage subsystem deals with blocks, simplifying storage management and eliminating metadata concerns

NAME NODES AND DATANODES

- An HDFS cluster has two types of node operating in a master-worker 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.
- 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, since this information is reconstructed from datanodes when the system starts.
- A *client* accesses the filesystem on behalf of the user by communicating with the namenode and datanodes.
- Datanodes are the workhorses of the filesystem.
- Hadoop can be configured so that the namenode writes its persistent state to multiple filesystems.
- The usual configuration choice is to write to local disk as well as a remote NFS mount.
- It is also possible to run a *secondary namenode*, which despite its name does not act as a namenode.

- Its main role is to periodically merge the namespace image with the edit log to prevent the edit log from becoming too large.
- The secondary namenode usually runs on a separate physical machine, since it requires plenty of CPU and as much memory as the namenode to perform the merge.
- It keeps a copy of the merged namespace image, which can be used in the event of the namenode failing.

HDFS FEDERATION

- The namenode keeps a reference to every file and block in the filesystem in memory, which means that on very large clusters with many files, memory becomes the limiting factor for scaling.
- HDFS Federation, introduced in the 0.23 release series, allows a cluster to scale by adding namenodes, each of which manages a portion of the filesystem namespace. For example, one namenode might manage all the files rooted under */user*, say, and a second Namenode might handle files under */share*.
- Under federation, each namenode manages a *namespace volume*, which is made up of the metadata for the namespace, and a *block pool* containing all the blocks for the files in the namespace.

HDFS HIGH-AVAILABILITY

- The combination of replicating namenode metadata on multiple filesystems, and using
- the secondary namenode to create checkpoints protects against data loss, but does not provide high-availability of the filesystem.
- The namenode is still a *single point of failure* (SPOF), since if it did fail, all clients—including MapReduce jobs—would be unable to read, write, or list files, because the namenode is the sole repository of the metadata and the file-to-block mapping.
- In such an event the whole Hadoop system would effectively be out of service until a new namenode could be brought online.
- In the event of the failure of the active namenode, the standby takes over its duties to continue servicing client requests without a significant interruption.

A few architectural changes are needed to allow this to happen:

- The namenodes must use highly-available shared storage to share the edit log.
- When a standby namenode comes up it reads up to the end of the shared edit log to synchronize its state with the active namenode, and then continues to read new entries as they are written by the active namenode.
- Datanodes must send block reports to both namenodes since the block mappings are stored in a namenode's memory, and not on disk.
- Clients must be configured to handle namenode failover, which uses a mechanism that is transparent to users.

FAILOVER AND FENCING

- The transition from the active namenode to the standby is managed by a new entity in the system called the *failover controller*.
- Failover controllers are pluggable, but the first implementation uses ZooKeeper to ensure that only one namenode is active.
- Each namenode runs a lightweight failover controller process whose job it is to monitor its namenode for failures and trigger a failover should a namenode fail.
- Failover may also be initiated manually by an administrator, in the case of routine maintenance, for example.
- The HA implementation goes to great lengths to ensure that the previously active namenode is prevented from doing any damage and causing corruption—a method known as *fencing*.

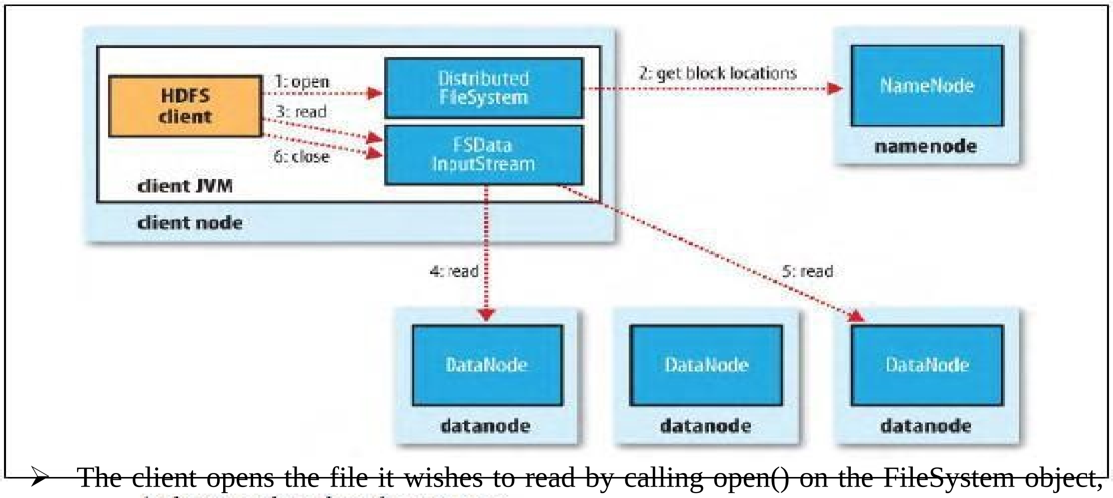
- ➢ The system employs a range of fencing mechanisms, including killing the namenode’s process, revoking its access to the shared storage directory, and disabling its network port via a remote management command.

4.8 DATAFLOW OF FILE READ & FILE WRITE.

Part-B	1.Illustrate data Flow in HDFS during File Read/Write Operations with Suitable Diagram(AU/Nov/Dec 2017)

DATA FLOW OF FILE READ

- ➢ To get an idea of how data flows between the client interacting with HDFS, the namenode and the datanode, consider the below diagram, which shows the main sequence of events when reading a file.



A client reading data from HDFS

- ➢ 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 RPC, to determine the locations of the blocks for 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), then it will read from the local datanode.
- ➢ The DistributedFileSystem returns a FSDataInputStream 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).