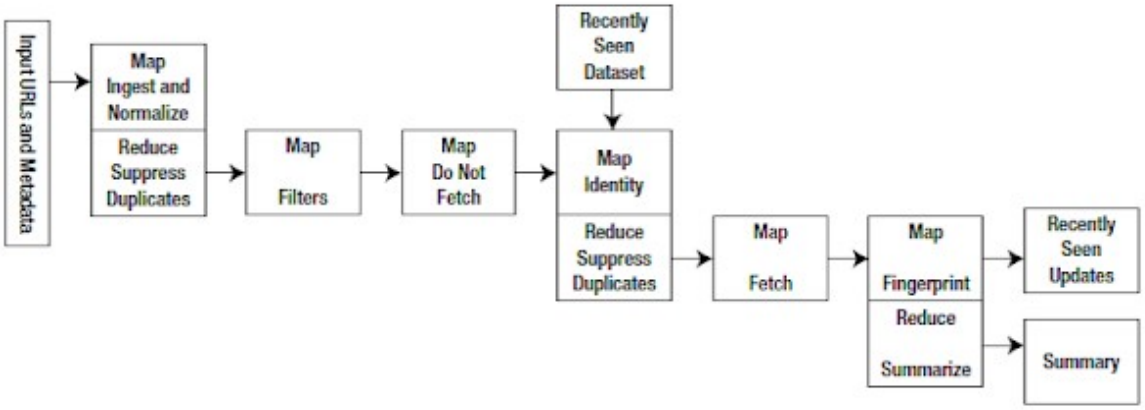
- **Pig** is a high-level data-flow language and execution framework for parallel computation. It is built on top of Hadoop Core.
- **ZooKeeper** is a highly available and reliable coordination system. Distributed applications use ZooKeeper to store and mediate updates for critical shared state.
- **Hive** is a data warehouse infrastructure built on Hadoop Core that provides data summarization, adhoc querying and analysis of datasets.

The Hadoop Core project provides the basic services for building a cloud computing environment with commodity hardware, and the APIs for developing software that will run on that cloud.
The two fundamental pieces of Hadoop Core are the

- **MapReduce framework, the cloud computing environment, and**
- **Hadoop Distributed File System (HDFS).**

The **Hadoop Core MapReduce framework** requires a shared file system. This shared file system does not need to be a system-level file system, as long as there is a distributed file system plug-in available to the framework.

The **Hadoop Core framework comes with plug-ins for HDFS**, CloudStore, and S3. Users are also free to use any distributed file system that is visible as a system-mounted file system, such as Network File System (NFS), Global File System (GFS), or Lustre.



The Hadoop Distributed File System (HDFS)MapReduce environment provides the user with a sophisticated framework to manage the execution of map and reduce tasks across a cluster of machines. The user is required to tell the framework the following:

- The location(s) in the distributed file system of the job input
- The location(s) in the distributed file system for the job output
- The input format
- The output format
- The class containing the map function
- Optionally. the class containing the reduce function
- The JAR file(s) containing the map and reduce functions and any support classes

MAPREDUCE, INPUT SPLITTING, MAP AND REDUCE FUNCTIONS, SPECIFYING INPUT AND OUTPUT PARAMETERS, CONFIGURING AND RUNNING A JOB

Part-B	1.Discuss MAPREDUCE with Suitable Diagram(AU/April/May2017)

MapReduce is a programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster.

- A MapReduce program is composed of a **Map()** procedure that performs filtering and sorting (such as sorting students by first name into queues, one queue for each name)
- **Reduce()** procedure that performs a summary operation (such as counting the number of students in each queue, yielding name frequencies).
- The **"MapReduce System"** (also called "infrastructure" or "framework") orchestrates the processing by marshalling the distributed servers, running the various tasks in parallel, managing all communications and data transfers between the various parts of the system, and providing for redundancy and fault tolerance.
- The core concept of MapReduce in Hadoop is that input may be split into logical chunks, and each chunk may be initially processed independently, by a map task. The results of these individual processing chunks can be physically partitioned into distinct sets, which are then sorted. Each sorted chunk is passed to a reduce task.

INPUT SPLITTING

- A map task may run on any compute node in the cluster, and multiple map tasks may be running in parallel across the cluster. The map task is responsible for transforming the input records into key/value pairs. The output of all of the maps will be partitioned, and each partition will be sorted. There will be one partition for each reduce task. Each partition's sorted keys and the values associated with the keys are then processed by the reduce task. There may be multiple reduce tasks running in parallel on the cluster.
- The application developer needs to provide only four items to the Hadoop framework: the class that will read the input records and transform them into one key/value pair per record, a map method, a reduce method, and a class that will transform the key/value pairs that the reduce method outputs into output records.
- MapReduce application was a specialized web crawler. This crawler received as input large sets of media URLs that were to have their content fetched and processed. The media items were large, and fetching them had a significant cost in time and resources.

The job had several steps:

1. Ingest the URLs and their associated metadata.
2. Normalize the URLs.
3. Eliminate duplicate URLs.
4. Filter the URLs against a set of exclusion and inclusion filters.
5. Filter the URLs against a do not fetch list.
6. Filter the URLs against a recently seen set.
7. Fetch the URLs.
8. Fingerprint the content items.9. Update the recently seen set.
10. Prepare the work list for the next application.

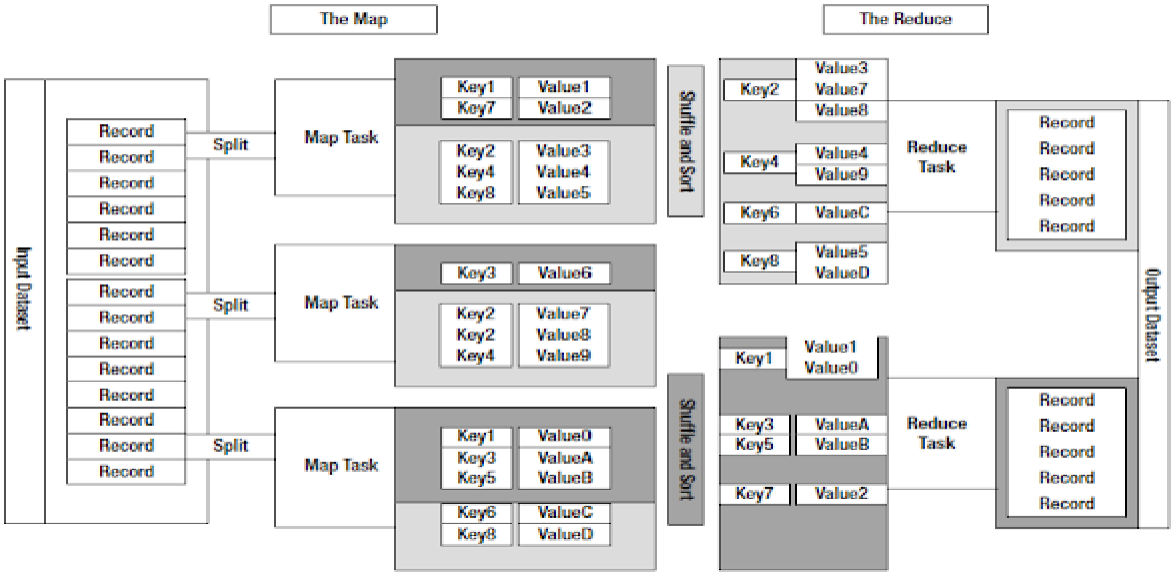


Figure: ***The MapReduce model***

MAP REDUCE DATA FLOW:

- The topmost layer of Hadoop is the MapReduce engine that manages the data flow and control flow of MapReduce jobs over distributed computing systems.
- Similar to HDFS, theMapReduce engine also has a master/slave architecture consisting of a single JobTrackeras the master and a number of TaskTrackers as the slaves (workers).
- The JobTracker manages the MapReduce job over a cluster and is responsible for monitoring jobs and assigning tasks to TaskTrackers.
- The Task Tracker manages the execution of the mapand/or reduce tasks on a single computation node in the cluster.
- Each Task Tracker node has a number of simultaneous execution slots, each executing either a map or a reduce task.
- Slots are defined as the number of simultaneous threads supported by CPUs of the TaskTracker node.

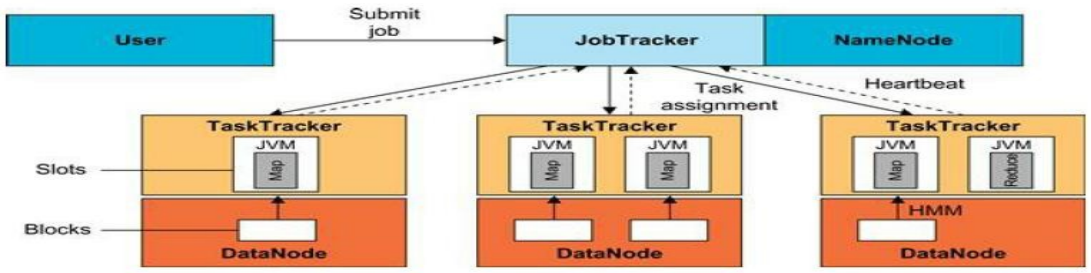


Fig: Data flow in running a MapReduce

- Three components contribute in running a job in this system: a user node, a JobTracker, and several TaskTrackers.
- The data flow starts by calling the *runJob(conf)* function inside a user program running on the user node, in which *conf*is an object containing some tuning parameters for the MapReduce framework and HDFS.

- The *runJob(conf)* function and *conf*are comparable to the *MapReduce(Spec, &Results)* function and *Spec* in the first implementation of MapReduce by Google
- **Job Submission** Each job is submitted from a user node to the JobTracker node that might be situated in a different node within the cluster through the following procedure:
- A user node asks for a new job ID from the JobTracker and computes input file splits.
- The user node copies some resources, such as the job's JAR file, configuration file,and computed input splits, to the JobTracker's file system.
- The user node submits the job to the JobTracker by calling the *submitJob()* function.
- **Task assignment** The JobTracker creates one map task for each computed input split by the user node and assigns the map tasks to the execution slots of the TaskTrackers.
- The overall structure of a user's program containing the Map, Reduce, and the Main functions is given below.
- The Map and Reduce are two major subroutines.
- They will be called to implement the desired function performed in the main program.

Map Function (… .)
{
… …
}

Reduce Function (… .)
{
… …
}
Main Function (… .)
{
Initialize **Spec** object
… …
MapReduce(Spec, & Results)
}

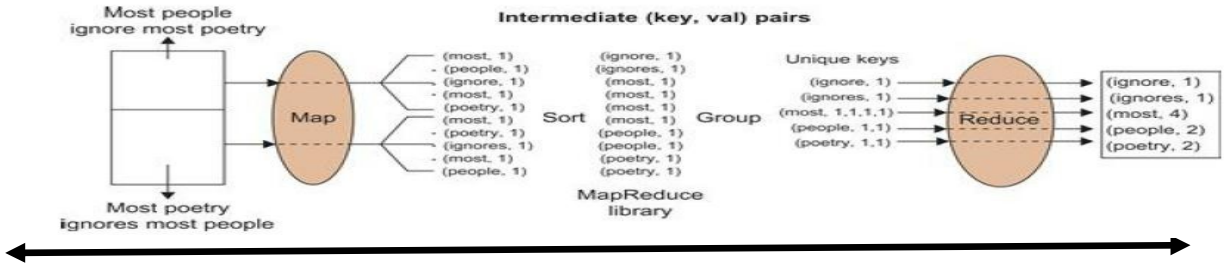
To solve map reduce:
Problem 1: Counting the number of occurrences of each word in a collection of documents
Solution: unique "key": each word, intermediate "value": number of occurrences
Problem 2: Counting the number of occurrences of words having the same size, or the same number of letters, in a collection of documents
Solution: unique "key": each word, intermediate "value": size of the word
Problem 3: Counting the number of occurrences of anagrams in a collection of documents. Anagrams are words with the same set of letters but in a different order (e.g., the words "listen" and "silent").
Solution: unique "key": alphabetically sorted sequence of letters for each word (e.g.,"eilnst"), intermediate "value": number of occurrences

Eg: word count

- MapReduce problems, namely word count, to count the number of occurrences of each
- word in a collection of documents is presented here.
- For a simple input file containing only two lines as follows:
- "most people ignore most poetry" and
- "most poetry ignores most people."

- 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.