SNS COLLEGE OF TECHNOLOGY

COIMBATORE - 35

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

19CSE310- GRID AND CLOUD COMPUTING

Unit IV – Programming Model

Topic : Introduction to Hadoop Framework

INTRODUCTION TO HADOOP FRAMEWORK

Hadoop is the Apache Software Foundation top-level project →various Hadoop subprojects that graduated from the Apache Incubator.
The Hadoop project provides and supports the development of open source software that supplies a framework for the development of highly scalable distributed computing applications.
The Hadoop framework handles the processing details, leaving developers free to focus on application logic

The Apache Hadoop project includes

- Hadoop Core, our flagship sub-project, provides a distributed filesystem (HDFS) and support for the MapReduce distributed computing metaphor.

- HBase builds on Hadoop Core to provide a scalable, distributed database

- 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.

INTRODUCTION TO HADOOP FRAMEWORK

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. The Hadoop Core framework comes with plug-ins for HDFS, CloudStore, and S3.

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

Input
Input
Input
Input

Map_1 | Map_2 | Map_3 | Map_4

Shuffle

$Reduce_1$ | $Reduce_2$ | $Reduce_3$ | $Reduce_4$

Output
Output
Output
Output

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

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")

- ❖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,
- ❖Poviding 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.

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

Input

Record
Record
Record
Record
Record
Record
Record
Record
Record
Record
Record
Record
Record
Record
Record
Record

Map Task (Key, Value) → Shuffle/Sort → Reduce → Output: Record, Record, Record, Record, Record

Map Task (Key, Value) → Shuffle/Sort → Reduce → Output: Record, Record, Record

Map Task (Key, Value) → Shuffle/Sort → Reduce → Output: Record, Record

Output

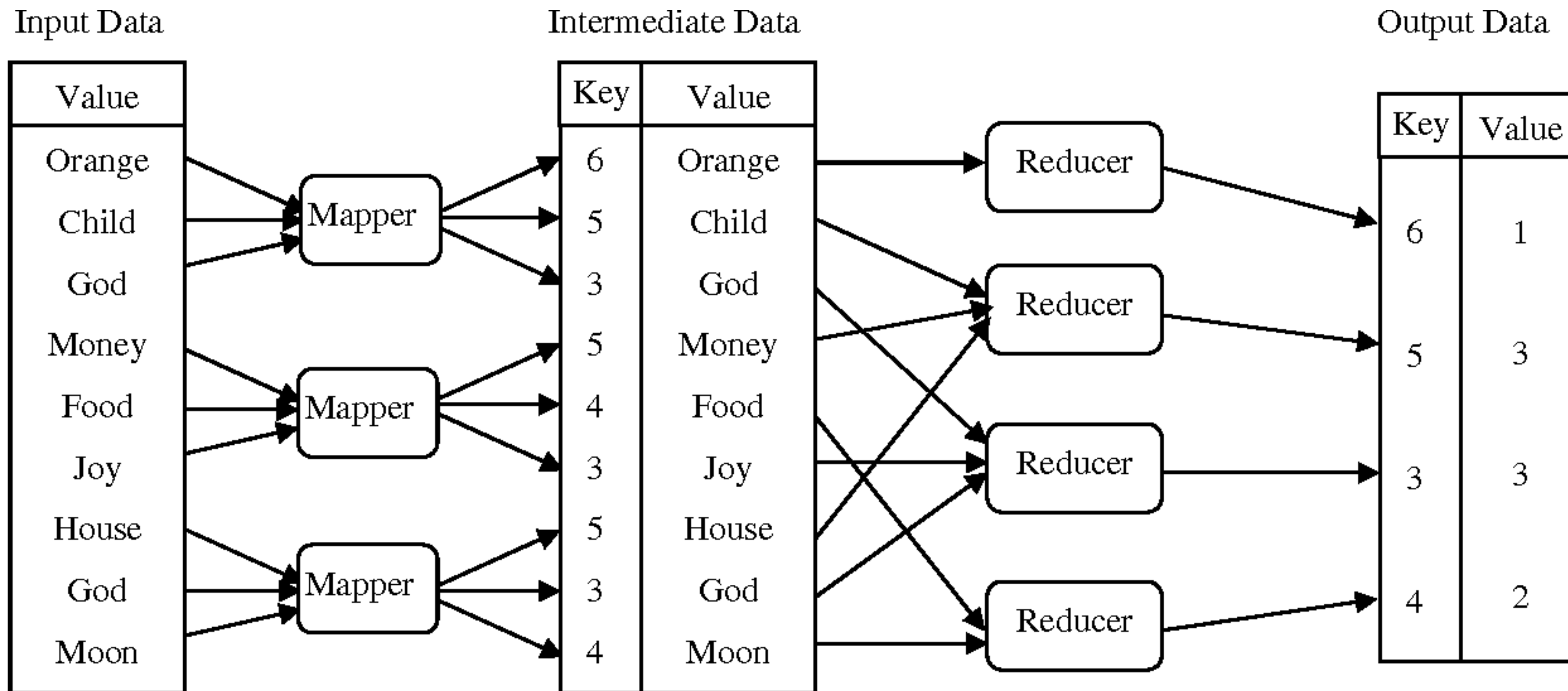

Fig. 2. Examples of MapReduce process

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

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

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 JobTracker as 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 map and/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.

MAP REDUCE DATA FLOW:

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

MAP REDUCE DATA FLOW:

➢ 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 confis an object containing some tuning parameters for the MapReduce framework and HDFS.

➢ The *runJob(conf) function and confare 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.

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

Job Tracker
Workload Monitor (WM)
Slot Assigner (SA)
Scheduler

(1) Estimate the present workloads

(2) Decide the best slot assignment of each node

(3) Assign task to slave nodes

Users

Submit a batch of jobs

Status Report

Task Assignment

Task Tracker
Task Manager
Slots
Map Reduce
Map Reduce

Task Tracker
Task Manager
Slots
Map Reduce
Map Reduce

...

Task Tracker
Task Manager
Slots
Map Reduce
Map Reduce

(4) Monitor the task execution and the slot occupation situation

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

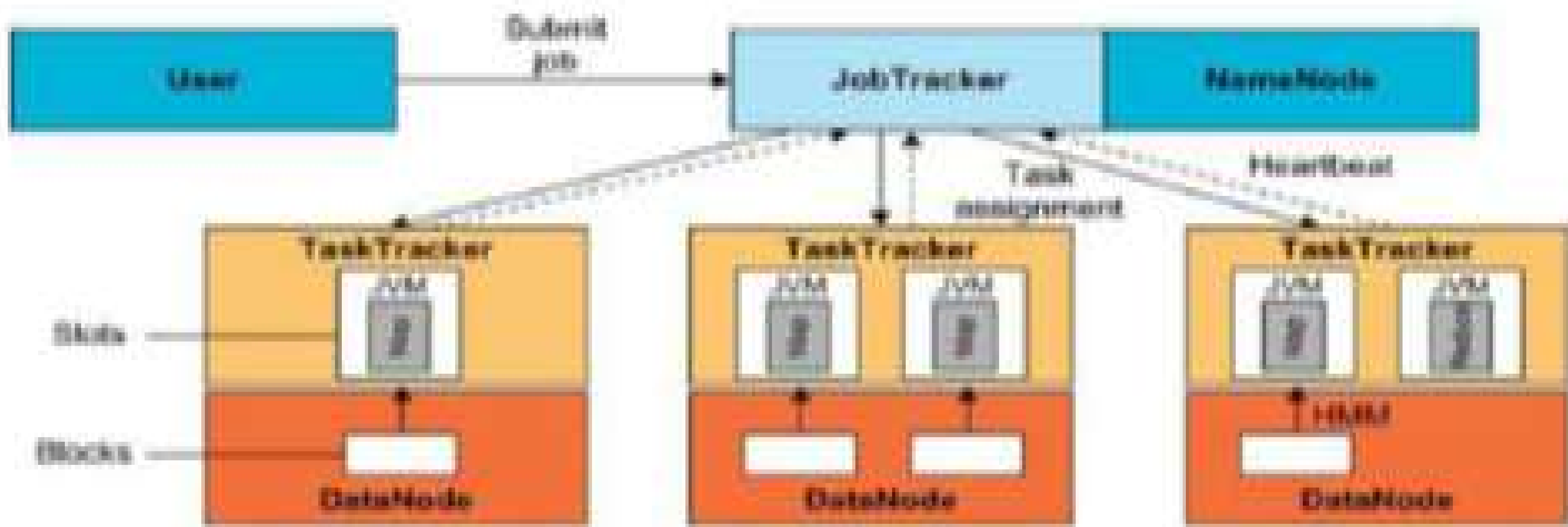

Fig: Data flow in running a MapReduce

Eg: word count

Input

Bus Car Train
Train Plane Car
Bus Bus Plane

Bus Car Train

Train Plane Car

Bus Bus Plane

Bus 1
Car 1
Train 1

Train 1
Plane 1
Car 1

Bus 2
Plane 1

BUS 2
BUS 1

CAR 1
CAR 1

TRAIN 1
TRAIN 1

PLANE 1
PLANE 1

BUS 3

CAR 2

TRAIN 2

PLANE 2

Output

BUS 3
CAR 2
TRAIN 2
PLANE 2

Splitting | Mapping | Intermediate Splitting | Reducing | Combining

Fig. WorkFlow of MapReducing

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

➢ 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. Files in 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.*

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.*

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

DATAFLOW OF FILE READ & FILE WRITE.

DATA FLOW OF FILE WRITE

HDFS client
1: create
3: write
6: close
Distributed FileSystem
FSData OutputStream
client JVM
client node
2: create
7: complete
NameNode
namenode
4: write packet
5: ack packet
Pipeline of datanodes
DataNode
datanode
4
5
DataNode
datanode
4
5
DataNode
datanode

DATA FLOW OF FILE READ

HDFS client
1: open
3: read
6: close
Distributed FileSystem
FSData InputStream
client JVM
client node
2: get block locations
NameNode
namenode
4: read
5: read
DataNode
datanode
DataNode
datanode
DataNode
datanode

Figure 3-2. A client reading data from HDFS