Hadoop Library from Apache

Cloud Computing

Introduction

Hadoop is an open source implementation of MapReduce coded and released in Java by Apache.

The Hadoop implementation of MapReduce uses the Hadoop Distributed File System (HDFS).

The Hadoop core is divided into two fundamental layers: the MapReduce engine and HDFS.

The MapReduce engine is the computation engine running on top of HDFS as its data storage manager

HDFS Architecture

HDFS is a distributed file system inspired by GFS that organizes files and stores their data on a distributed computing system.

HDFS has a master/slave architecture containing a single NameNode as the master and a number of DataNodes as workers (slaves).

To store a file in this architecture, HDFS splits the file into fixed-size blocks (e.g., 64 MB) and stores them on workers (DataNodes).

The mapping of blocks to DataNodes is determined by the NameNode.

The NameNode (master) also manages the file system's metadata and namespace

In such systems, the namespace is the area maintaining the metadata, and metadata refers to all the information stored by a file system that is needed for overall management of all files.

For example, NameNode in the metadata stores all information regarding the location of input splits/blocks in all DataNodes.

Each DataNode, usually one per node in a cluster, manages the storage attached to the node.

Each DataNode is responsible for storing and retrieving its file blocks.

HDFS Features

Distributed file systems have special requirements, such as performance, scalability, concurrency control, fault tolerance, and security requirements, to operate efficiently.

HDFS is not a general-purpose file system, as it only executes specific types of applications, it does not need all the requirements of a general distributed file system.

- HDFS Fault Tolerance
- HDFS High-Throughput Access to Large Data Sets (Files)

HDFS Fault Tolerance

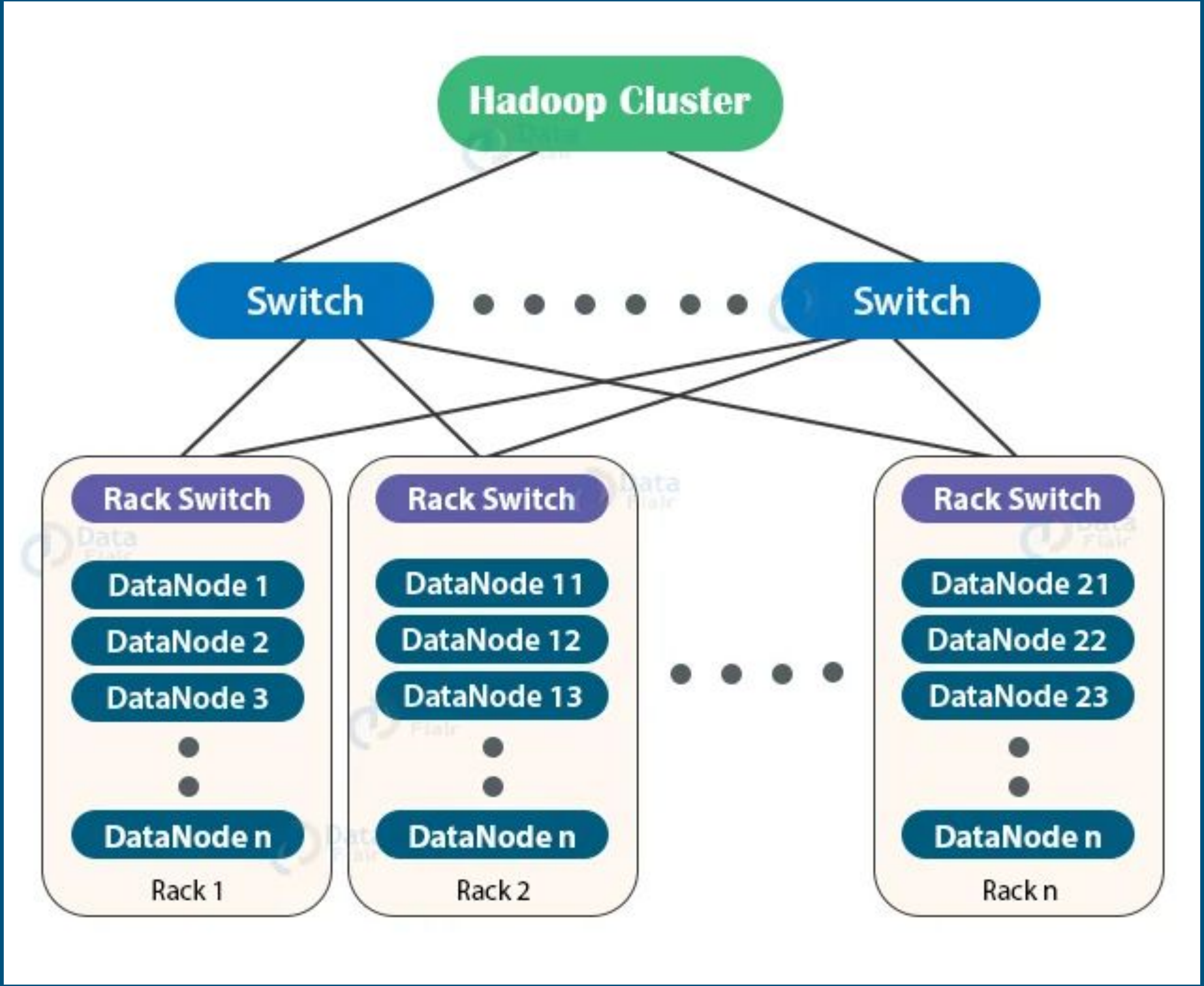
Hadoop is designed to be deployed on low-cost hardware by default, a hardware failure in this system is considered to be common rather than an exception.

Block replication

To reliably store data in HDFS, file blocks are replicated in this system.

HDFS stores a file as a set of blocks and each block is replicated and distributed across the whole cluster.

The replication factor is set by the user and is three by default.


Hadoop Cluster
Switch
Switch
Rack Switch
DataNode 1
DataNode 2
DataNode 3
DataNode n
Rack 1
Rack Switch
DataNode 11
DataNode 12
DataNode 13
DataNode n
Rack 2
Rack Switch
DataNode 21
DataNode 22
DataNode 23
DataNode n
Rack n

Replica placement

Storing replicas on different nodes (DataNodes) located in different racks across the whole cluster provides more reliability

It is sometimes ignored as the cost of communication between two nodes in different racks is relatively high in comparison with that of different nodes located in the same rack.

HDFS compromises its reliability to achieve lower communication costs.

For example, for the default replication factor of three, HDFS stores one replica in the same node the original data is stored, one replica on a different node but in the same rack, and one replica on a different node in a different rack to provide three copies of the data

Heartbeat and Blockreport messages

Heartbeats and Block reports are periodic messages sent to the NameNode by each DataNode in a cluster.

Receipt of a Heartbeat implies that the DataNode is functioning properly, while each Blockreport contains a list of all blocks on a DataNode.

The NameNode receives such messages because it is the sole decision maker of all replicas in the system.

HDFS High-Throughput Access to Large Data Sets (Files)

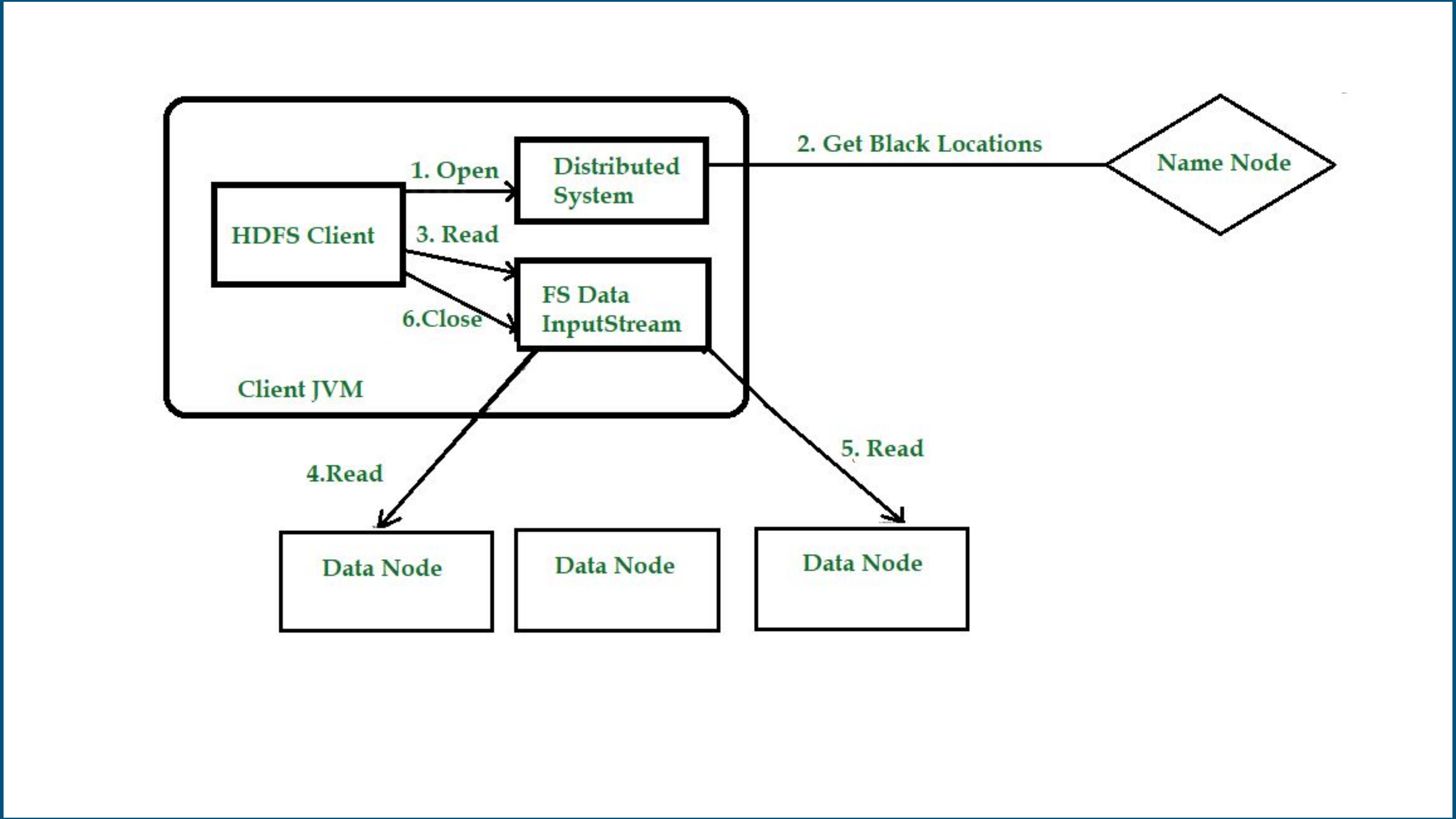
HDFS is primarily designed for batch processing rather than interactive processing, data access throughput in HDFS is more important than latency.

Applications run on HDFS typically have large data sets, individual files are broken into large blocks (e.g., 64MB) to allow HDFS to decrease the amount of metadata storage required per file.

Advantages:

The list of blocks per file will shrink as the size of individual blocks increases, and by keeping large amounts of data sequentially within a block, HDFS provides fast streaming reads of data.

HDFS Operation



Reading a file

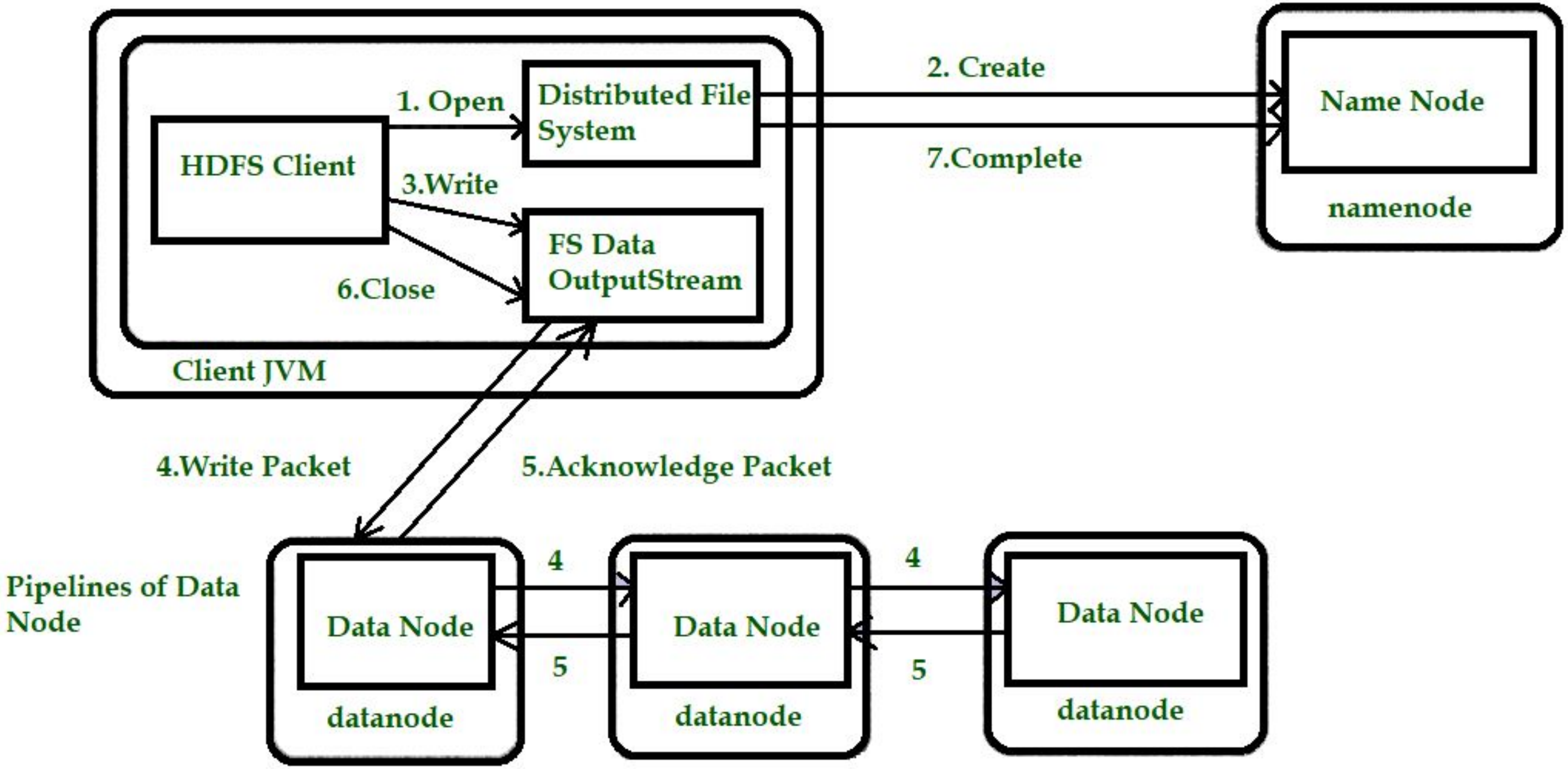
To read a file in HDFS, a user sends an “open” request to the NameNode to get the location of file blocks.

For each file block, the NameNode returns the address of a set of DataNodes containing replica information for the requested file.

The number of addresses depends on the number of block replicas.

Upon receiving such information, the user calls the read function to connect to the closest DataNode containing the first block of the file.

After the first block is streamed from the respective DataNode to the user, the established connection is terminated and the same process is repeated for all blocks of the requested file until the whole file is streamed to the user.


HDFS Client
1. Open
Distributed File System
2. Create
7.Complete
Name Node
namenode
3.Write
FS Data OutputStream
6.Close
Client JVM
4.Write Packet
5.Acknowledge Packet
Pipelines of Data Node
Data Node
datanode
4
5
Data Node
datanode
4
5
Data Node
datanode

Writing to a file

To write a file in HDFS, a user sends a “create” request to the NameNode to create a new file in the file system namespace.

If the file does not exist, the NameNode notifies the user and allows him to start writing data to the file by calling the write function.

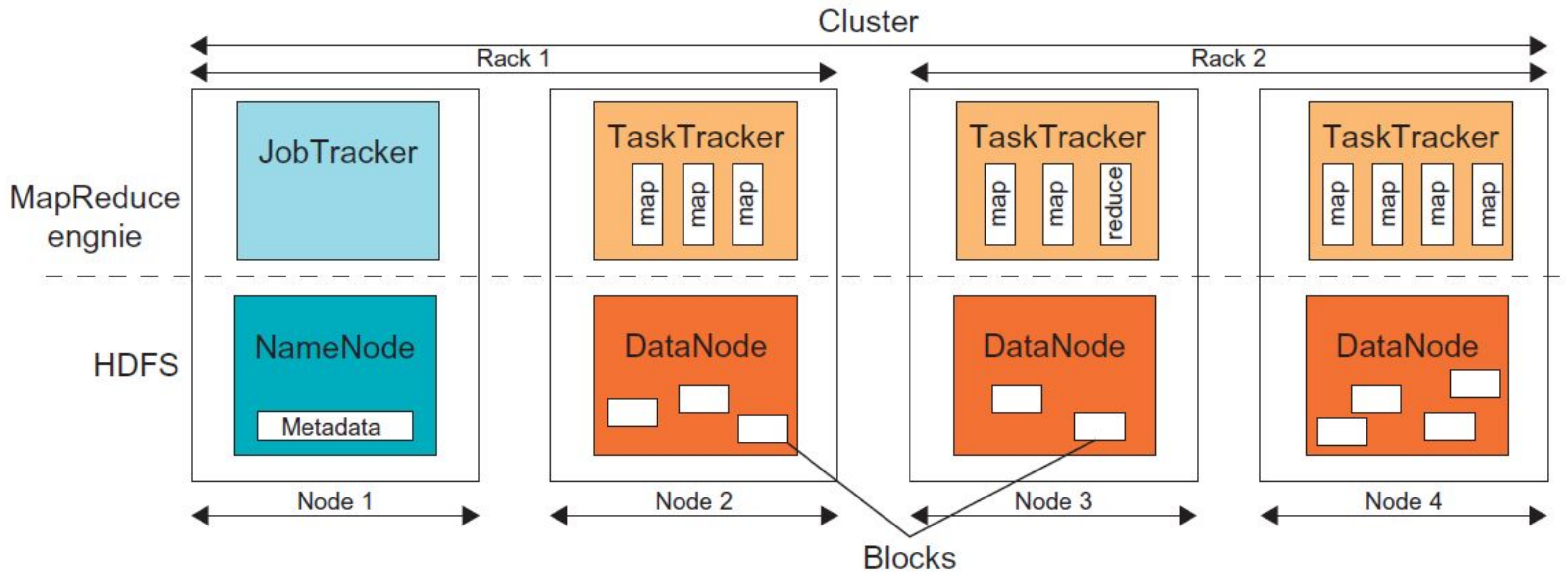
The first block of the file is written to an internal queue termed the data queue while a data streamer monitors its writing into a DataNode.

Each file block needs to be replicated by a predefined factor, the data streamer first sends a request to the NameNode to get a list of suitable DataNodes to store replicas of the first block.

The steamer then stores the block in the first allocated DataNode. Afterward, the block is forwarded to the second DataNode by the first DataNode.

The process continues until all allocated DataNodes receive a replica of the first block from the previous DataNode.

Once this replication process is finalized, the same process starts for the second block and continues until all blocks of the file are stored and replicated on the file system.



Architecture of MapReduce in Hadoop

The top most layer of Hadoop is the MapReduce engine that manages the data flow and control flow of MapReduce jobs over distributed computing systems.

The MapReduce engine also has a master/slave architecture consisting of a single JobTracker as the master and a number of Task Trackers 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 TaskTracker manages the execution of the map and/or reduce tasks on a single computation node in the cluster

Each TaskTracker 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.

For example, a TaskTracker node with N CPUs, each supporting M threads, has M * N simultaneous execution slots.

It is worth noting that each data block is processed by one map task running on a single slot.

Therefore, there is a one-to-one correspondence between map tasks in a TaskTracker and data blocks in the respective DataNode.

Running a Job in Hadoop

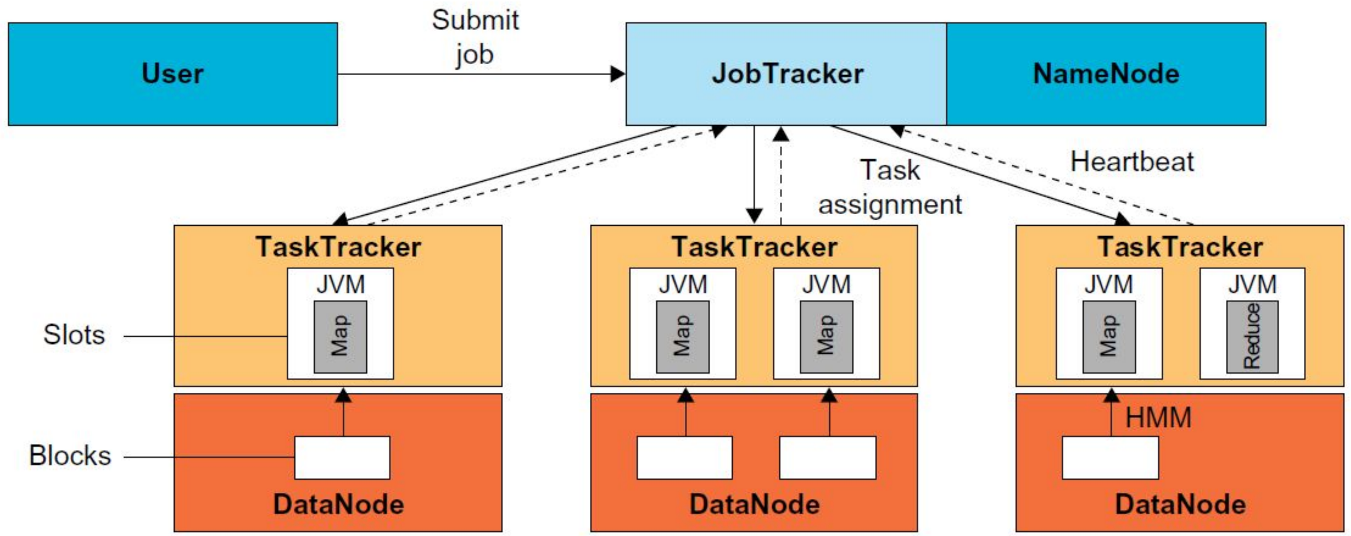


FIGURE 6.12

Data flow in running a MapReduce job at various task trackers using the Hadoop library.

Running a Job in Hadoop

Three components contribute in running a job in this system: a user node, a Job Tracker, and several Task Trackers.

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.

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 Job Tracker’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 JobTracker considers the localization of the data when assigning the map tasks to the TaskTrackers.

The JobTracker also creates reduce tasks and assigns them to the TaskTrackers.

The number of reduce tasks is predetermined by the user, and there is no locality consideration in assigning them.

Task execution

The control flow to execute a task (either map or reduce) starts inside the TaskTracker by copying the job JAR file to its file system.

Instructions inside the job JAR file are executed after launching a Java Virtual Machine (JVM) to run its map or reduce task.

Task running check

A task running check is performed by receiving periodic heartbeat messages to the JobTracker from the TaskTrackers.

Each heartbeat notifies the JobTracker that the sending TaskTracker is alive, and whether the sending TaskTracker is ready to run a new task.