engine so that the user can transform any data before it is moved between sources and sinks.

Apache Mahout: it is a collection of scalable data mining and machine learning algorithms implemented in Java. Four main groups of algorithms are:

- Recommendations, a.k.a. collective filtering
- Classification, a.k.a categorization
- Clustering
- Frequent itemset mining, a.k.a parallel frequent pattern mining

It is not merely a collection of algorithms as many machine learning algorithms are non-scalable, the algorithms in Mahout are written to be distributed in nature and use the MapReduce paradigm for execution.

Oozie: it is used to manage and coordinate jobs executed on Hadoop.

2.3 Hadoop Distributed File System

The distributed file system in Hadoop is designed to run on commodity hardware. Although it has many similarities with other distributed file systems, the differences are significant. It is highly fault-tolerant and is designed to run on low-cost hardware. It also provides high-throughput to stored data, hence can be used to store and process large datasets. To enable this streaming of data it relaxes some POSIX standards. HDFS was originally built for the Apache Nutch project and later forked in to an individual project under Apache [21].

HDFS by design is able to provide reliable storage to large datasets, allowing high-bandwidth data streaming to user applications. By distributing the storage and computation across several servers, the resource can scale up and down with demand while remaining economical.

HDFS is different from other distributed file systems in the sense that it uses a write-once-read-many model that relaxes concurrency requirements, provides simple data coherency, and enables high-throughput data access [22]. HDFS prides on the principle and proves to be more efficient when the processing is done near the data rather than moving the data to the applications space. The data writes are restricted to one writer at a time. The bytes are appended to the end of the stream and are stored in the order written. HDFS has many notable goals:

- Ensuring fault tolerance by detecting faults and applying quick recovery methods.

- MapReduce streaming for data access.
- Simple and robust coherency model.
- Processing is moved to the data, rather than data to processing.
- Support heterogeneous commodity hardware and operating systems.
- Scalability in storing and processing large amounts of data.
- Distributing data and processing across clusters economically.
- Reliability by replicating data across the nodes and redeploying processing in the event of failures.

2.3.1 Characteristics of HDFS

Hardware Failure: Hardware failure is fairly common in clusters. A Hadoop cluster consists of thousands of machines, each of which stores a block of data. HDFS consists of a huge number of components and with that there is a good chance of failure among them at any point of time. The detection of these failures and the ability to quickly recover is part of the core architecture.

Streaming Data Access: Applications that run on the Hadoop HDFS need access to streaming data. These applications cannot be run on general-purpose file systems. HDFS is designed to enable large-scale batch processing, which is enabled by the high-throughput data access. Several POSIX requirements are relaxed to enable these special needs of high throughput rates.

Large Data Sets: The HDFS-based applications feed on large datasets. A typical file size is in the range of high gigabytes to low terabytes. It should provide high data bandwidth and support millions of files across hundreds of nodes in a single cluster.

Simple Coherency Model: The write-once-read-many access model of files enables high throughput data access as the data once written need not be changed, thus simplifying data coherency issues. A MapReduce-based application takes advantage of this model.

Moving compute instead of data: Any computation is efficient if it executes near the data because it avoids the network transfer bottleneck. Migrating the computation closer to the data is a cornerstone of HDFS-based programming. HDFS provides all the necessary application interfaces to move the computation close to the data prior to execution.

Heterogeneous hardware and software portability: HDFS is designed to run on commodity hardware, which hosts multiple platforms. This features helps widespread adoption of this platform for large-scale computations.

Drawbacks:

Low-latency data access: The high-throughput data access comes at the cost of latency. Latency sensitive applications are not suitable for HDFS. HBase has shown promise in handling low-latency applications along with large-scale data access.

Lots of small files: The metadata of the file system is stored in the namenode's memory(master node). The limit to the number of files in the file-system is dependent on the namenode memory. Typically, each file, directory, and block takes about 150 bytes. For example, if you had one million files, each taking one block, you would need at least 300 MB of memory. Storing millions of files seems possible, but hardware is incapable of accommodating billions of files [24].

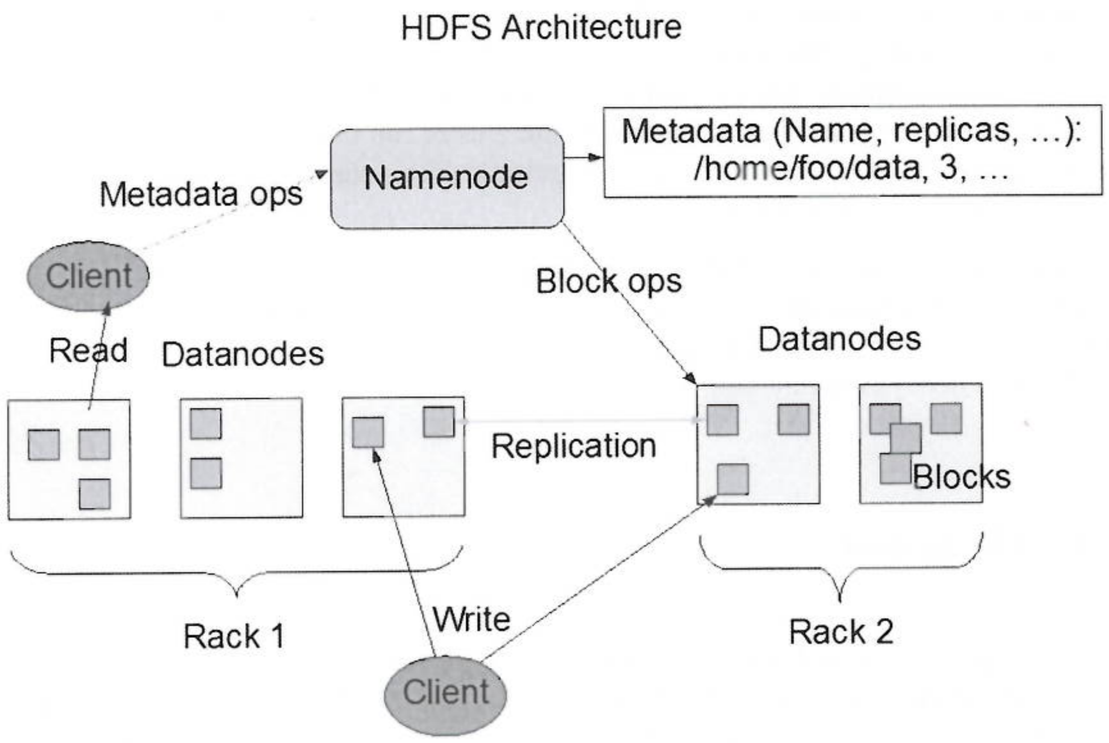

Fig. 2.2: HDFS Architecture [1]

2.3.2 Namenode and Datanode

The HDFS architecture is built in tandem with the popular master/slave architecture. An HDFS cluster consists of a master server that manages the file-system namespace and regulates files access, called the *Namenode*. Analogous to slaves, there are a number of *Datanodes*. These are typically one per cluster node and manage the data stored in the nodes. The HDFS file-system exists independently of the host file-system and allows data to be stored in its own namespace . The Namenode allows typical file-system operations like opening, closing, and renaming files and directories. It also maintains data block and Datanode mapping information. The Datanodes handle the read and write requests. Upon Namenode's instructions the Datanodes perform block creation, deletion, and replications operations. HDFS architecture is given in Figure 2.2

Namenode and Datanodes are software services provided by HDFS that are designed to run on heterogeneous commodity machines. These applications typically run on Unix/Linux-based operating systems. Java programming language is used to build these services. Any machine that supports the Java runtime environment can run Namenode and Datanode services. Given the highly portable nature of Java programming language HDFS can be deployed on wide range of machines. A typical cluster installation has one dedicated machine that acts as a master running the Namenode service. Other machines in the cluster run one instance of Datanode service per node. Although you can run multiple Datanodes on one machine, this practice is rare in real-world deployments.

A single instance of Namenode/master simplifies the architecture of the system. It acts as an arbitrator and a repository to all the HDFS meta-data. The system is designed such that there is data flow through the Namenode. Figure 2.3 shows how the Hadoop ecosystem interacts together.

2.3.3 File System

The file organization in HDFS is similar to the traditional hierarchical type. A user or an application can create and store files in directories. The namespace hierarchy is similar to other file-systems in the sense that one can create, remove, and move files from one directory to another, or even rename a file. HDFS does not support hard-links or soft-links.

Any changes to the file-system namespace is recorded by the Namenode. An application can specify the replication factor of a file that should be maintained by HDFS. The number of copies of a file is called the replication factor.

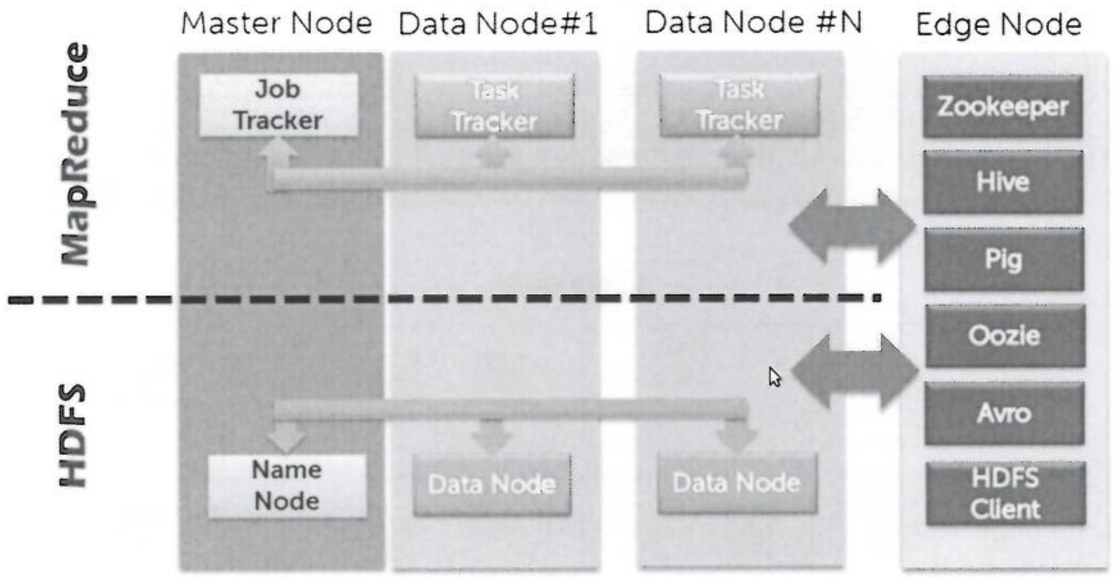

Fig. 2.3: Interaction between HDFS and MapReduce [1]

2.3.4 Data Replication

HDFS provides a reliable storage mechanism even though the cluster spans thousands of machines. Each file is stored as a sequence of blocks, where each block except the last one is of the same size. Fault tolerance is ensured by replicating the file blocks. The *block size* and *replication factor* is configurable for each file either by the user or an application. The replications factor can be set at file creation and can be modified later. Files in HDFS are write-once and strictly adhere to a single writer at a time property (Figure 2.4).

Namenode, acting as the master, takes all the decisions regarding data block replication. It receives a heartbeat, check Figure 2.5 on how it works, and block report from the Datanodes in the cluster. Heartbeat implies that the Datanode is functional and the block report provides a list of all the blocks in a Datanode.

Replication is vital to HDFS reliability and performance is improved by optimizing the placement of these replicas. This optimized placement contributes significantly to the performance and distinguishes itself from the rest of the file-systems. This feature requires much tuning and experience to get it right. Using a rack-aware replication policy improves data availability, reliability, and efficient network bandwidth utilization. This policy is the first of its kind and has seen much attention with better and sophisticated policies to follow.

Typically large HDFS instances span across multiple racks. Communication between these racks go through switches. Network bandwidth between machines in different racks is less than machines within the same rack.

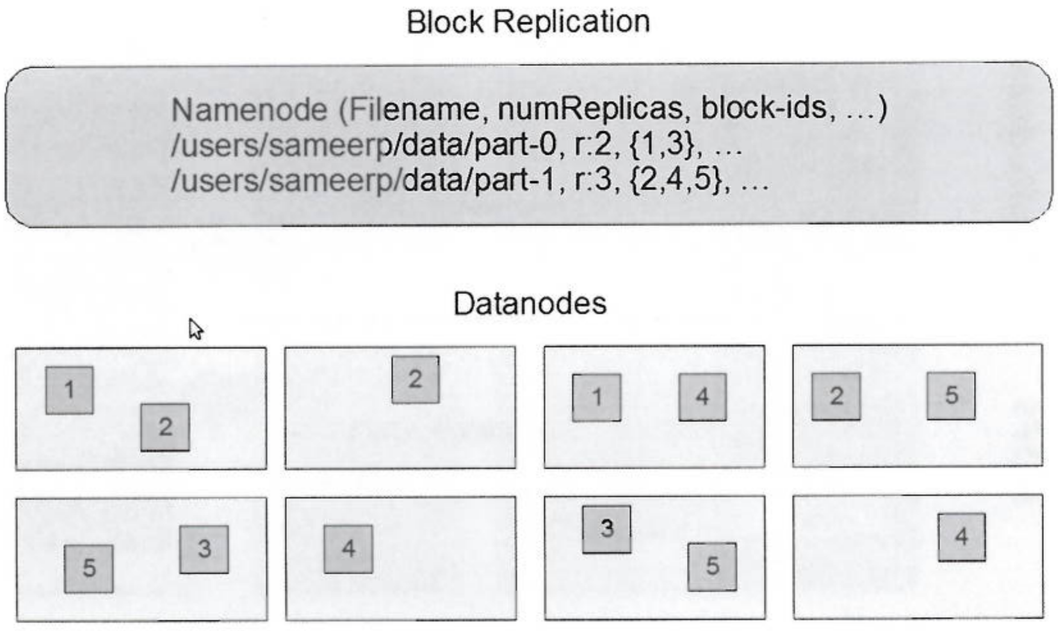
Block Replication
Namenode (Filename, numReplicas, block-ids, ...)
/users/sameerp/data/part-0, r:2, {1,3}, ...
/users/sameerp/data/part-1, r:3, {2,4,5}, ...
Datanodes
1
2
2
1
4
2
5
5
3
4
3
5
4

Fig. 2.4: Data Replication in Hadoop [22]

The Namenode is aware of the rack-ids each Datanode belongs to. A simple policy can be implemented by placing unique replicas in individual racks; in this way even if an entire rack fails, although unlikely, the replica on another rack is still available. However, this policy can be costly as the number of writes between racks increases.

When the replication factor is, say 3, the placement policy in HDFS is put a replica in one node per rack. This policy cuts the inter-rack write, which improves performance. The chance of rack failure is far less than node failure. This policy significantly reduces the network bandwidth used when reading data as a replica is placed in only two unique racks instead of three. One-third of replicas are on one node and one-third in another rack, while another third is evenly distributed across remaining racks. This policy improves performance along with data reliability and read performance.

Replica Selection: Selecting a replica that is closer to the reader significantly reduces the global bandwidth consumption and read latency. If a replica is present on the same rack then it is chosen instead of another rack, similarly if the replica spans data centers then the local data center hosting the replica is chosen.

Safe-mode: When HDFS services are started, the Namenode enters a safe-mode. The Namenode receives heartbeats and block reports from the Datanodes. Each block has a specified number of replicas and Namenode checks if these replicas are present. It also checks with Datanodes through the heartbeats. Once a good percentage of blocks are present the Namenode exits the safe-mode (takes about 30s). It completes the remaining replication in other Datanodes.

2.3.5 Communication

The TCP/IP protocols are foundations to all the communication protocols built in HDFS. Remote Procedure Calls (RPCs) are designed around the client protocols and the Datanode protocols. The Namenode does not initiate any procedure calls, it only responds to the request from the clients and the Datanodes.

Robustness: The fault-tolerant nature of HDFS ensures data reliability. The three common type of failures are Namenode, Datanode, and network partitions.

Data Disk Failure, Heartbeats, and Re-replication: Namenode receives periodic heartbeats from the Datanodes. A network partition, failure of a switch, can cause all the Datanodes connected via that network to be invisible to the Namenode. Namenode used heartbeats to detect the condition of nodes in the cluster, it marks them dead if there are no recent heartbeats and does not forward any I/O requests. Any data part of the dead Datanode is not available and the Namenode keeps track of these blocks and triggers replications whenever necessary. There are several reasons to re-replicate a data block: Datanode failure, corrupted data block, storage disk failure, increased replication factor .

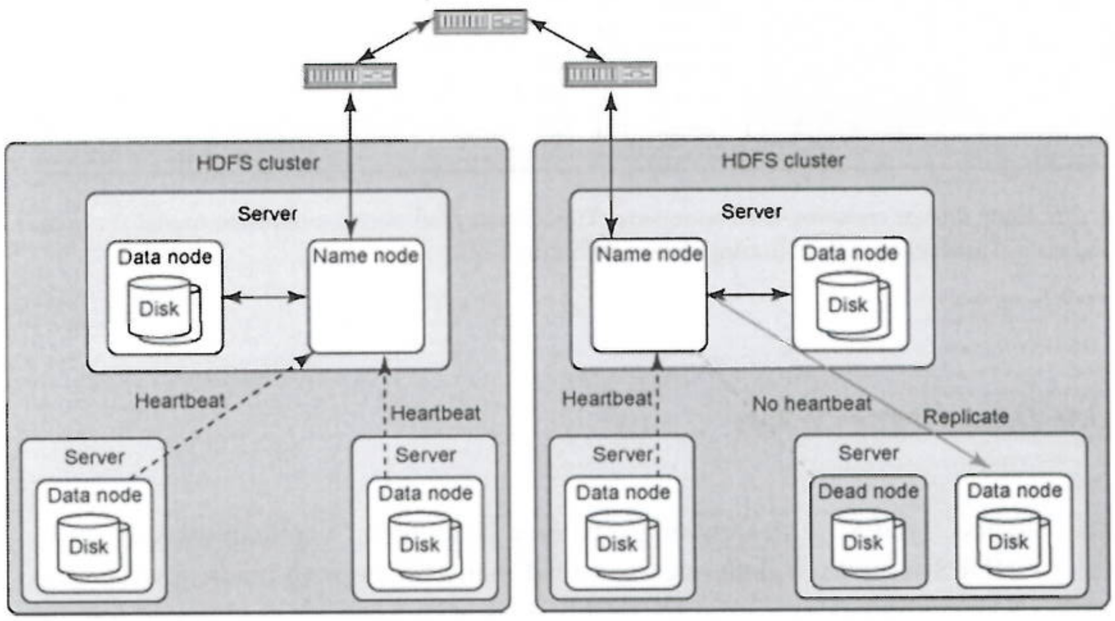

Fig. 2.5: Heart Beat Mechanism in Hadoop [22]

Cluster Rebalancing: Data re-balancing schemes are compatible with the HDFS architecture. It automatically moves the data from a Datanode when its free space falls below a certain threshold. In case a file is repeatedly used in an application, a scheme might create additional replicas of the file to re-balance the thirst for the data on the file in the cluster.

Data Integrity: Data corruption can occur for various reasons like storage device failure, network faults, buggy software, etc. To recognize corrupted data blocks HDFS implements a checksum to check on the contents of the retrieved HDFS files. Each block of data has an associated checksum that is stored in a separate hidden file in the same HDFS namespace. When a file is retrieved the quality of the file is checked with the checksum stored, if not then the client can ask for that data block from another Datanode (Figure 2.6).

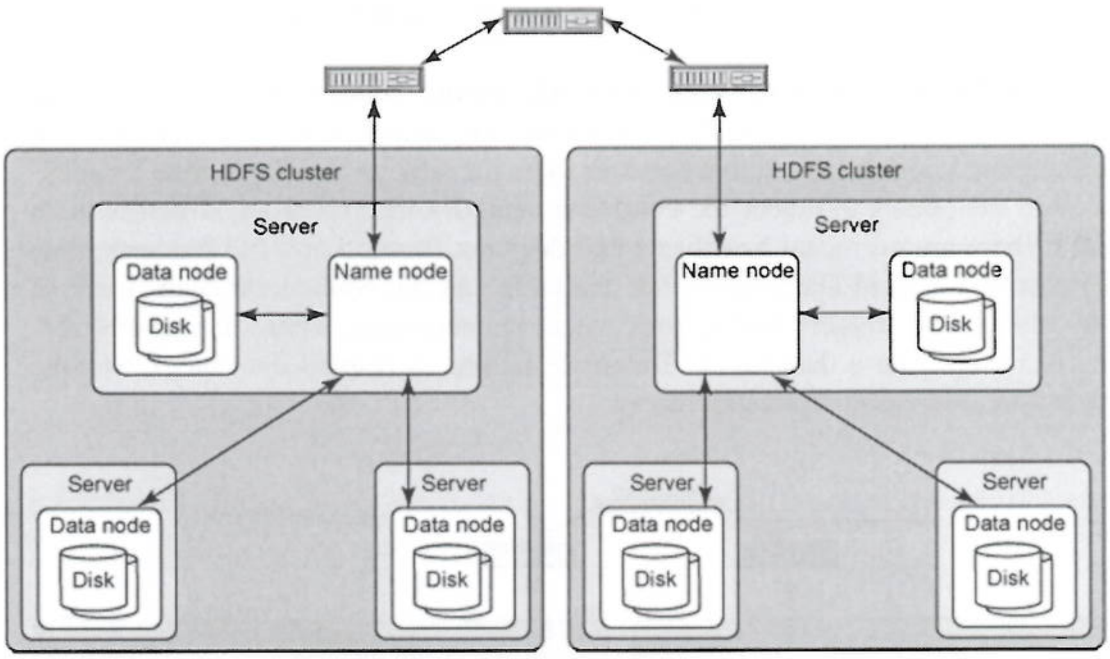

Fig. 2.6: Each cluster contains one Namenode. This design facilitates a simplified model for managing each Namespace and arbitrating data distribution [22]

2.3.6 Data Organization

Data Blocks: HDFS by design supports very large files. Applications written to work on HDFS write their data once and read many times, with reads at streaming speeds. A typical block size used by HDFS is 64 MB. Each file is chopped into 64 MB chunks and replicated.

Staging: A request to create a file does not reach the Namenode immediately. Initially, The HDFS client caches the file data into a temporary local file. Application writes are redirected to this temporary files. When the local file accumulates content over the block size, the client contacts the Namenode. The Namenode creates a file in the file-system and allocates a data block for it. The Namenode responds to the client with the Datanode and data block identities. The client flushes the block of data from the temporary file to this data block. If the file is closed the Namenode is informed, and it starts a file creation operation into a persistent store. Suppose, if the Namenode dies before commit the file is lost.

The benefits of the above approach is to allow streaming write to files. If a client writes a file directly to a remote file without buffering, the network speed and network congestion impacts the throughput considerably. There are earlier file-systems that successfully use client side caching to improve performance. In case of HDFS a few POSIX rules have been relaxed to achieve high performance data uploads.

Replication Pipelining: Local file caching mechanism described in the previous section is used for writing data to an HDFS file. Suppose the HDFS file has a replication factor of three. In such event the local file accumulates a data block of data, and the client receives a list of Datanodes from the Namenode that will host the replica of the data block. The client then flushes the data block in the first Datanode. Each Datanode in the list receives data block in smaller chunks of 4KB, the first Datanode persists this chunk in its repository after which it is flushed to the second Datanode. The second Datanode in turn persists this data chunk in its repository and flushes to the third Datanode, which persists the chunk in its repository. Thus the Datanodes receive data from the previous nodes in a *pipeline* and at the same time forwarding to the next Datanode in the pipeline.

2.4 MapReduce Preliminaries

Functional Programming: MapReduce framework facilitates computing with large volumes of data in parallel. This requires the workload to be divided across several machines. This model scales because there is no intrinsic sharing of data as the communication overhead needed to keep the data synchronized prevents the cluster from performing reliably and efficiently at large scale.

All the data elements of MapReduce cannot be updated, i.e., immutable. If you change the (key, value) pair in a mapping task, the change is not persisted. A new output (key, value) pair is created before it is forwarded by the Hadoop system into the next phase of execution.