

Distributed file systems

Replication strategy

By default, three replicas of each block are stored, although this can be changed by users for every file. Note that there is no such thing as a “main replica”, there are just several replicas.

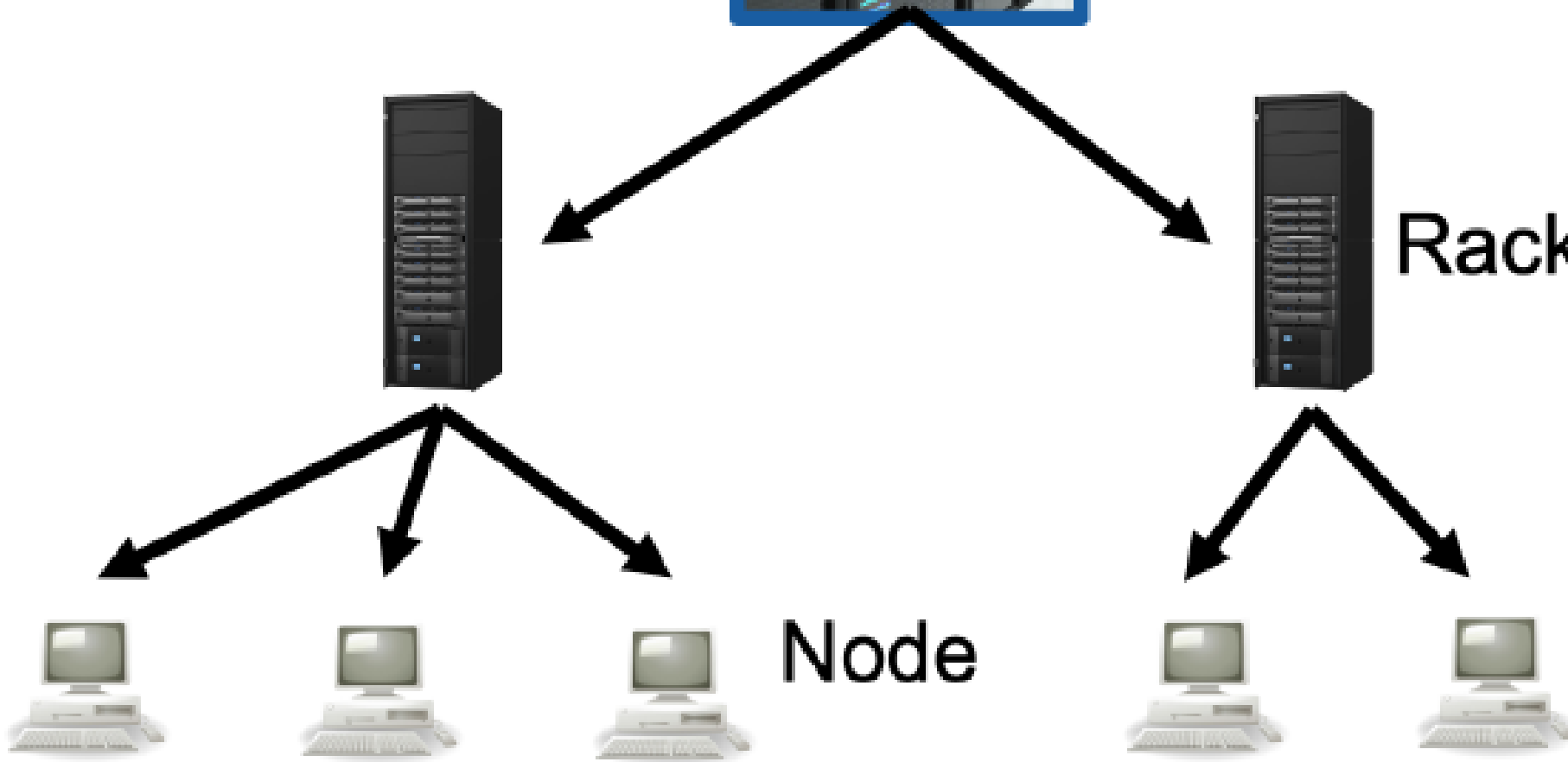
The placement strategy of replicas is based on knowledge of the physical setup of the cluster. As you may recall from the previous chapter, servers, called nodes, are organized in racks, and several racks are placed in the same room or data center. This gives a natural tree topology.

Cluster



Rack

Node





With this topology in mind, one can define a notion of distance between two nodes: two nodes in the same rack have a distance of 2 (one edge from the first node to the rack, and one from the rack to the other node). Two nodes in different racks have a distance of 4 (one edge from the first node to the first rack, one from the first rack to the data center, one from the data center to the other rack, one from the other rack to the other node). It is this distance that can then be used by the NameNode to sort DataNodes by increasing distance from the client.



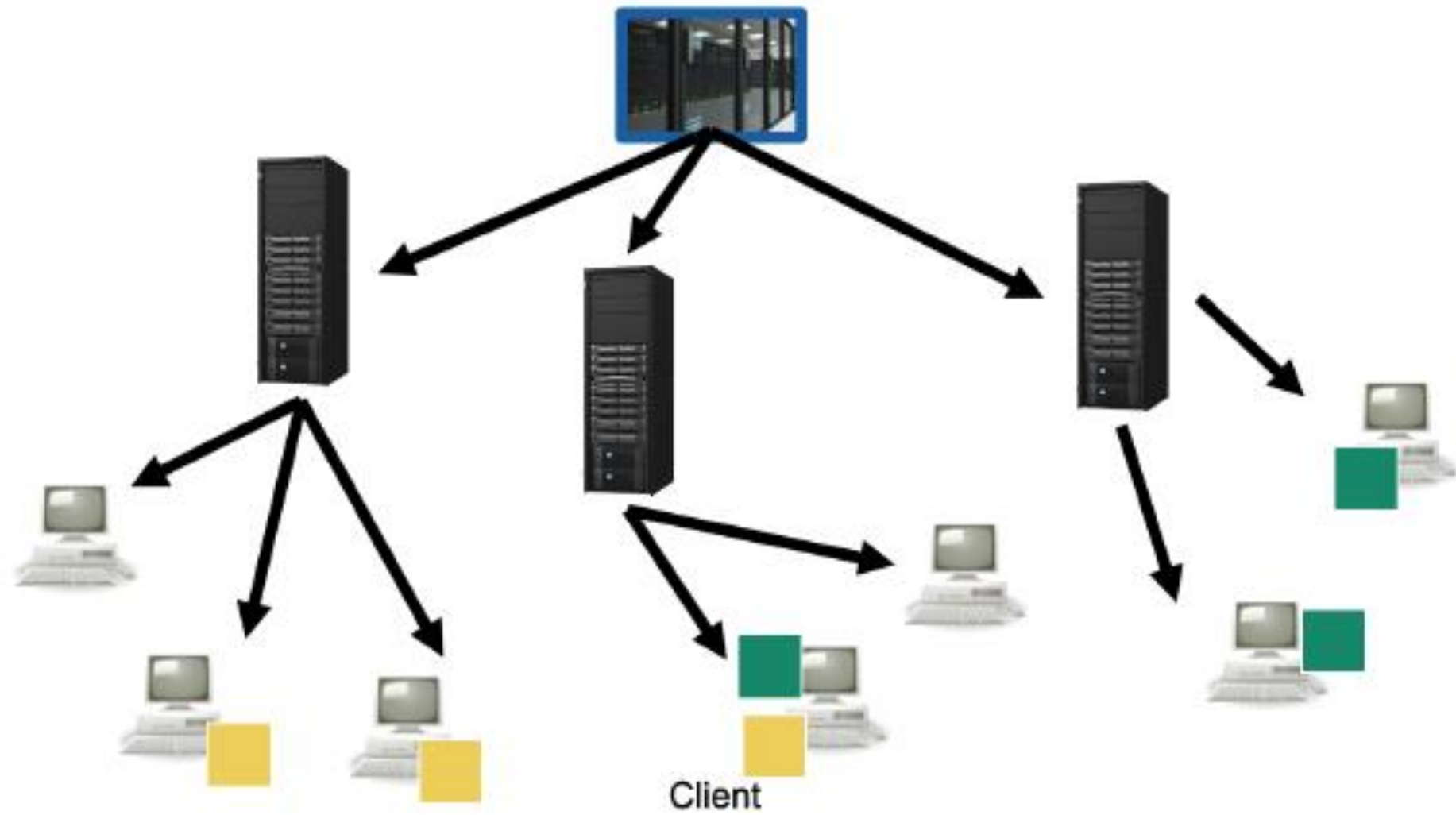
Now, the strategy for placing replicas of a new blocks is as follows. First, it is important to understand that, in practice, the client that is writing the file is a process running on one of the nodes of the HDFS cluster. This will become obvious when we study massive parallel computing (e.g., MapReduce and Spark), where reading and writing is done in a distributed fashion over the same cluster. But even without parallel computing, when users create clusters in public clouds such as AWS or Azure or Google Cloud, they connect to the machines using SSH (a safe protocol for remotely accessing a machine). Thus, anything they will do will originate from the machine they are connected to, which is in the cluster.



Having this in mind, the first replica of the block, by default, gets written to the same machine that the client is running on – keep in mind that this machine is typically a DataNode (again, this will become obvious when we look into MapReduce and Spark).

The second replica is written on a DataNode sitting in a different rack than the client, that we call B. The third replica is written to another DataNode on the same rack B. And further replicas are written mostly at random, but respecting two simple rules for resilience: at most one replica per node, and at most two replicas per rack.

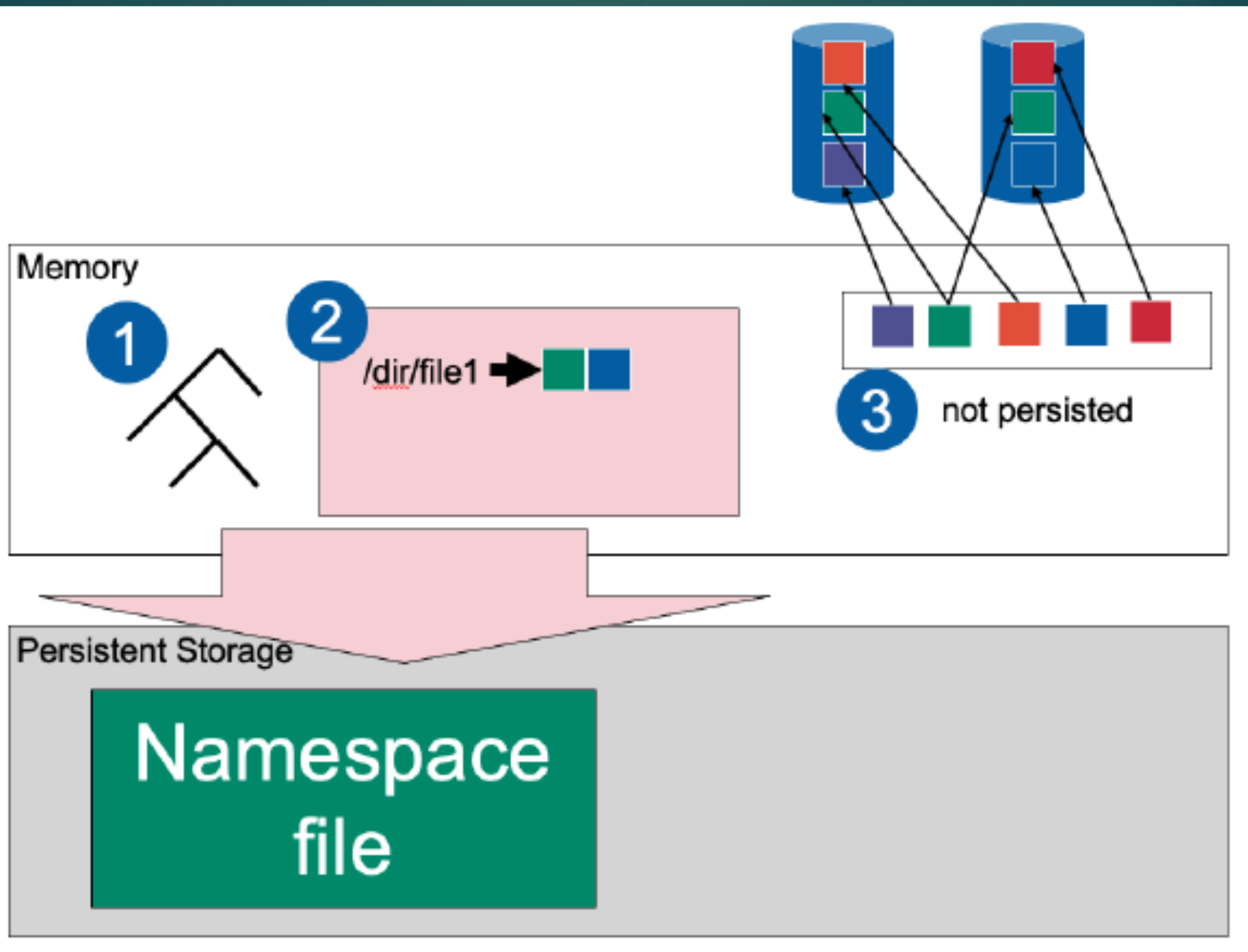
Below is an example with the three replicas of two different blocks, written by the same client.



Fault tolerance and availability

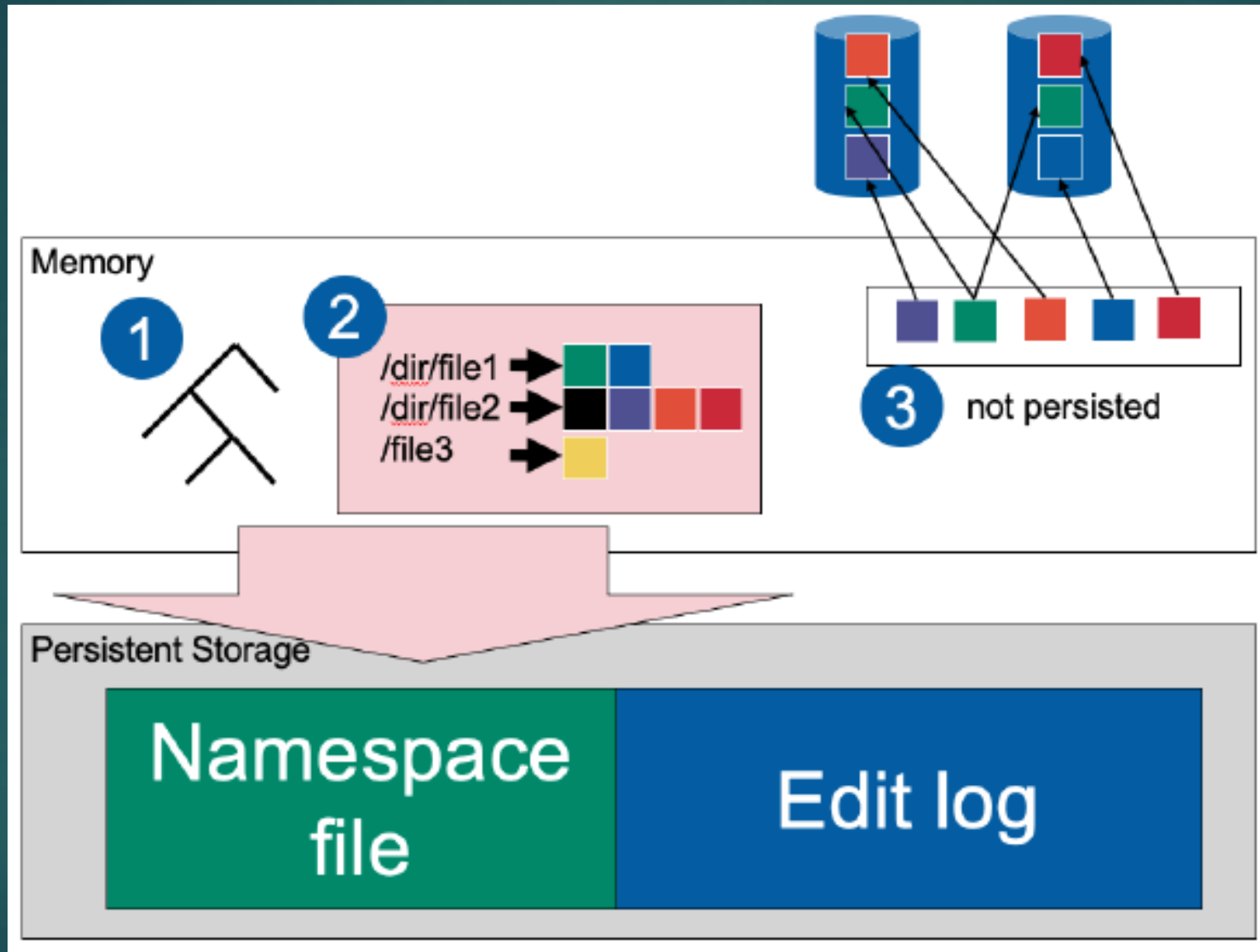
HDFS has a single point of failure: the NameNode. If the metadata stored on it is lost, then all the data on the cluster is lost, because it is not possible to reassemble the blocks into files any more.

For this reason, the metadata is backed up. More precisely, the file namespace containing the directory and file hierarchy as well as the mapping from files to block IDs is backed up to a so-called snapshot. Note that the mapping of block IDs to DataNodes does not require a backup, as it can be recovered from the periodic block reports.





Since the HDFS system is constantly updated, it would not be viable to do a backup upon each update. It would also not be viable to do backups less often, as this could lead to data loss. Thus, what is done is that updates to the file system arriving after the snapshot has been made are instead stored in a journal, called edit log, that lists the updates sorted by time of arrival.

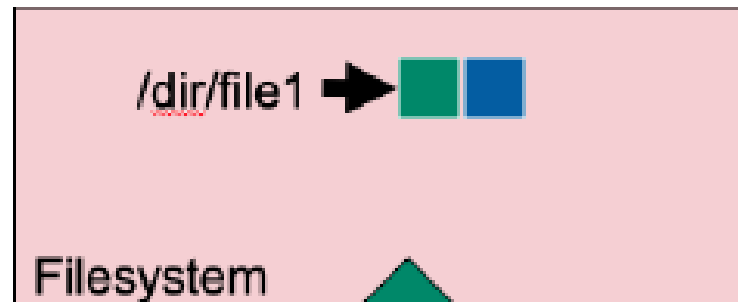




The snapshot and edit log are stored either locally or on a network-attached drive (not HDFS itself). For more resilience, they can also be copied over to more backup locations.

If the NameNode crashes, it can be restarted, the snapshot can be loaded back into memory to get the file namespace and the mapping of the files to block IDs.

Memory



Filesystem

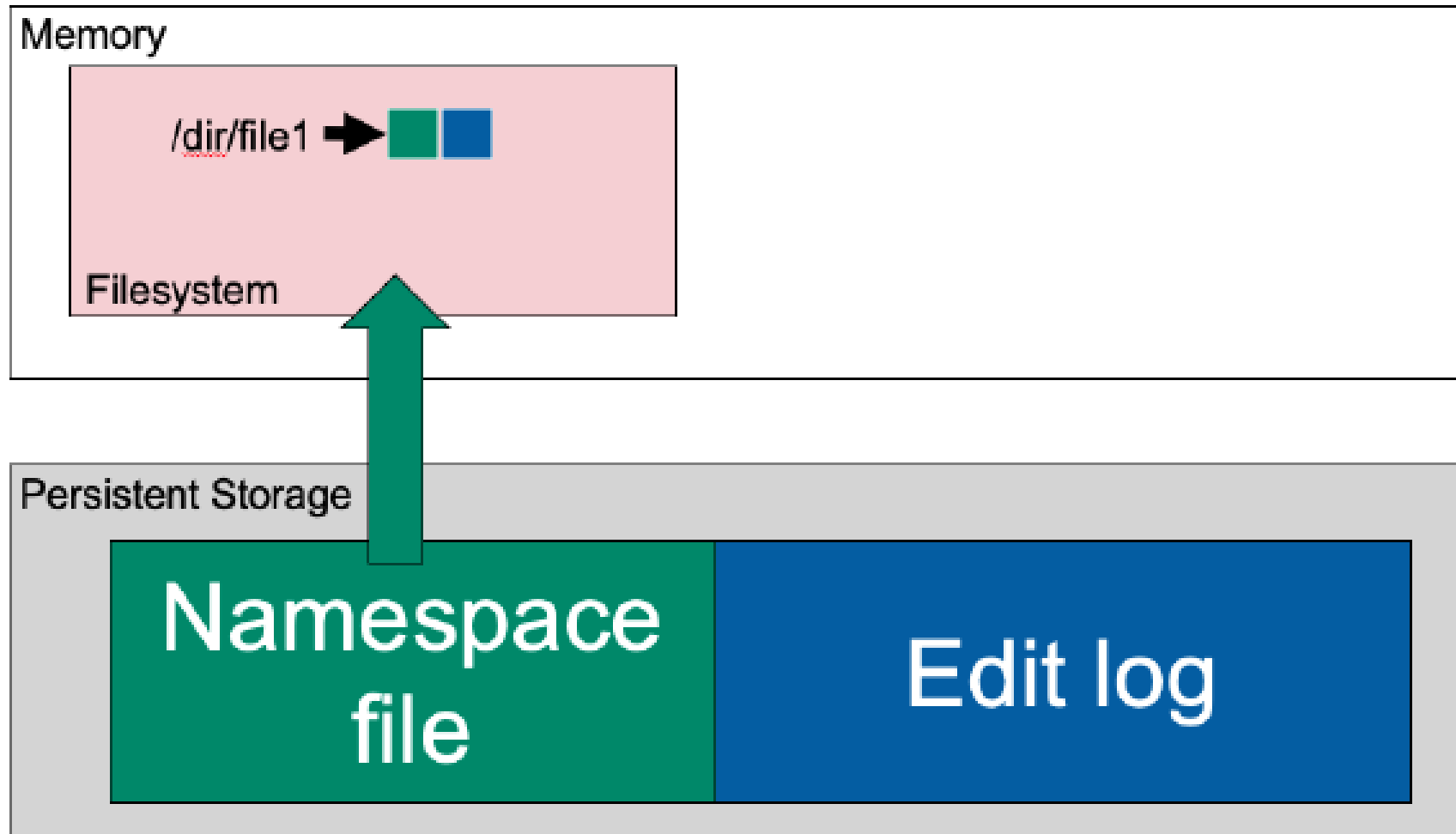
`/dir/file1` →

Persistent Storage

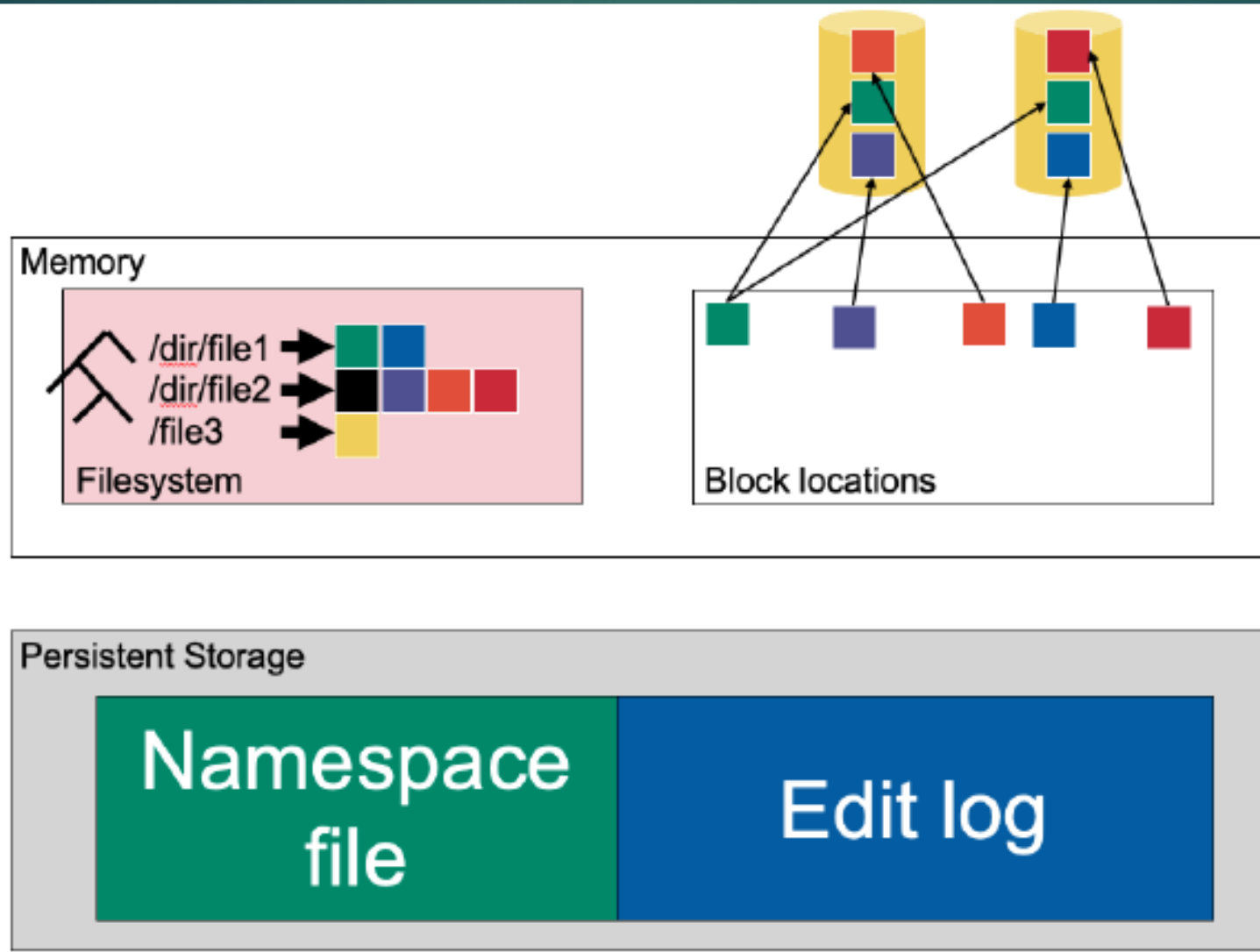
Namespace
file

Edit log

Then the edit log can be replayed in order to apply the latest changes.



And the NameNode can wait for (or trigger) block reports to rebuild the mapping from block IDs to the DataNodes that have a replica of them.





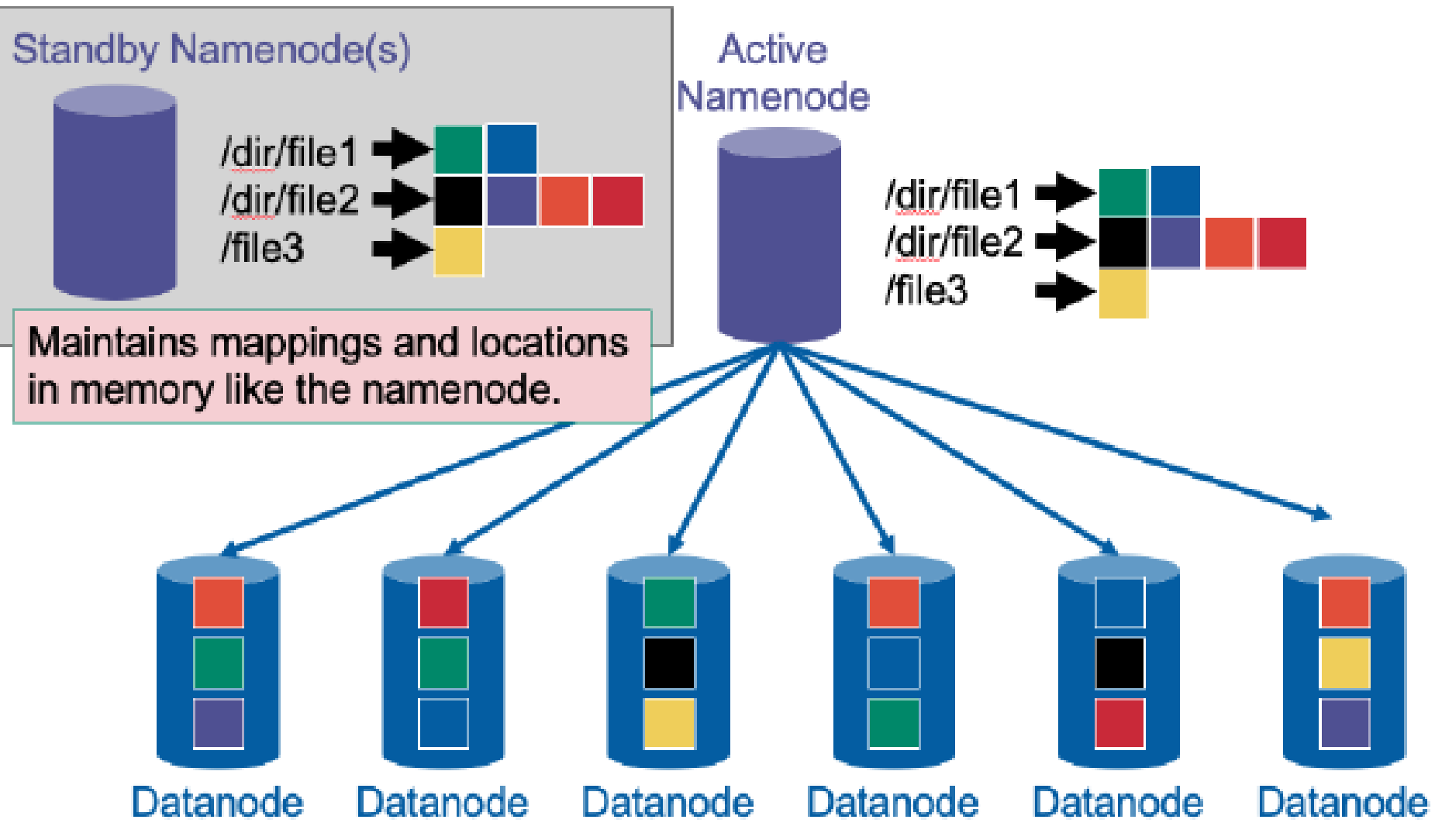
However, as more and more updates are applied, the edit log grows. This can easily lead to a long delay of 30+ minutes to restart the NameNode after a crash. More strategies had to be put in place. This was done incrementally in each HDFS release, and there are many different kinds of additional NameNodes that came into place and succeeded to one another: the Checkpoint NameNode, the Backup NameNode, the Secondary NameNode, the Standby NameNode, etc. We will not go into the details of each one of the version, but summarize the most important take aways.



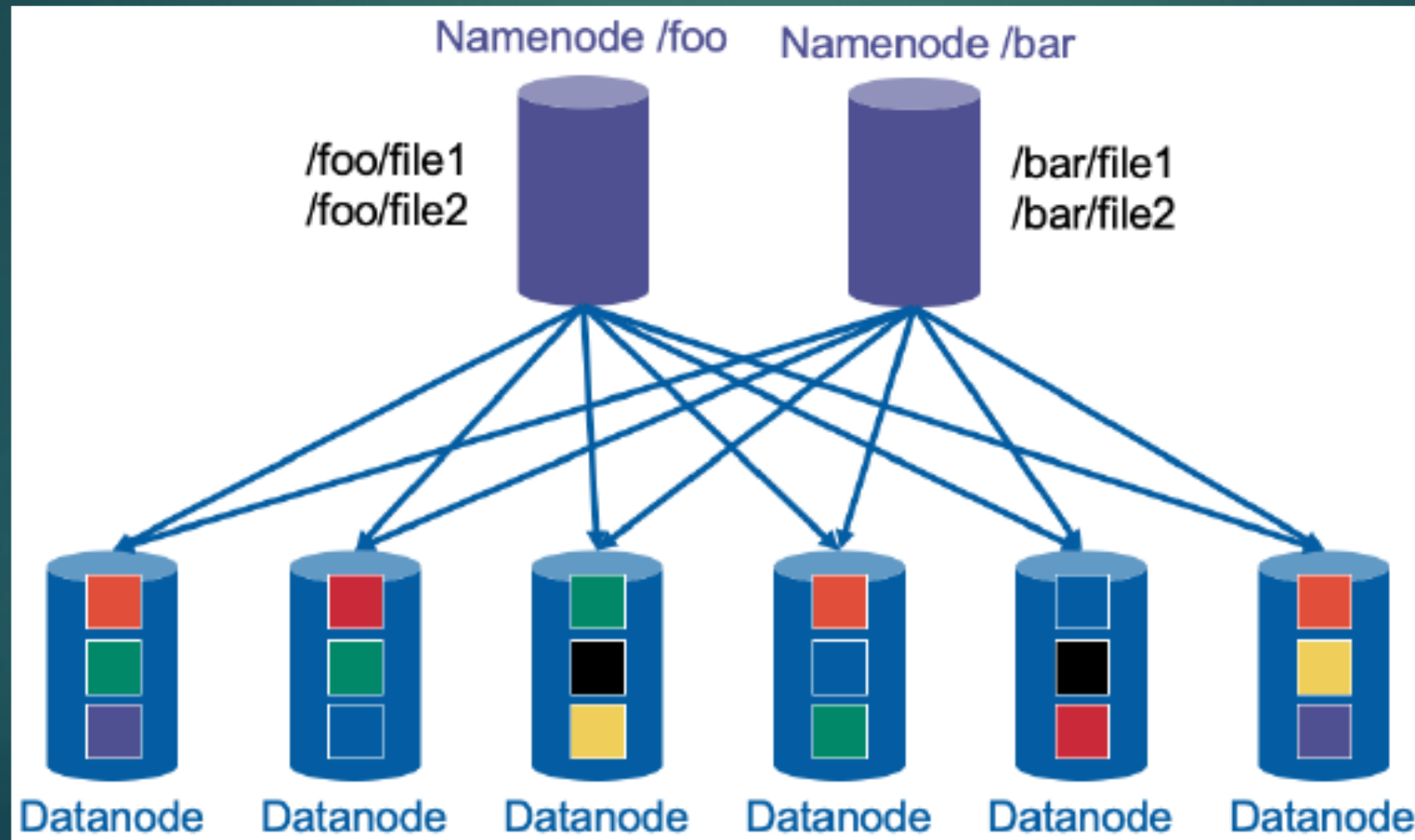
First, the edit log is periodically merged back into a new, larger snapshot and reset to an empty edit log. This is called a checkpoint. This can be done with a “phantom NameNode” (our terminology) that keeps the exact same structures in its memory as the real NameNode, and performs checkpoints periodically.

Second, it is possible to configure a phantom NameNode to be able to instantly take over from the NameNode in case of a crash. This considerably shortens the amount of time to recover.

As of the time of writing, these are called Standby NameNodes, but we do warn that the HDFS architecture will continue to evolve.



Another change to the architecture was made later: Federated NameNodes. In this change, several NameNodes can run at the same time, and each NameNode is responsible for specific (non overlapping) directories within the hierarchy. This spreads the management workload over multiple nodes.



Using HDFS

One of the ways to access HDFS is via the command line (a shell). The HDFS command line interface is mostly POSIX compliant, which means that the names of the commands are quite similar to those found on Linux or macOS.

HDFS command start with

```
hdfs dfs <put command here>
```

Although we recommend to use the `hadoop` command instead, because it can also connect to other file systems (S3, your local drive, etc).

```
hadoop fs <put command here>
```

You can list the contents of the current directory with:

```
hadoop fs -ls
```



You can print to screen the contents of a file with:

```
hadoop fs -cat /user/hadoop/dir/file.json
```

You can delete a file with:

```
hadoop fs -rm /user/hadoop/dir/file.json
```

You can create a directory with:

```
hadoop fs -mkdir /user/hadoop/dir/file.json
```

You can upload files from your local computer to HDFS with:

```
hadoop fs -copyFromLocal localfile1 localfile2  
                               /user/hadoop/targetdirectory
```

You can upload a file from HDFS to your local computer with:

```
hadoop fs -copyToLocal /user/hadoop/file.json  
                        localfile.json
```

Paths and URIs

Although in the previous examples we used relative paths, generally, paths are URIs, as seen in the previous chapter. The scheme determines the file system.

This is what an HDFS URI looks like:

```
hdfs://www.example.com:8021/user/hadoop/file.json
```

This is what an S3 URI looks like (there are also the s3a and s3n schemes):

```
s3://my-bucket/directory/file.json
```

In Azure blob storage:

```
wasb://mycontainer@myaccount.blob.core.windows.net  
/directory/file.json
```

And on the local file system:

```
file:///home/user/file.json
```

Typically, a default file system can be defined in a configuration file called `core-site.xml`:

```
<properties> <property>  
<name>fs.defaultFS</name>  
<value>hdfs://hdfs.mycluster.example.com:8020</value>  
<description>NameNode hostname</description>  
</property> </properties>
```

Then, the scheme is not needed to access files on this system and an absolute path starting with a slash can be used instead:

```
/user/hadoop/file.json
```

Note that other file systems can still be accessed (if set up properly) by using a URI scheme.

As for relative paths, they are resolved against the working directory, which should be unsurprising to anybody familiar with the command line interface even outside the context of HDFS.

Please mind that the current working directory might be on a different file system than the default file system; then you have situations in which a relative path will be local, while an absolute path will be on cloud, or distributed storage. This can cause unexpected issues. A common mistake is for users to use a relative path within a massive parallel computing framework (MapReduce, Apache Spark) when the working directory is local, which causes the data to be read, or written to the local disk rather than the HDFS cluster. If the job succeeds but the output is nowhere to be seen, it is likely it was just spit all over the local disks of the cluster. You can play treasure hunt and go after each machine to download the results, or you might realize that it will be easier to just rerun your job with the correct paths.

Logging and importing data

Two tools are worth mentioning: Apache Flume lets you collect, aggregate and move log data to HDFS. Apache Sqoop lets you import data from a relational database management system to HDFS.

This completes the chapter on distributed file systems. The most adventurous readers are encouraged to create a small HDFS cluster (this only takes a few clicks with Amazon EMR or Azure HDInsights), upload a few files, and log onto the machines with SSH to try to locate the physical block replicas as local files on the DataNodes.

Distributed file systems

Learning objectives

- a. Can you explain the difference between block storage and object storage?
- b. Can you explain the difference between the (logical) key-value model and a file system?
- c. Can you contrast block storage and object storage in terms of maximum number of objects/files?
- d. Can you contrast block storage and object storage in terms of the maximum size of objects/files?

- e. Do you know the order of magnitude of a block size for a local filesystem and for a distributed file system? Can you explain the rationale behind them with respect to latency and throughput?
- f. Do you know where HDFS shines and why?
- g. Do you know that HDFS files are updated by appending atomically and why?
- h. Do you know how HDFS performs in terms of throughput and latency?
- i. Do you know the main benefits of HDFS (commodity hardware, etc.)?
- j. Can you contrast centralized architectures to decentralized (peer-to-peer) architectures?

- k. Can you explain the HDFS architecture, what a NameNode is and what a DataNode is, how blocks are replicated?
- l. Can you sketch how the various components communicate with each other (client, NameNode, DataNode)?
- m. Can you point to the single points of failure of HDFS and explain how they can be addressed?
- n. Can you explain how the NameNode stores the file system namespace, in memory and on disk? In particular, can you explain how the namespace file and the edit log work together at startup time and how they get modified once the system is up and running?

- o. Can you explain what a Standby NameNode is? (Note: it has many predecessors that only have historical relevance in the development of HDFS: Backup NameNode, Secondary NameNode, Checkpoint NameNode, etc, but this is not important for the course)
- p. Are you able to use the HDFS shell (creating directories, reading, uploading and downloading files, etc.)?

HON LAL MIDTERM

Hadoop Ecosystem

The Apache™ Hadoop® project (Apache Software Foundation, 2006) facilitates open-source software for reliable, scalable, distributed computing. The Hadoop Ecosystem is an open-source framework, which facilitates the distributed storage and parallel processing of huge volumes of data across clusters of nodes or computers. The advantages of the Hadoop Ecosystem over traditional systems are summarized in Table 4.1.

Table 4.1 Comparison between Traditional System and Hadoop Ecosystem

<i>Traditional System</i>	<i>Hadoop Ecosystem</i>
Less scalable	Highly scalable
Vertical scaling	Horizontal scaling
Less fault-tolerant	More fault-tolerant
More investment needed	Economical
Handles only structured data	Handles any type of data

Components of the Hadoop Ecosystem

Organizations face many challenges while handling big data such as acquiring and storing huge volumes of data, analyzing large datasets for information needs, searching and sharing data and the synchronization and control of activities.

The Hadoop Ecosystem facilitates all of these activities with ease by providing different components or tools.

The Hadoop Ecosystem is comprised of several key components which can be distributed across four layers based on the functionalities as shown in Figure 4.1.

Data Storage

The bottom layer of the Hadoop Ecosystem provides the facility to store large volumes of data in different formats including structured, semi-structured and unstructured data. This layer uses two different components, one for mainly archived data and one for real-time data.

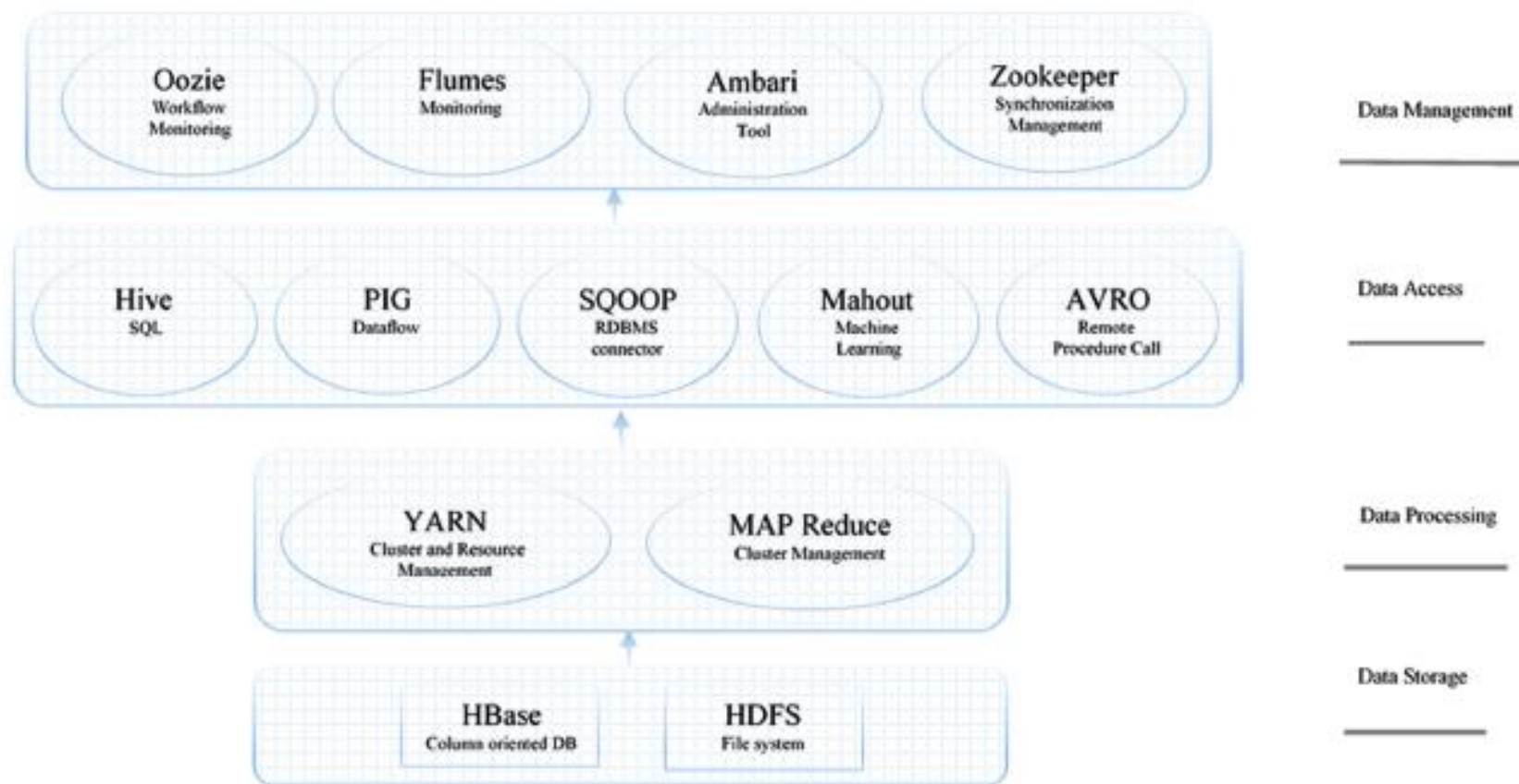


Figure 4.1 Layers in Hadoop Ecosystem.

Data Processing

This layer provides tools to facilitate the parallel processing of big data. The components used in this layer are YARN and MapReduce.

Data Access

This layer provides different tools to facilitate access to the big data repository to elicit the information required as per the business need. The main components used in this layer are HIVE, PIG, SQOOP, Mahout and Avro.

Data Management

This layer provides tools to monitor, manage, control and synchronize the data and activities in the Hadoop Ecosystem. The main components used in this layer are Oozie, Flumes, Ambari and Zookeeper.

Data Storage Component

Google File System (GFS)

The Apache Hadoop framework is based on the Google File System (GFS). The GFS is a distributed system for handling large data files developed by Google. GFS is based on master slave architecture where we have one master server and many slave machines called chunk servers. The files to be stored are divided into chunks and stored on the chunk servers. To maintain data reliability, replicas of each file are stored on multiple chunk servers. The user can set the number of replicas to be stored as per requirements; by default the number of replicas to be stored is three. The master server maintains the metadata to perform activities like namespace monitoring and access control, mapping the requested file to the chunk, etc. (Ghemawat et al., 2003). The database used in GFS framework is Google Bigtable.

Bigtable

To store huge volumes of data in petabytes, GFS uses a scalable sparsely populated table which stores data in the form of a sorted key/value map. Bigtable is very efficient for storing different types of data like graphs, IoT data and time series data. For workload balance and effective query response, the content of the Bigtable is divided into different tablets. Each tablet is a block of contiguous rows. These tablets are stored on Google's internal, highly durable file system called Colossus, using storage devices in Google's data centers (Chang et al., 2006).

Hadoop Distributed File System (HDFS)

Hadoop's principal storage system is the Hadoop Distributed File System (HDFS). It's a distributed file system that works well with large data files and can be deployed on a cluster of inexpensive computers. The Hadoop Distributed File System is a core component in the Hadoop architecture which sits in the data storage layer in the Hadoop Ecosystem.

HDFS follows a master-slave architecture where the single master system is called the NameNode and the slave machines are called the DataNodes as shown in Figure 4.2. The NameNode and DataNode are software written in Java and can run on any commodity machine that supports Java. The HDFS architecture provides a file system namespace to store user data in equal sized chunks which are stored in different DataNodes (Borthakur, 2008).

NameNode

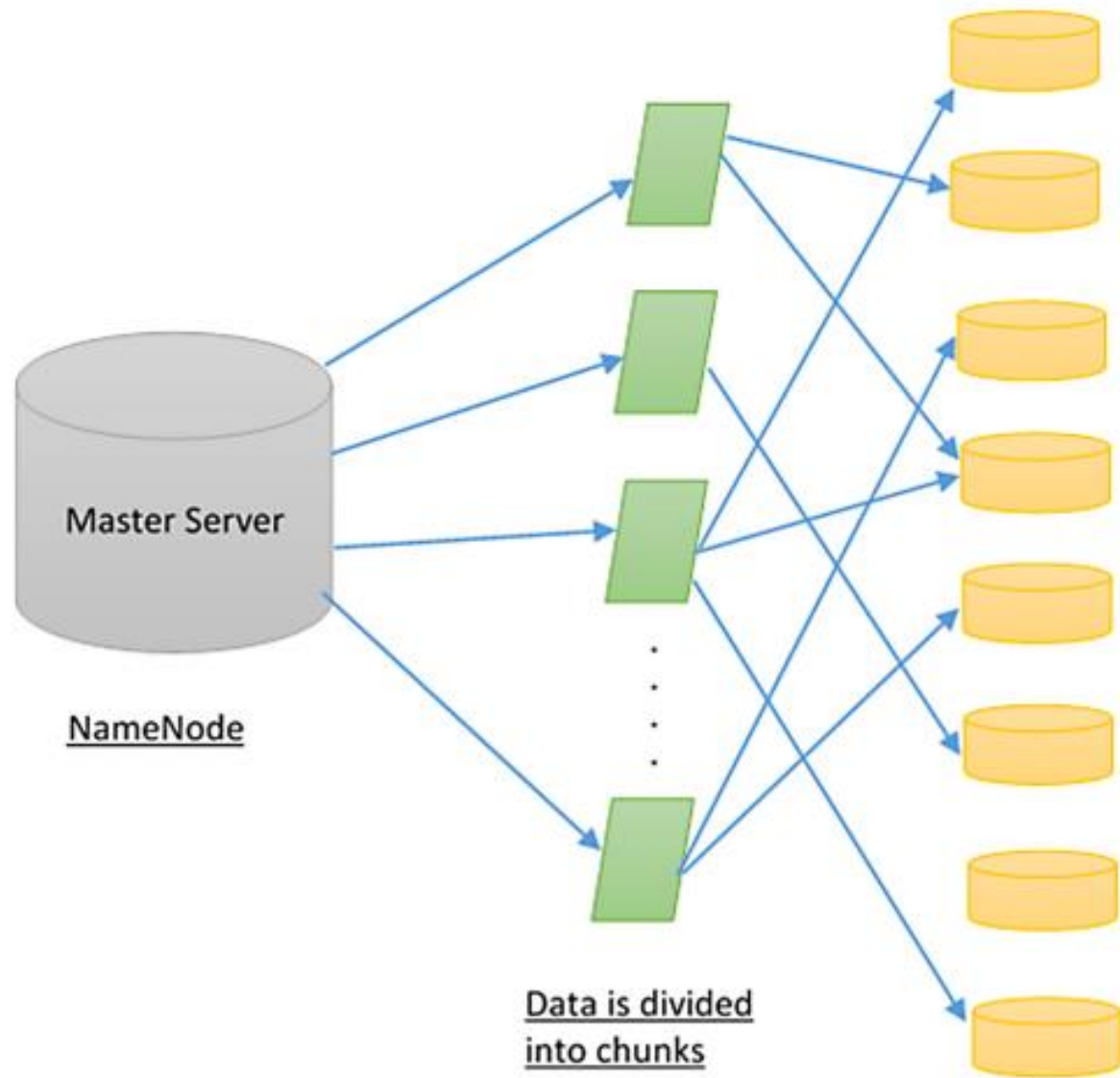


Figure 4.2 HDFS architecture.

Data Chunks are stored in DataNodes



Every HDFS system has a single master server called the NameNode. The NameNode is responsible for managing the file system namespace and allowing synchronized access to files by clients. The NameNode stores the metadata and performs operations like opening, closing and renaming files and directories. It also provides the mapping of chunk storages to DataNodes.

DataNode

The nodes that make up a Hadoop cluster and actually do the work are called DataNodes. They are utilized for the purpose of storing data chunks and transmitting these chunks to the clients in response to their read and write requests. The DataNodes also perform chunk creation, deletion and replication based on instructions from the NameNode.

Features of HDFS

The key features of the Hadoop Distributed File System are:

- Hierarchical file organization: HDFS provides a hierarchical organization to store data files. The HDFS architecture supports the creation of directories to store files. Just like any other file system, a user can create, move, rename or delete files.

- **Reliability:** The HDFS architecture ensures data reliability by maintaining duplicate copies of files on different machines. The number of duplicate copies, called the replication factor, is generally set by users at the time of configuration. The NameNode instructs the DataNodes to store the replicas of the files based on the replication factor given by the user. Rack-based automatic replication is a standard feature of the HDFS. DataNodes in the same rack save two copies of each block, while DataNodes in a different rack save the third copy of each block.



■ **Data integrity:** To ensure data integrity, HDFS architecture uses the error checksum method. At the time of file creation, the user generates the checksum for each chunk. When a user accesses a data file from any DataNode, the contents of the file are verified using the stored checksum. If the data is found to be corrupt, the user accesses the replicated copy of the file from another DataNode.

■ **Robustness:** HDFS provides a robust data storage architecture using Heartbeats and data re-replication. Each DataNode sends a periodic message called Heartbeat to the NameNode indicating that the DataNode is live and working fine. If a heartbeat is not received from a DataNode, the NameNode identifies that the node is dead and data files cannot be accessed from that node. Based on the replication factor, the NameNode performs the re-replication of data chunks.



HDFS architecture maintains a secondary NameNode apart from the primary or master NameNode. The secondary NameNode communicates with the primary NameNode periodically and copies the metadata to create checkpoints. If the primary NameNode crashes, these checkpoints on the secondary NameNodes are used for recovery.

Comparison between GFS and HDFS

HDFS is based on GFS and handles large data volumes consisting of a variety of data in a distributed framework. However, there are many differences between GFS and HDFS. Some of the key differences are summarized in Table 4.2.

HBase

To handle real-time, random huge-volume datasets, the Hadoop framework uses a non-relational and scalable distributed data management system that works on top of HDFS. Apache HBase provides Bigtable-like capabilities. HBase provides a fault-tolerant way of storing sparse datasets. The HBase provides the following features for effective real-time big data management (Apache HBase TM Reference Guide, 2008):

- Supports high linear and modular scalability
- Consistent read and write operations
- Automatic and configurable splitting of large tables into blocks
- Robust
- Supports real-time queries using block caching and Bloom Filters

Table 4.2 GFS vs HDFS

<i>Parameter</i>	<i>GFS</i>	<i>HDFS</i>
Proprietary	GFS is a proprietary file system solely used by Google.	HDFS is an open-source file system which anyone can use.
Chunk size	In GFS, files are divided into fixed sized chunks of 64 MB to be replicated and stored on chunk servers.	In HDFS, files are divided into fixed sized blocks of 128 MB to be replicated and stored on DataNodes.
Communication protocol	GFS uses TCP connections to perform communication between chunks and clusters. Pipelining is used for data transfer.	TCP/IP protocol. Also uses Remote Procedure Call (RPC) for external communication between blocks and clusters.
Cache management	Cache metadata are saved in client's memory.	HDFS uses distributed cache files which can be private or publicly accessible.
Database	BigTable.	HBase.

Comparison between Bigtable and HBase

Apache HBase is based on Google's Bigtable. Both are types of non-relational data storage for handling large volumes of datasets. However, there are some differences between Bigtable and HBase which are summarized in Table 4.3.

Table 4.3 Bigtable vs HBase

<i>Parameter</i>	<i>Bigtable</i>	<i>HBase</i>
Proprietary	Bigtable is a proprietary system.	HBase is an open-source project.
Installation	Available only as a cloud service from Google.	Uses Apache Hadoop's HDFS as underlying storage and can be installed on any environment.
Concept	Bigtable is a three-dimensional mapping database that combines row key, column key and timestamp into a single byte array.	HBase is a column-oriented database.
Data integrity	Uses CRC checksum.	Uses checksum feature of HDFS.
Security management	Bigtable security relies on Google's Cloud Identity and Access Management.	Apache HBase uses Access Control Lists.
Timestamp	Timestamps in microseconds.	Timestamps in milliseconds.

Comparison between HDFS and HBase

HBase is a top-level Apache project which runs on top of the HDFS. Some of the key differences between HDFS and HBase are summarized in Table 4.4.

Table 4.4 HDFS vs HBase

<i>Parameter</i>	<i>HDFS</i>	<i>HBase</i>
Concept	A file distribution system with rigid architecture.	HBase data storage runs on top of HDFS and supports dynamic changes.
Access method	Only sequential read/write operations. HDFS is based on write once read many times.	Random access using hash table. Supports random read and write operations into file system.
Data processing	Supports offline batch processing.	Efficient for real-time processing.
Latency	Provides high latency for access operations.	Provides low latency access to small amount of data.

Data Processing Component

MapReduce

Hadoop MapReduce is the main processing component of the Hadoop framework. It is based on the original MapReduce algorithm developed by Google and facilitates the distributed parallel processing of data. However, Google MapReduce can only work with the GFS file system whereas Hadoop MapReduce is a part of the Apache open-source project and is therefore used in many different architectures. Hadoop MapReduce has the ability to process large volumes of data, both structured and unstructured, and to manage very large data files in parallel by dividing the task into a number of independent jobs (MapReduce Tutorial, 2022).

MapReduce uses the divide and conquer approach for processing big data. It splits a task into multiple sub goals and processes them concurrently on different machines. Map

and reduce are the two stages in a MapReduce as shown in Figure 4.3.

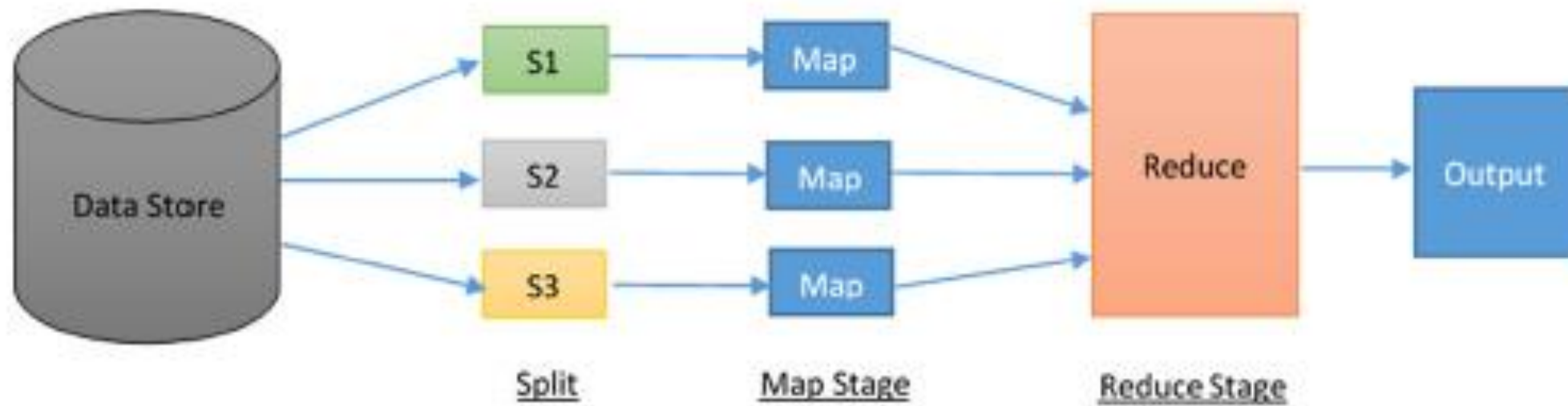


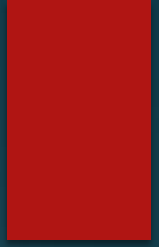
Figure 4.3 MapReduce architecture.

- **Map stage:** Each map task executes different operations like filter, group, shuffle and sort on a split of input data on different machines simultaneously and generates a key-value pair.
- **Reduce stage:** The output of the map stage is acted upon by the reduce task where operations like aggregation and summarization are performed to generate the final result to be stored in HDFS.

YARN

YARN is the acronym of “Yet Another Resource Negotiator” and is a resource management component in Hadoop just like an operating system and works on top of the HDFS. In Hadoop 1.0, a JobTracker process was responsible for resource management, scheduling and the monitoring of job processing. Another process called TaskTrackers was used to perform individual map and reduce tasks and report back on their progress. Because this resource allocation and management was centralized at JobTracker, it created a performance bottleneck. Hence, YARN was created in Hadoop 2.0 to decentralize the task and alleviate this issue. The main role of YARN is to manage and allocate cluster resources and schedule the jobs with effective load balancing. In addition to this, it is also

responsible for monitoring and the deployment of security controls. YARN assigns which tasks should be carried out by each node in the cluster and distributes the system resources necessary for each application that is running within a Hadoop cluster (Apache Hadoop YARN, 2022). YARN is made up of the following components as shown in Figure 4.4:



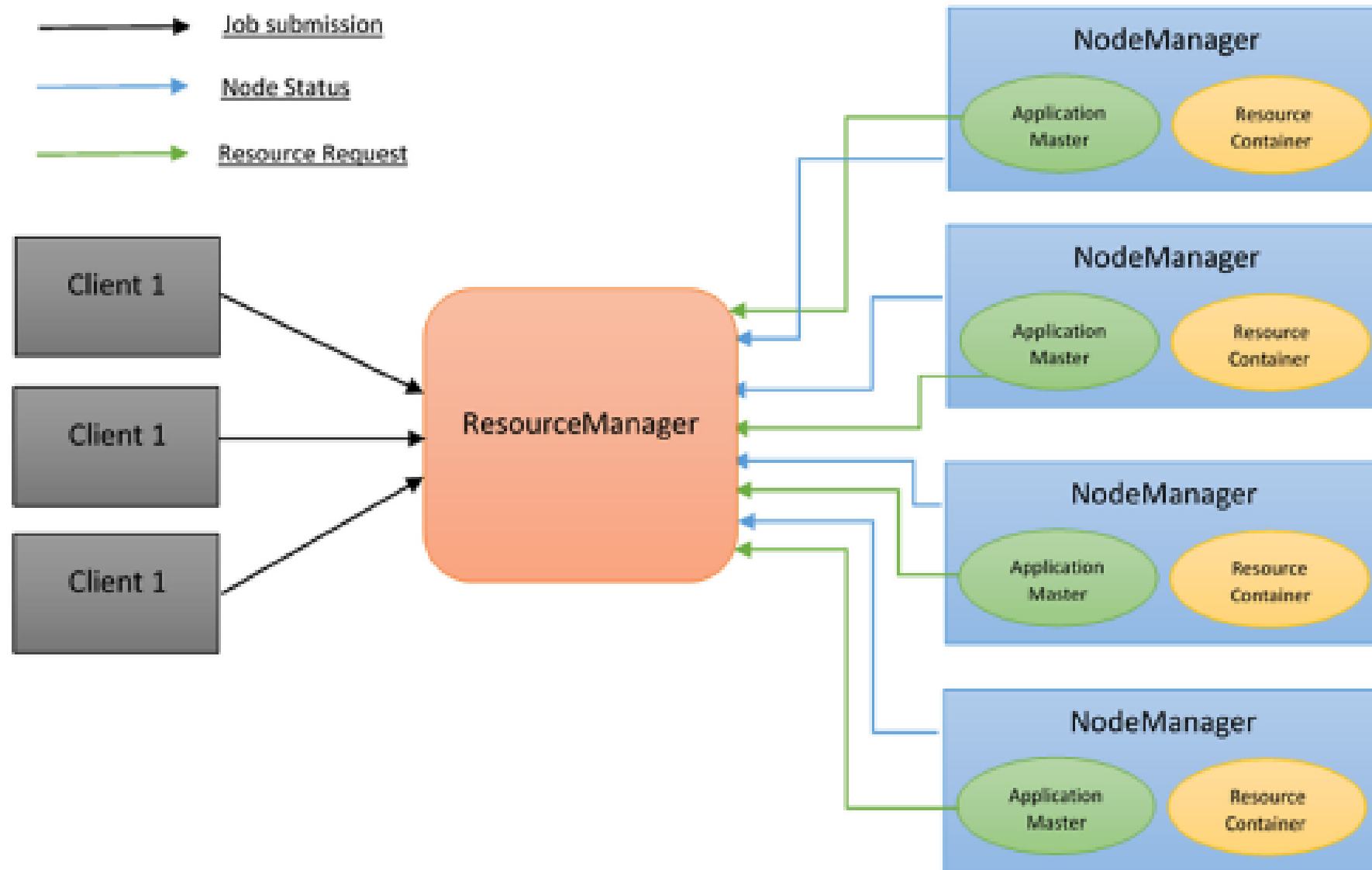


Figure 4.4 YARN architecture.

- **ResourceManager:** Jobs from the users are submitted to the ResourceManager which schedules these jobs and allocates resources to them.
- **NodeManager:** On every DataNode, NodeManager is installed which acts as a monitoring and reporting agent of the ResourceManager. These NodeManagers are responsible for monitoring resource usage including processing units, memory, disk storage and network capacities and reporting back to the ResourceManager.
- **ApplicationMaster:** Each application has a framework-specific library to negotiate resources based on the job request from the ResourceManager and work with the NodeManager to execute and monitor tasks.
- **ResourceContainers:** Each NodeManager has a component called the containers which contain a collection of resources including processing units, memory and disk storage. The ResourceContainers are controlled by NodeManagers and are used to assign the system resources allocated to individual applications.

Features of YARN

- Dynamic allocation of system resources
- Enables Hadoop real-time data processing along with MapReduce batch processing
- Enhanced resource utilization and application performance
- Supports various scheduling algorithms
- Reservation feature enables users to reserve cluster resources in advance
- Increased scalability with support from 10,000 to multiple tens of thousands of nodes

Data Access Component

Hive

Hive is an ETL and data warehousing tool developed by Apache. Hive makes it easier to query and manage massive datasets that are stored in multiple locations. Hive supports the summarization, querying and analysis of unstructured and semi-structured data in the Hadoop Ecosystem. The read, write and update operations on massive datasets in a distributed environment are performed through Hive Query Language (HQL) that is similar to Structured Query Language (SQL) and translates queries into MapReduce jobs automatically.

Components of Hive

- **Hive command line interface:** To execute HQL commands Hive provides a command line interface.
- **JDBC/ODBC driver:** For database connectivity, Java Database Connectivity (JDBC) and Object Database Connectivity (ODBC) drivers are used.

Hive Features

Some of the key features of Hive are:

- High scalability
- Real-time processing as well as batch query processing, i.e. interactive query processing
- Compatibility with each and every SQL primitive data type
- Support for user-defined functions (UDF) along with pre-set functions
- Automatic translation of queries into map and reduce functions through HQL
- Efficient extensive sequential scans

Limitations of Hive

- Not effective where fast response times are crucial because queries may encounter unusually high delays (up to several minutes).
- Hive is a read-based system and not suited for jobs typically involving many write operations.