Lecture Notes

An Introduction to Hadoop and MapReduce Programming

In this module, you will learn about Hadoop, which is an open-source framework used for storing and processing big data on clusters of commodity hardware, as well as its two major components that are Hadoop Distributed File System (HDFS) and MapReduce. In the first part of this module, you will learn about Hadoop and its various components and features. You will get a brief introduction of Distributed Systems. You will then learn about Google File System (GFS) and MapReduce developed by Google, which later on inspired the development of Hadoop. After this, you will be introduced to the second version of Hadoop and its components. You will then learn how task processing is done in Hadoop and about some tools commonly used in the Hadoop ecosystem. In the following session, you will understand in detail about the Storage component of Hadoop, i.e., HDFS. You will get an idea of how files are stored in HDFS and some of the considerations made in HDFS. Next, you will learn how you can interact with HDFS and also some basic commands to navigate the Hadoop clusters. You will also get an idea of the Read and Write operations in HDFS and its features and limitations. Finally, you will learn about the Processing framework component, MapReduce. In this session, you will understand the working of MapReduce and its various components as well as coding MapReduce programs. You will then learn about Hadoop Streaming and coding on the EMR instance for building and running MapReduce jobs on the HDFS cluster. Subsequently, you will learn about two important components of MapReduce: Combiner and Partitioner. Finally, we will discuss how jobs are scheduled in MapReduce and how it maintains fault tolerance.

An Introduction to Distributed Systems

A Distributed System is one in which several independent computers are connected to each other over a network via middleware to look like a single machine. The computers in the network communicate with each other to carry out tasks.

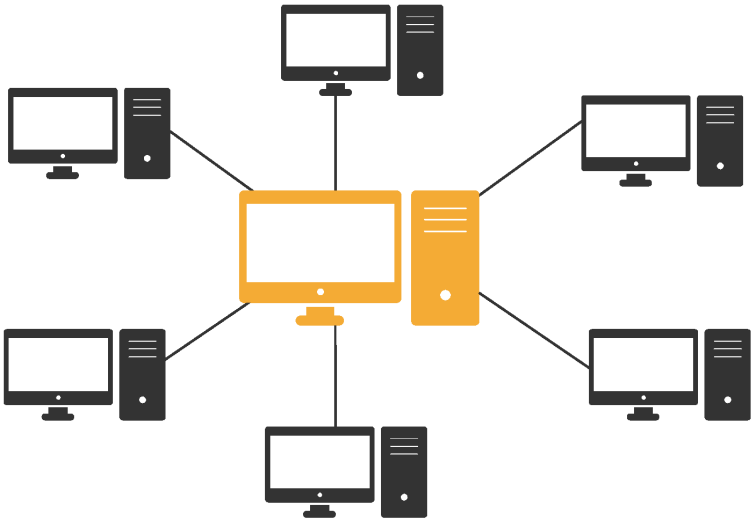

Figure 01: Distributed System Cluster

One of the major aspects of Distributed Systems is that the data being stored cannot be handled by a single system, both for computation and storage.
Data is thus divided into multiple chunks known as blocks and stored across multiple servers.

DISTRIBUTED COMPUTING

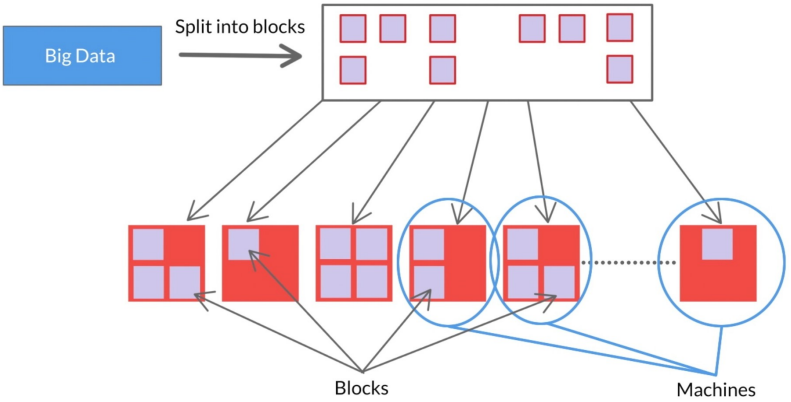

Figure 02: Files Stored in the Form of Multiple Blocks

These chunks are created such that they can be easily managed by a single machine for storage and computation.

Now, let's have a look at some of the primary challenges faced by Distributed Systems:

- **Performance** - System should include challenges like communication fault delay or computation fault delay.
- **Fault tolerance** - System must be able to tolerate faults and function normally.
- **Scalability** - System must remain effective even under heavy load.
- **Security** - System can face probable threats like information leakage, integrity violation, denial of services, and illegitimate use.
- **Concurrency** - Shared access to resources must be made available for the required processes.
- **Migration** - Tasks should be allowed to move within the system without affecting other operations.
- **Load balancing** - Load must be distributed among available resources for better performance.

An Introduction to GFS and MapReduce

Google File System (GFS or GoogleFS) is a proprietary distributed file system created by Google for storing and processing large amounts of data in multiple commodity machines within large clusters, ensuring reliability, scalability and efficiency.

Following are some of the considerations made while developing GFS :

- Use of commodity hardware due to the low cost
- Easy horizontal scalability without interrupting the rest of the system
- Failure of commodity hardware each time
- Fault tolerance and automatic error recovery crucial to maintain constant data availability for clients

GFS has resolved many of the challenges and problems faced by traditional Distributed Systems:

- Replication helps maintain high availability of data and fault tolerance
- Automatic and efficient data recovery
- High aggregate throughput
- Each chunk server verifies the integrity of its copies using checksums and is itself constantly monitored by GFS master
- Modularity allows GFS to easily expand to account for increased loads
- Master Node ensures maintenance of load balancing

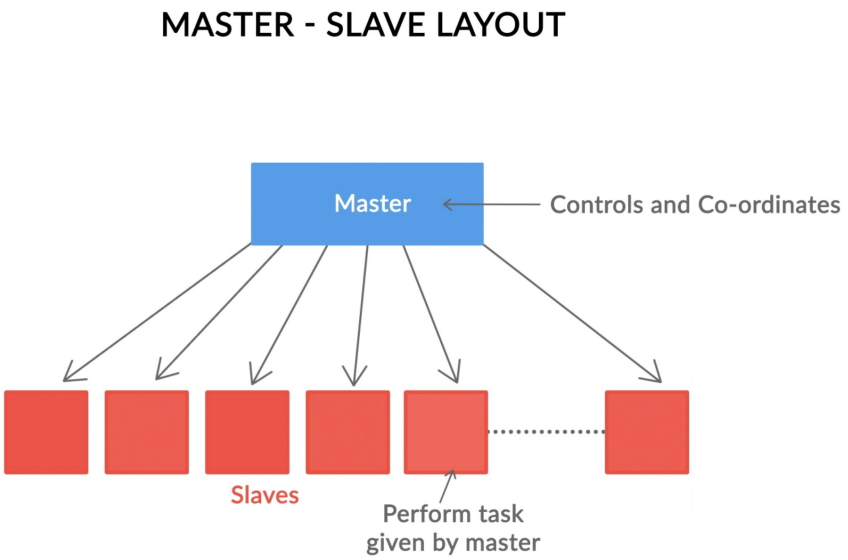

Figure 03: Master-Slave Architecture of GFS

GFS has a master-slave architecture, wherein the master coordinates access and keeps metadata, and the slave, also known as chunk server, stores the data. This architecture includes:

- **Master Node** coordinates the tasks of the chunk servers and also contains the metadata of the chunk servers and their files. It keeps a check on the health of the various slave nodes under it. It also ensures load balancing . Copies of the master node, which are meant to keep a regular log of it, may also exist to ensure that the system does not fail if the master node fails.
- **Slave Nodes** or **Chunk Servers** are the nodes where the actual files are stored. Data is generally split into fixed-sized chunks, which is 64 MB in the case of GFS.

Apart from GFS, Google has made another major development, which helped shape Hadoop later on, especially its computational component called **MapReduce**.

In 2004, Google introduced **MapReduce** in their paper **'MapReduce: Simplified Data Processing on Large Clusters'. MapReduce** is a programming model and an implementation for processing and generating large data sets. It was designed specifically to work with a cluster-based distributed file system such as the GFS.

The Apache Software Foundation used MapReduce in their '**Nutch**' Project, which was highly scalable and supported unstructured data. MapReduce jobs were able to withstand hardware failures, ensuring fault tolerance.

Google later reported its implementation of GFS in a white paper titled '**The Google File System'** and defined it as a scalable distributed file system for large distributed data-intensive applications. The development of Hadoop started as a sub-project of Apache Nutch when the Apache community realised that the implementation of MapReduce and Nutch DFS could be used for other tasks as well. Finally, on 1 April 2006, Apache Hadoop was released.

An Introduction to Hadoop

Hadoop is an open-source framework used for storing and processing big data on clusters of commodity hardware. It partitions a large data set into smaller chunks, which are then stored in a cluster of machines. This helps in achieving parallelisation.

The two main components of Hadoop are as follows:

- **Hadoop Distributed File System (HDFS):**
  - This is the storage layer of Hadoop and written in Java.
  - Data is stored in commodity machines in a distributed manner, with multiple copies to ensure reliability.
  - Its architecture is inspired by GFS.
  - It is horizontally scalable.
  - The cost of setting up the system is quite low, as commodity machines are cheap and easily available.

- **MapReduce:**
  - It is a programming model that enables distributed big data processing.
  - It provides many libraries, which are collectively known as MapReduce.
  - It comprises two phases - Map phase and Reduce phase:
    - The tasks performed in the Map phase are called Map tasks.
    - The tasks performed in the Reduce phase are known as Reduce tasks.
    - Both tasks have dedicated slots.
  - MapReduce programs are natively written in Java, but these can also be written in Python, Ruby and C++.

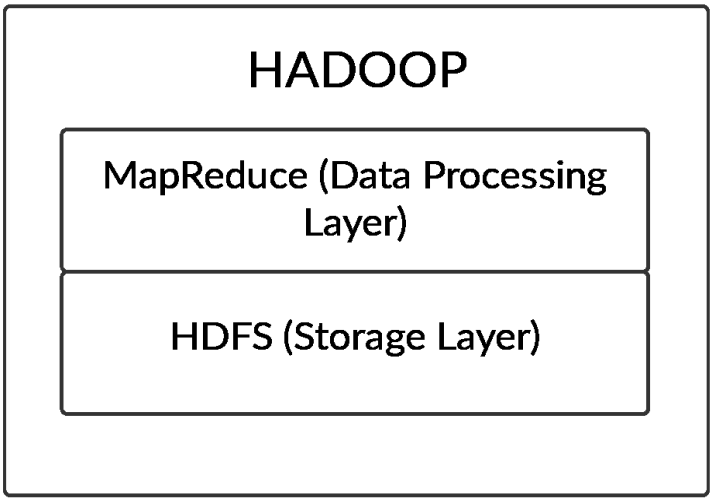

Figure 04: Components of Hadoop

Now, coming to HDFS, it mainly consists of three components:

- **NameNode:**
  - It is the master server, which runs on the Master Node in this configuration.
  - It is responsible for maintaining the metadata (data of the locations of file blocks across the cluster, ownership rights, etc.) of the different files present in the cluster.
  - It is also responsible for assigning work to the slave nodes as well as executing file system namespace operations such as opening and renaming files, and other modifications.
- **Secondary NameNode:**
  - It is responsible for maintaining the metadata of NameNode.
  - The NameNode then uses this metadata to update its own metadata.
  - The metadata present in the Secondary NameNode is used for implementing a new NameNode if the current NameNode fails.
- **Standby NameNode:**
  - Another component of HDFS, the Standby Namenode provides fault tolerance against the problem of single point of failure (SPOF) related to NameNode.
  - It provides automatic failover in case an active NameNode fails.
- **DataNodes:**
  - They are the slave nodes.
  - They are responsible for storing and processing data of the Hadoop cluster.
  - They provide access to data files when requested.
  - They also send heartbeat messages regularly to the NameNode to indicate they are alive.
- **HDFS Client:**
  - It acts as an intermediate command line tool for interacting with HDFS.
  - It has access to Java libraries needed for the various applications in HDFS.
  - When accessing the HDFS, it is this HDFS client which interacts with NameNode and DataNodes to get the job done.

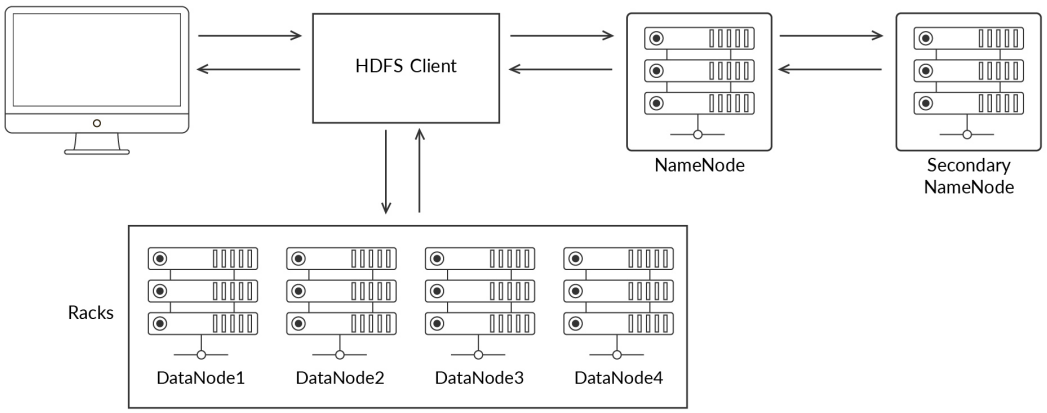

Figure 05: Components of HDFS

Hadoop 2.x

Briefly, the initial version of Hadoop, called Hadoop 1.x, had two main core components, HDFS and MapReduce. HDFS acted as the storage layer, while MapReduce was the processing and resource management layer.

Here, Job Tracker and Task Tracker are the daemons of MapReduce; these two are essentially the background processes that handle the requests for the services provided by the MapReduce layer.

- Job Tracker runs on the Master Node and handles jobs requested by clients, and then initiates separate tasks on the various DataNodes in the cluster.
- Task Tracker runs on the slave nodes and initiates the tasks assigned by Job Tracker. It also returns the status of the jobs running on the slave nodes to the Job Tracker.

Note: A 'Job' is a MapReduce operation or any other form of query that a client needs to perform on a data set. Each 'Job' is further partitioned into multiple small parts called 'Tasks'.

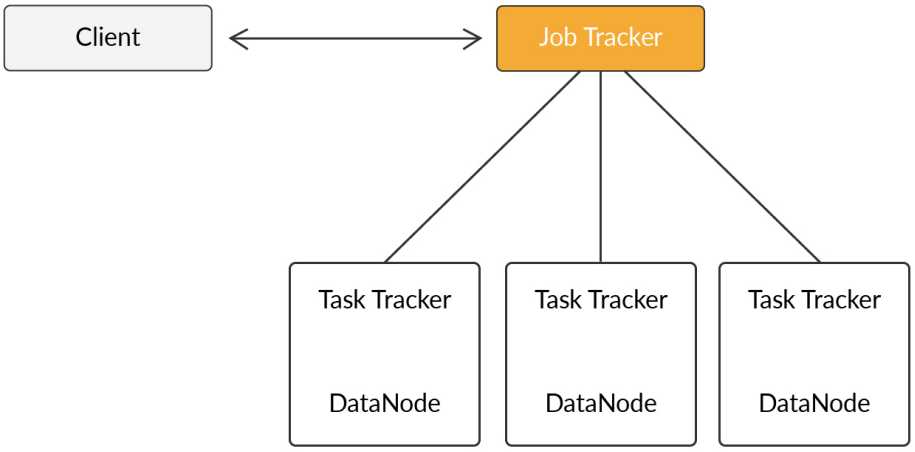

Figure 06: MapReduce Architecture in Hadoop 1.x

The diagram above clearly shows that Job Tracker is overburdened, as it has the responsibilities of managing and maintaining the overall resources, such as the various available CPUs and memory, configuration files, jar files, etc., of the cluster, as well as job scheduling and monitoring. Apart from this, making it an SPOF also affected its scalability, as the number of possible Map and Reduce slots depends directly on the Job Tracker.

The main motivation behind Hadoop 2.x was to solve the problems mentioned above. In Hadoop 1.x, if the Job Tracker fails, then the whole cluster would fail to communicate with each other. Hadoop 2.x fixed this limitation by introducing Yet Another Resource Negotiator (YARN), which focuses only on resource management.

Hadoop 2.x has the following core components: YARN; HDFS; and data-processing frameworks like MapReduce, Tez and Spark.

YARN: Also called Yet Another Resource Negotiator, YARN is responsible for cluster resource management. Some of its features, in brief, are as follows:

- It Reduces the load on MapReduce by allowing it to focus only on processing data.
- It allows other data processing frameworks, such as Tez and Spark, to process data.
- It ensures that resources are treated equally and can process any task.

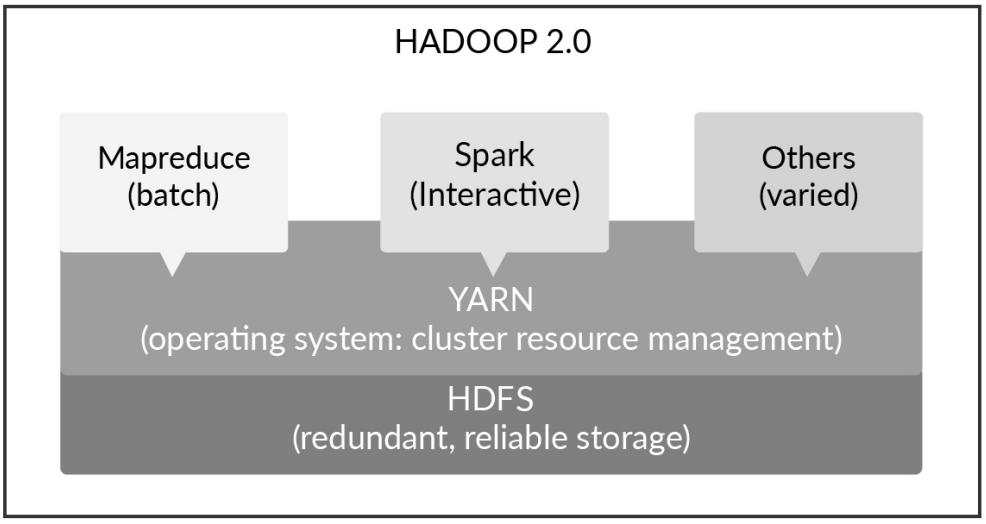

Figure 07: Components of Hadoop 2.x

Hadoop 3.x has also been released and has several improvements over Hadoop 2.x. However, in this course, we will only deal with Hadoop 2.x. Some of the improvements in Hadoop 3.x are as follows:

- It has a minimum runtime version requirement to Java 8.
- It supports erasure coding in HDFS.
- It introduced a more powerful YARN Timeline Service, i.e., v.2, with improved scalability and reliability.
- It has some other improvements such as support for more than two NameNodes and support for GPUs in YARN.
- It provides intranode disk balancing.

Hadoop 3.x has become a major milestone in the Big Data industry, and its improvements have received widespread appreciation in the community.

YARN

YARN was introduced in Hadoop 2.x mainly to split into separate components the responsibilities of the Job Tracker in Hadoop 1.x so as to ensure scalability as well as reliability of the whole system. This was of help as Hadoop no longer worked with fixed sets of slots for the Map and Reduce phases.

Note: YARN was still a single point of failure (SPOF; specifically, the Resource Manager) until Hadoop v2.4 was released. After this, a standby Resource Manager has been added to Hadoop with an Automatic failover support, making YARN fault tolerant.

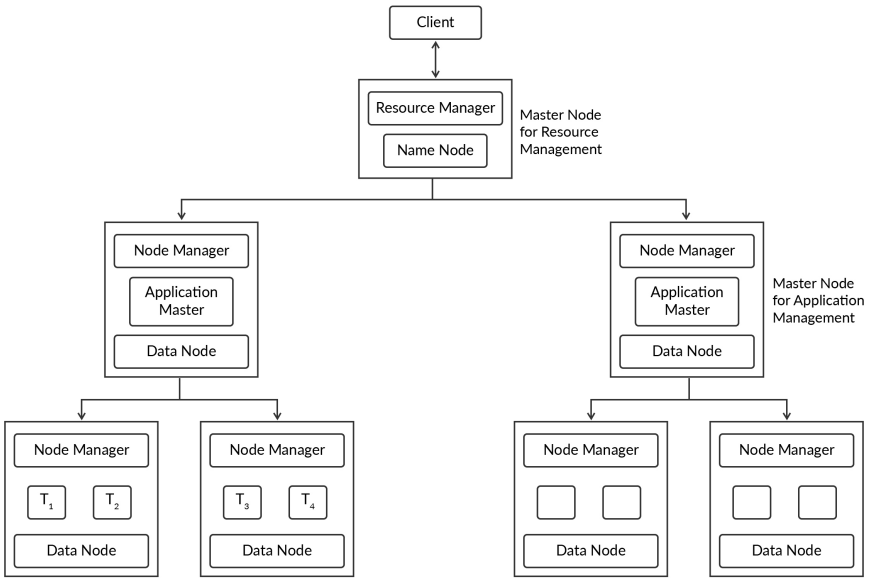

Figure 08: Architecture of YARN

Components of YARN:

- **Containers:**
  - As the smallest units of resources that can be used for processing data, they consist of different hardware components required for processing, for example, memory.
  - A single node can have multiple containers, each processing a single piece of work. Jobs can be divided into multiple tasks and containers are used to handle all these tasks in parallel.
- **Resource Manager:**
  - The Resource Manager resides at the Master Node.
  - It has two components:
    - Scheduler: It allocates resources to various running tasks but does not monitor or track the status of applications.
    - Application Manager: It handles job submission from the client, initialisation of job-specific Application Masters, and tracking their status.

- **Node Manager:**
  - It is the counterpart of the Resource Manager at the node level.
  - DataNodes and Node Managers run on the same system so as to ensure that the computations are done as close to the data as possible.
  - It launches and manages containers, in addition to monitoring resource usage at the node level.
  - It also sends heartbeat messages to the Resource Manager to share its health status.
- **Application Managers:**
  - They are responsible for monitoring jobs and reside at the slave nodes.
  - They are responsible for requesting containers from the Scheduler.
  - Every job has a dedicated Application Master, which also coordinates with one or more Node Managers to monitor the tasks being executed in their containers.

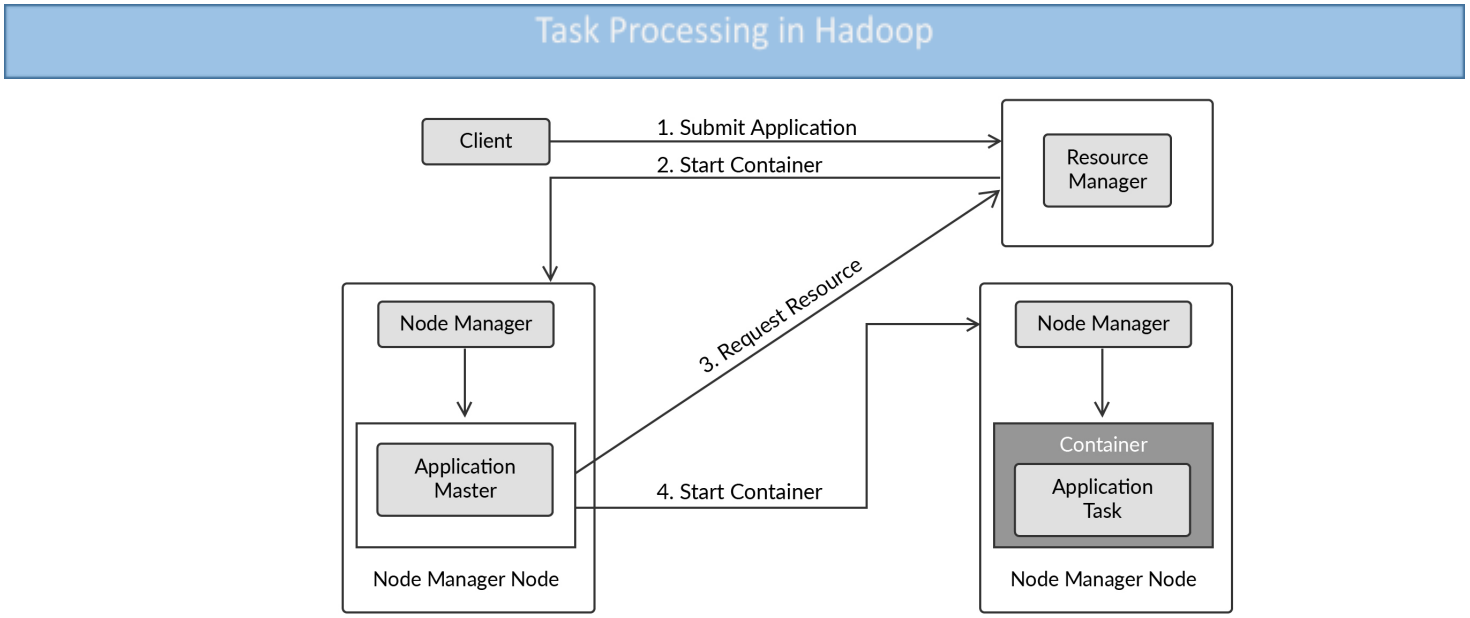

Figure 09: Task Processing in Hadoop

Task processing in Hadoop is carried out in the following steps:

- A job is submitted to the Application Manager in the Resource Manager.
- After receiving a job, the Resource Manager's primary task is to launch an Application Master for the job.
- The Application Manager assigns a container to a DataNode where a new Application Master can be initiated for the new job.
- As and when the Application Master is launched, it asks the Scheduler to provide resource containers, which would be needed for processing the job.
  - The container created has a fixed memory size, disk space and CPU cores assigned to it in accordance with the tasks for each job.
- The Application Master notifies the Node Manager about the DataNodes in which the job has to be processed and then requests the Node Manager to launch a container for the task.
- The job is then processed inside the containers in the node.

- The Application Master negotiates containers for launching new tasks for the job. It is also responsible for monitoring the progress of a job and its tasks, restarting failed tasks in new containers and, finally, reporting back the developments to the job's client.

- After the job has been processed, the Application Master proceeds to shut itself down and release its containers. The results are sent back to the client at this time.

What is noteworthy in this process is that the Resource Manager is responsible for continually checking the Application Master, so that even if it fails, the Resource Manager can restart it in a new container.

Tools for Hadoop

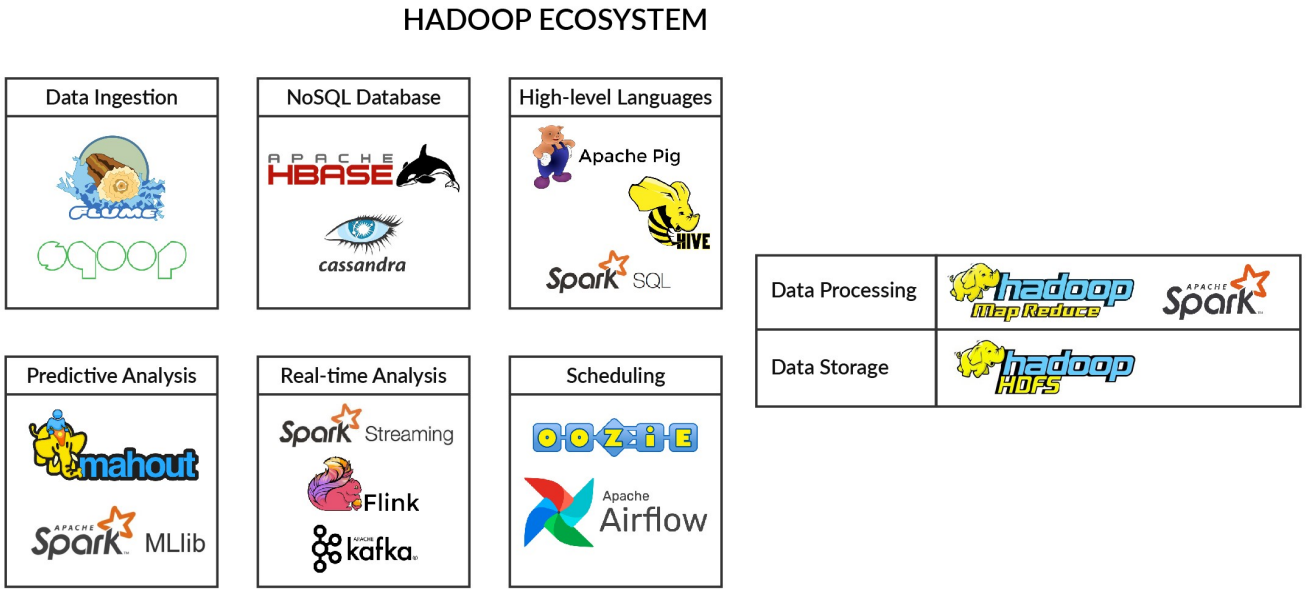

Figure 10: Hadoop Ecosystem

Hadoop Ecosystem has a large number of tools used for various purposes. Let's first discuss some of the most basic components of the Hadoop Ecosystem:

- **HDFS**: As the storage layer of Hadoop, HDFS offers a distributed, reliable, and scalable file system for storing big data.
- **MapReduce**: It is the legacy data-processing layer of Hadoop, which enables distributed big data processing and parallel programming in data processing.

Spark is another programming model that was supported after the release of Hadoop 2.x.

- **Apache Spark:** It is a unified computing engine with a set of libraries for parallel data processing on cluster systems and is used for processing large-scale big data.
- It can also process real-time data streams, unlike MapReduce.
- Spark can also run applications on memory, enabling some tasks to run up to 100x faster than MapReduce.

Data Ingestion Tools:

- **Flume**: Apache Flume is a system for collecting, aggregating, and transporting huge volumes of streaming data such as log files, events, and many more, from various sources into the HDFS.
- **Sqoop**: It gets its name from SQL-to-Hadoop and is a command-line-based application used for transferring data between relational databases, such as Oracle and MySQL, and HDFS.

NoSQL Databases:

- **HBase**: Apache HBase is a popular and highly efficient column-based NoSQL database built on top of HDFS. It offers horizontal scalability and allows you to perform operations such as read/write on real-time data from large data sets.
- **Cassandra**: Apache Cassandra is an open-source distributed, wide-column store, NoSQL Database management system. It is designed to easily handle large amounts of data across multiple commodity machines, while also providing high availability and maintaining fault tolerance by ensuring that the system does not have an SPOF.

High-Level Languages:

- **Pig**: Apache Pig is a high-level data flow platform for executing Hadoop MapReduce programs. It can help in writing MapReduce jobs in a fraction of the code length in standard Java.
- **Hive**: Hive is an ETL and data warehousing tool built on top of HDFS. It provides an SQL-like interface to perform functions such as data summarisation and analysis as well as to query databases and file systems in Hadoop.
- **SparkSQL**: It is a Spark module for structured data processing and brings native support for SQL to Spark.

Predictive Analysis:

- **Mahout**: It is a distributed linear algebra framework that is primarily focussed on providing an implementation of scalable and distributed machine learning (ML) models linear regression, clustering, and more.
- **SparkML**: This tool provides a uniform set of high-level APIs that help in creating and tuning practical ML pipelines.

Real-Time Analysis:

- **Spark Streaming**: It is an extension of the core Spark API, which enables scalable and fault-tolerant stream processing of live data.
- **Flink**: It is an open-source stream-processing framework.
- **Kafka**: Apache Kafka is an open-source stream-processing software platform developed by LinkedIn.

Scheduling:

- **Oozie and Airflow**: These are scalable and reliable server-based workflow scheduling systems for managing Hadoop jobs. These tools can be used for pipelining all types of programs in the desired order and can also be scheduled for the programs at a particular time.

File Storage in HDFS

In HDFS, files are stored as fixed-size chunks or blocks. Each file is divided into multiple fixed-size blocks, and typically, the last block does not occupy the full space of a block.

Note: Usually, the block size, which is the memory occupied by one block, in a PC file system (Windows, MacOS, Linux, etc.) is 4 KB.

Ideally, these blocks should be placed continuously to Reduce the I/O overhead, but this is not always possible. So, file systems usually maintain the metadata of the files along with their block IDs and respective locations so as to minimise the seek time.

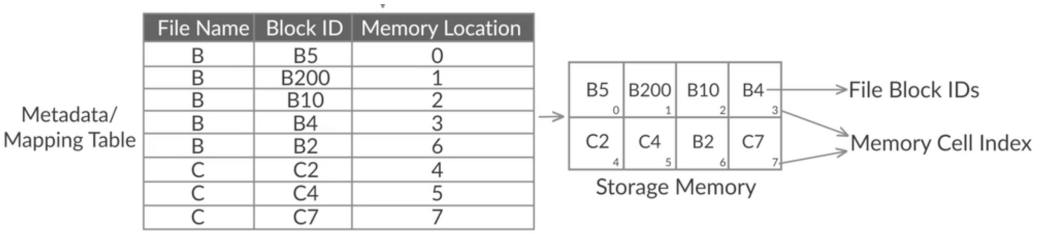

Figure 11: Metadata of Files in HDFS

Similar to the local file system on a single disk, files in HDFS are split into blocks. The default block size in HDFS is 128 MB. The main difference between the local file system and HDFS is that in HDFS, different blocks are stored in different machines as separate files; hence, the last block in HDFS does not always occupy the full storage of a block.

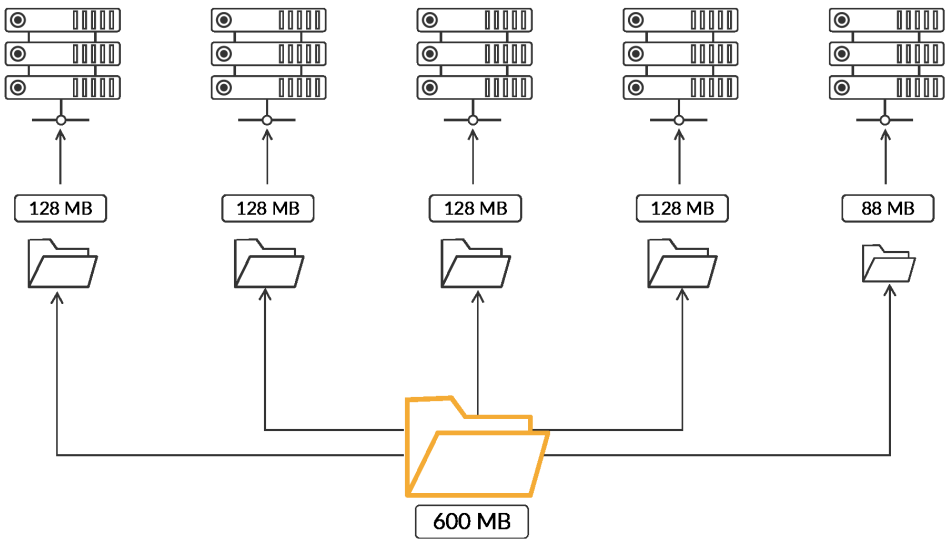

Figure 12: Storage of Files in the Form of Blocks

HDFS also prefers large-sized blocks because:

- It helps minimise the cost of seeks.
- Fewer separate TCP connections are made to different machines.
- Overall size of the metadata is significantly Reduced.

- Location information of all blocks can be cached by the client.

A large-sized block comes with its own set of limitations, as it limits the parallel computing aspect of HDFS as a lesser number of overall blocks will mean lesser degree of parallelisation possible for computational purposes, thereby reducing the overall system throughput.

A very important aspect of HDFS is maintaining fault tolerance. As commodity machines are used, the failure rate is high. HDFS tackles this by making additional replicas. By default, HDFS keeps two replicas of a block to maintain the fault tolerance. This way, every block of a file has a total of three copies in the file system. The replication factor here would refer to three.

This specific number of replicas is ideal because it ensures optimal storage use and maintenance of an appropriate availability, as you can see in the table below.

Number of Replicas	Failure Probability of a Machine	Probability of All Machines Failing Simultaneously	Probability That Data is Not Lost or At Least One Replica is Available	Marginal Improvement
1	0.05	0.05	0.95	-
2	0.05	0.0025	0.9975	0.0475
3	**0.05**	**0.000125**	**0.999875**	**0.002375**
4	0.05	0.000006250	0.999993750	0.000118750
5	0.05	0.000000313	0.999999688	0.000005937
6	0.05	0.000000016	0.999999984	0.000000297

Additionally, the number of replicas is odd to avoid any confusion in the consistent state of the replicas. If the number of total replicas is even then there is a possibility of the presence of an equal number of conflicting replicas. In HDFS, if there is an inconsistency between the various blocks, then the replicas that are in majority among the replicas are considered consistent. So, as you can see, an odd number of replicas is preferable.

Figure 13: Consistency with Odd Number of Replicas

Figure 14: Consistency with Even Number of Replicas

The actual file transfer process is as follows:

- For a file to be extracted, you must have information about the number of blocks in the file and the location of each block.
- The NameNode is responsible for maintaining the file hierarchy, metadata of different files in the file system, and consequently, the location of blocks and directories in the file system.
- This information is maintained in the main memory for faster access, and without the NameNode, no file can be accessed.
- The DataNode stores the actual file blocks, and along with the NameNode, it constitutes the entire file system's functions.

Write Operation in HDFS

HDFS has many operational limitations. It allows clients to create and store files but does not allow them to modify these files. Until the recent versions of Hadoop, HDFS did not even allow clients to append data to the last written block until it reached the block size. This is to maintain concurrency between the different replicas of the blocks.

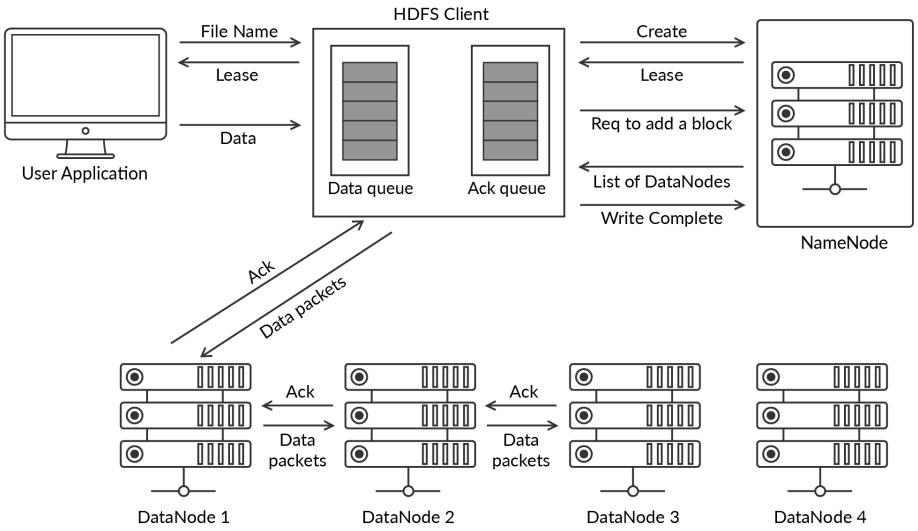

Figure 15: Write Operation Procedure

Details of steps that HDFS takes for a file write operation:

- Initially, the client inputs the filename to the HDFS client.
- HDFS client sends a Remote Procedure Call (RPC) to the NameNode to create a file. (A program can use the RPC protocol to request a service from a program present in another computer or network without any prior knowledge of the underlying network.)
- The NameNode then creates a file and grants a lease to the client. This lease is vital because it indicates that the client now has the rights to write to the file.
- HDFS client proceeds to buffer the data until it reaches the size of the block. When the limit is reached, an RPC request is sent by the HDFS client to add a block to the file created.

- The NameNode then returns a list of available DataNodes, where the HDFS client can write data. These DataNodes collectively constitute a pipeline.
- The HDFS client splits the data into fixed-size packets and adds them to a Data Queue.
- The data packets are then sent to the first DataNode after establishing a connection. This DataNode also has an acknowledgement queue (Ack Queue), where the packets are moved while being written by the HDFS client.
- Whenever a DataNode successfully receives the packet, it acknowledges the same to the HDFS client. For all DataNodes, this happens in the reverse direction of the pipeline. The HDFS client waits for all the acknowledgements, and only after all of them are received, the packet is removed from the Ack Queue.
- After the packet is removed from the Ack Queue, the HDFS client sends a Write Complete RPC message to the NameNode.
- The NameNode then checks for the minimum number of replicas present; the default is three. If yes, it will commit the block and return success to the HDFS client; otherwise, it will follow the same steps, starting with sending an RPC for adding a block.

In case of errors during this process, HDFS takes the following steps:

- The HDFS client closes the current pipeline and creates a new one, ignoring the failed DataNode. The packets still present in the Ack Queue are removed and added to the front of the data queue.
- Finally, a new identity is assigned to the current block that is being written to, and the same is communicated to the NameNode. In this way, partially written blocks on the failed DataNode can be deleted after they are recovered.

A noteworthy point here is that an HDFS client does not need to wait for all the acknowledgements and can redefine the pipeline if it finds an error while transferring data to other DataNodes. This is done to ensure that at least one of the replicas created exists so that a minimum of one DataNode is in the pipeline.
It is also important to note that unless the HDFS client sends a Write Complete RPC message, the NameNode does not commit that block.

Rack Awareness in Hadoop

Rack Awareness is a concept in Hadoop that ensures fault tolerance and data locality.

Let's assume you have a Hadoop cluster with multiple DataNodes (in multiple racks) in three data centres at different geographical locations globally. The replication factor is the default (three).
(A rack is a collection of multiple DataNodes that are physically stored close together and are connected to the same network switch.)

In rack awareness, the NameNode ensures the following:

- Fault tolerance
- Data locality

In order to maintain fault tolerance, the NameNode ensures that the replicas are stored in different racks (within a single data centre) because the probability of all these racks being down simultaneously is low (every rack has a different network switch). But what if the connectivity to the entire data centre (where all these racks are placed) is compromised? To avoid this, the NameNode distributes the replicas over different data centres, reducing the probability of data loss to a significantly small number.
Let's say the NameNode stores three replicas in three different data centres. To maintain the fault tolerance, the NameNode cannot be made to store the three replicas in three different data centres, as this would be inefficient and a waste of bandwidth. This is because every time a block is written, it has to be written on three data centres present in three different geographical locations (recall HDFS pipeline write-up). Similarly, in the case of a data read operation, in order to ensure the consistency of a data block, the NameNode has to comprehend replicas from all the three different data centres.

To optimise this, the NameNode maintains by default the second and third replicas in a single data centre (within a single rack), which means the three replicas are stored in two data centres (in two racks). The probability of the failure of these two racks is still low enough to be deemed satisfactory. In this way, the cluster achieves both data locality (saving network I/O) and fault tolerance.

Read Operation in Hadoop

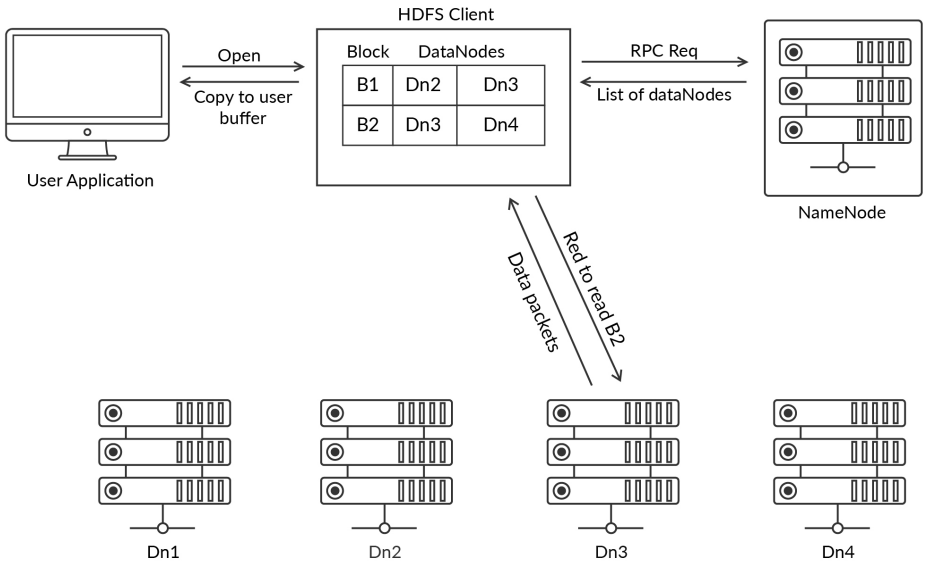

Figure 16: Read Operation Procedure

HDFS follows the below steps to read a file:

- To read a file, the client first opens the file and starts reading it using the API provided by the HDFS client.
- For this, the client's user application will first open a file by calling its API.

- The HDFS client then sends an RPC request to the NameNode, which, in turn, retrieves a list of DataNodes for the first few blocks. This list is then sorted according to the proximity to the client.
- The HDFS client retrieves the data block by block. For each block, the client uses sockets to connect to the first DataNode in the list and then sends the information such as block ID, offset and length of data to be read to the connected DataNode.
- The data is then streamed back to HDFS by the DataNode.
- When an HDFS client receives the packets, it does not directly send them to the client. It first verifies the checksum, and if there is an error, it informs the NameNode and then contacts the next DataNode.
- In this way, the HDFS client sends RPC messages to the NameNode to fetch the addresses of the DataNode, where the remaining blocks are present, to retrieve them.

Features and Limitations of HDFS

The main features of HDFS:

- **Cost**: HDFS is composed of commodity machines, which are easily available and cheap, enabling horizontal scalability.
- **Data versatility**: HDFS can handle a variety of data, be it structured, semi-structured or unstructured, in large quantities.
- **Fault tolerance**: HDFS has inbuilt fault tolerance measures in place, such as having multiple replicas and checksums to ensure the data is always available to multiple clients concurrently.
- **Performance**: HDFS provides a high throughput owing to parallel processing.
- **Data locality**: HDFS helps in reducing the network load, thereby improving the overall performance.

Some limitations of HDFS:

- HDFS does not support data access with low latency. It cannot handle data blocks directly because the NameNode has the block location information. It also cannot handle parts of a block and needs to have access to the whole block even if it has to work with only a small part of the block.
- HDFS is designed to be used with large blocks and not optimised for handling small files. Since it stores metadata for each file, the NameNode will have to store a lot more metadata for multiple small files as opposed to one large file.
- HDFS does not support arbitrary modifications to a file and only supports writing at the end of a file. It also does not allow multiple writers to write on a single file.

An Introduction to MapReduce Framework

MapReduce is the data processing layer of Hadoop. It is a programming model or method of processing a large amount of data.

A MapReduce program will solve the problem mentioned in the video in two phases: Map phase and Reduce phase. This can be achieved by writing separate scripts for both these phases. The Mapper code transforms the data into a key-value pair, while the Reducer code aggregates the processed data and provides an output that is specified by the user in the Reducer code.

Hadoop Streaming

Java is the native language used for writing MapReduce jobs. However, you are not restricted to Java for performing MapReduce jobs in Hadoop.

Hadoop provides an inbuilt API called Hadoop Streaming, which allows you to use scripts written in any language as Mappers and Reducers and, thus, perform MapReduce jobs.

Hadoop Streaming utilises Unix standard streams to accept inputs and generate outputs, which it then uses as an interface between the HDFS and your programs.

Both Mappers and Reducers basically become executables that read the input from stdin line-by-line and then generate the output to stdout.

Note: Outputs from Mapper as well as Reducer are in the form of a string and not a tuple. These scripts also take input from Standard Input.

Note that for each Mapper or Reducer task, a separate instance of the executables is initiated.

The Combiner

Combiner is a component of MapReduce which can be useful in certain applications such as finding the average in a data set. It can perform a Reducer's job before the Reducer phase.

A Combiner helps in significantly reducing the network I/O and bandwidth, as calculations are performed in the DataNodes itself, where the data is present.

An important point to note here is that although a Combiner works in the same way as a Reducer, the Combiner has a different way of generating output for further processing by the Reducer.

Also, a Combiner cannot be used in tasks like finding the median of a data set, as this operation requires the whole data set to be present in the same place at the same time.

The Partitioner

Partitioner is another useful component of MapReduce that can be used to partition the key-value pairs such that the values for each key will be partitioned together and then have the same Reducer allocated to each partition.

The partition class used in the Partitioner command decides how the key-value pairs are partitioned. The partitioner acts after the Map phase but before the Reduce phase.

It helps in reducing the time a Reducer takes to perform a job, as similar data are placed together during the processing phase, which enables faster processing.

Hadoop uses the Hash Partitioner class by default. Hash Partitioner calculates the Hash of key to decide the partition for a key-value pair. It implements a simple Hash function with the key as an argument. Each key gets a Hash value and the key-value pair is assigned a partition based on the result.

Job Scheduling and Fault Tolerance

In Hadoop 2.x, the execution process is handled in the context of MapReduce 2, also known as YARN or 'Yet Another Resource Negotiator'. YARN also follows the Master-Slave architecture, where the master is the Resource Manager and the slaves are Node Managers. All the instances of Node Managers usually run on the servers running the DataNodes. The Node Manager regularly sends heartbeat signals to the Resource Manager, indicating that it is 'alive'. While the Resource Manager is responsible for managing the overall resources of the cluster, the Application Masters handle the scheduling of tasks that run on the slave machines. Application Masters, like Node Managers, also regularly send heartbeat signals to the Resource Manager.

HDFS has multiple DataNodes and a single NameNode that manages these DataNodes. Map tasks and Reduce tasks are specified by the user and then created by the NameNode.

The NameNode is also responsible for maintaining a queue for the different Map and Reduce tasks along with their status. This status can be:

- Idle,
- Completed, or
- Processing.

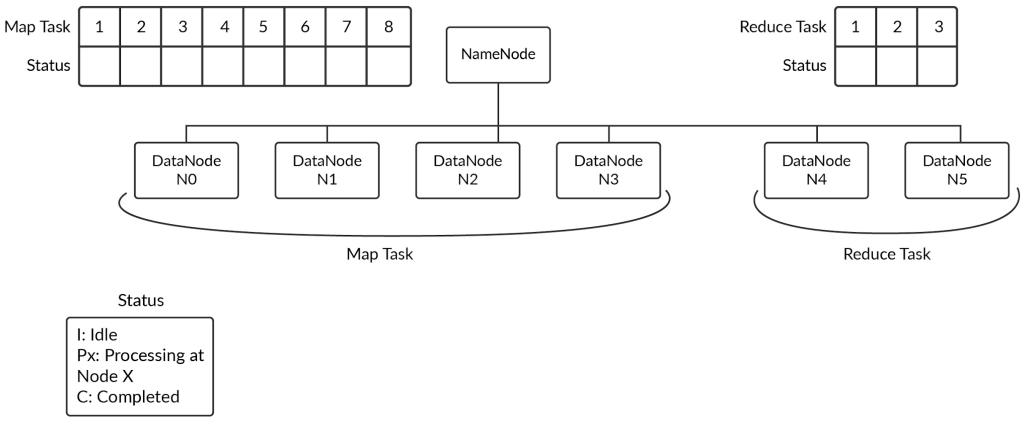

Figure 17: Job Scheduling in MapReduce

The processing begins when the Map tasks are allocated their DataNodes. The available DataNodes are assigned the idle Map tasks for processing.

If a MapReduce task containing a DataNode fails, then the tasks that are currently processing as well as the tasks that have been completed are reset and their status changed to idle. This is because the intermediate key-value pairs are stored in these DataNodes. Hence, they will not be available after the DataNode fails. All these Map tasks are then added back to the queue. Once the DataNodes become available, these tasks are sent to them for processing.
The NameNode also informs the Reduce tasks about the updated DataNodes from where it has to receive the outputs of the corresponding Map tasks.

If a DataNode corresponding to a Reduce task fails, then the process is simple, as the results of the completed Reduce tasks are stored in the HDFS file system. As these results are still accessible, resetting these tasks is not required. The already processing tasks are set to idle and are reassigned to healthy DataNodes when they become available.

Note that the number of Map tasks is set higher than the number of DataNodes in order to optimise the processor utilisation. However, the number of Reduce tasks is set low, as a higher number of Reduce tasks would result in the creation of intermediate files and partitions on the local file system corresponding to the number of Reduce jobs, which is undesirable.

Also note that the program is moved to the DataNodes rather than the other way round. The fault tolerance is maintained by reassigning the tasks from the failed DataNodes to the healthy DataNodes.

Disclaimer: *All content and material on the upGrad website is copyrighted material, belonging to either upGrad or its bona fide contributors, and is purely for the dissemination of education. You are permitted to access, print, and download extracts from this site purely for your own education only and on the following basis:*

- *You can download this document from the website for self-use only.*
- *Any copies of this document, in part or full, saved to disc or to any other storage medium, may be used for subsequent, self-viewing purposes or to print an individual extract or copy for non-commercial personal use only.*
- *Any further dissemination, distribution, reproduction, and copying of the content of the document herein, or the uploading thereof on other websites, or use of the content for any other commercial/unauthorized purposes in any way that could infringe the intellectual property rights of upGrad or its contributors is strictly prohibited.*
- *No graphics, images, or photographs from any accompanying text in this document will be used separately for unauthorized purposes.*
- *No material in this document will be modified, adapted, or altered in any way.*
- *No part of this document or upGrad content may be reproduced or stored on any other website or included in any public or private electronic retrieval system or service without upGrad's prior written permission.*
- *Any rights not expressly granted in these terms are reserved.*