Big Data Analytics – Module 3 (Hadoop)

Prologue

- File systems that manage the storage across a network of machines are called **distributed file systems.**
- But complications of network programming persist
- One of the biggest challenges is making the file system tolerate node failure without suffering data loss

HDFS - Overview

1. History
2. Comparison with RDBMS, Grid Computing
3. Hadoop Ecosystem (Hadoop Architecture)

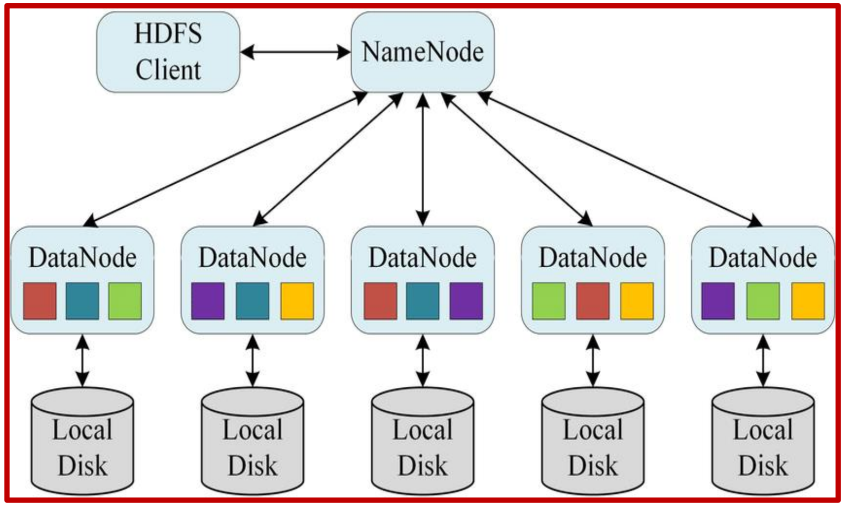
Hadoop Components

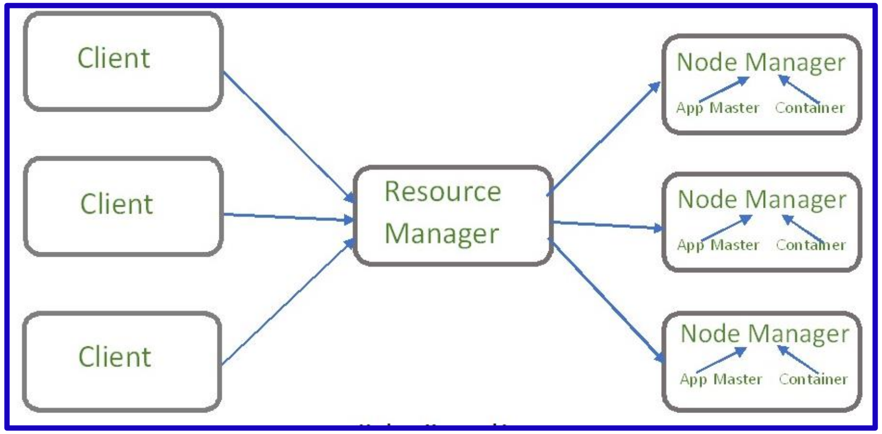
Hadoop consists of 3 main components:

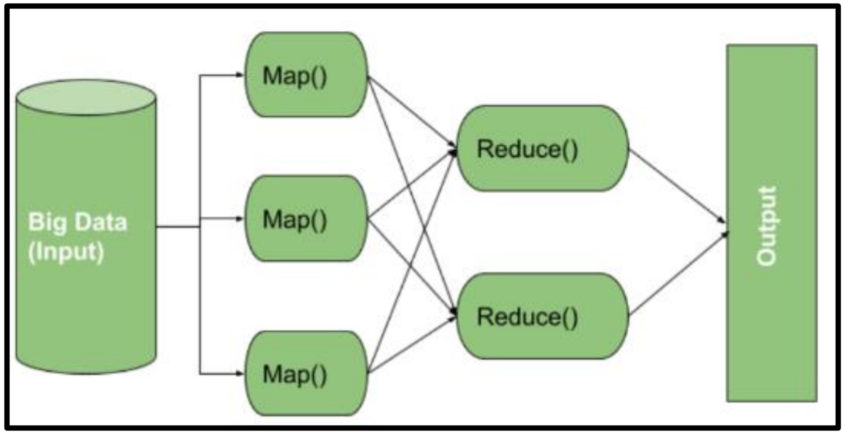
1. Hadoop Distributed File System (HDFS) - Storage System

2. Yet Another Resource Negotiator (YARN) - Resource Management

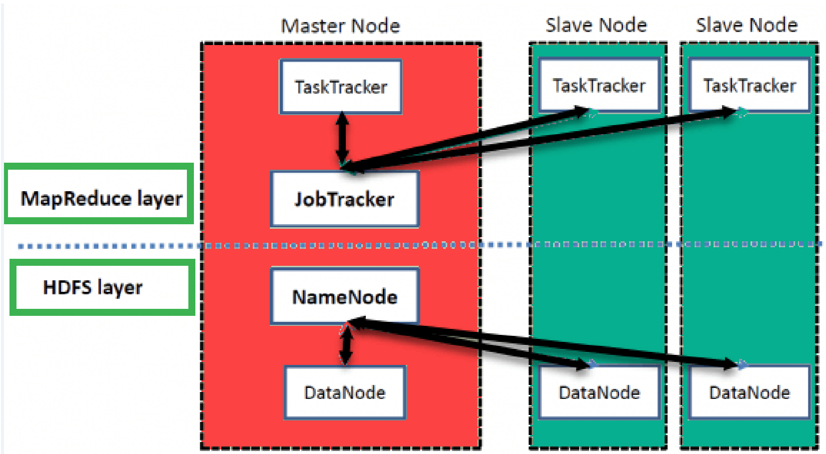
3. MapReduce(MR) – Processing Unit

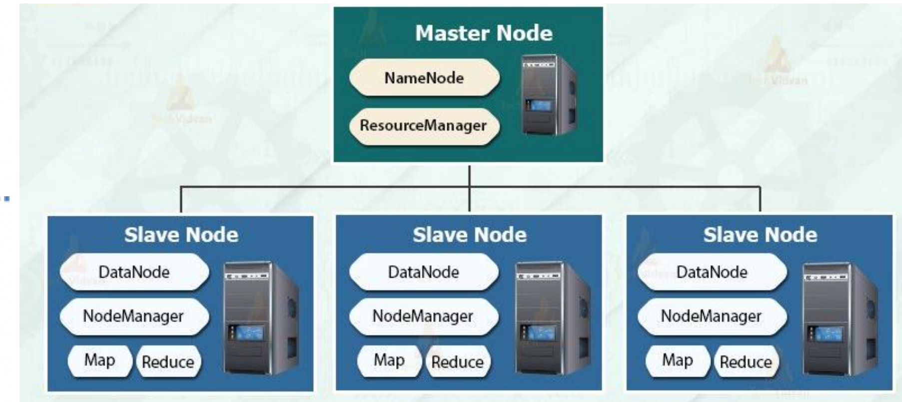


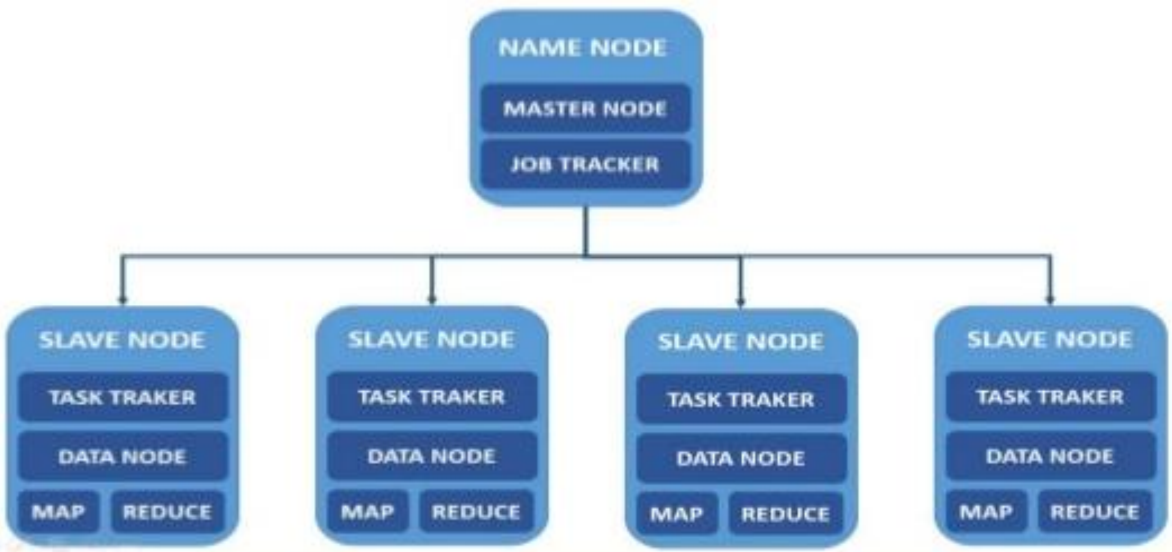




Hadoop Architecture







The Design of HDFS

HDFS, a file system of Hadoop, is designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware.

- Very large files – GB, TB, PB etc.
- Streaming data access – Write once, read many times
- Commodity hardware – low cost, available H/W

HDFS is not a good fit for:

- Low-latency data access
- Lots of small files
- Multiple writers, arbitrary file modifications

HDFS Concepts - Blocks

- A block in HDFS is a much larger unit – 64 MB by default (128 MB in Hadoop V 2.0).
- Files in HDFS are broken into block-sized chunks, which are stored as independent units.
- Unlike a file system for a single disk, a file in HDFS that is smaller than a single block does not occupy a full block's worth of underlying storage. (Ex: 1 MB file stored in a 128 MB block size uses 1 MB of disk space but nit 128 MB).
- HDFS blocks are large in size to reduce seek time and increase transfer time.

Benefits of block abstraction for DFS

- A file can be larger than any single disk in the network
- Making the unit of abstraction a block rather than a file simplifies the storage subsystem
- Blocks fit well with replication for providing fault tolerance and availability

HDFS's ***fsck*** command understands blocks.

For example, running:

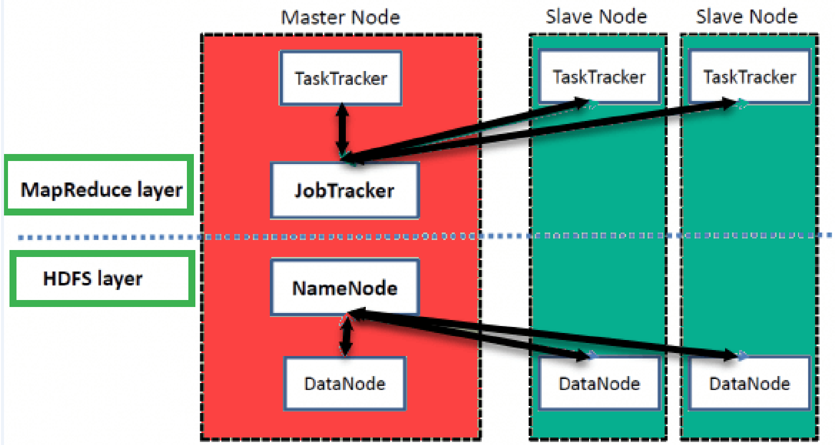
% hadoop fsck / -files -blocks

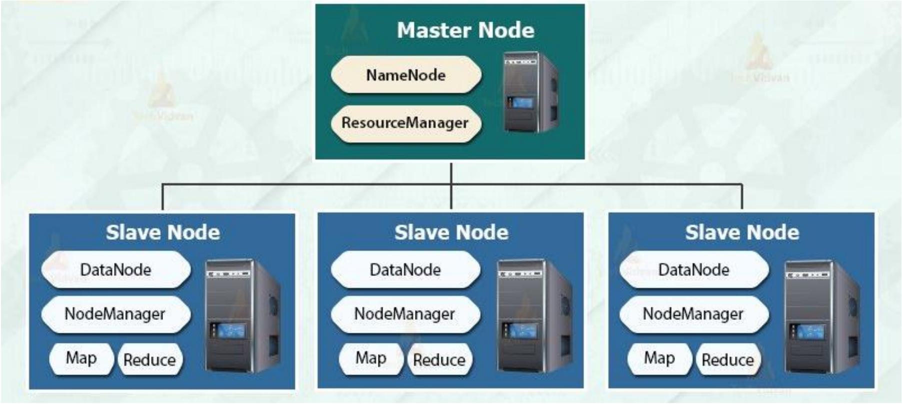
will list the blocks that make up each file in the file system

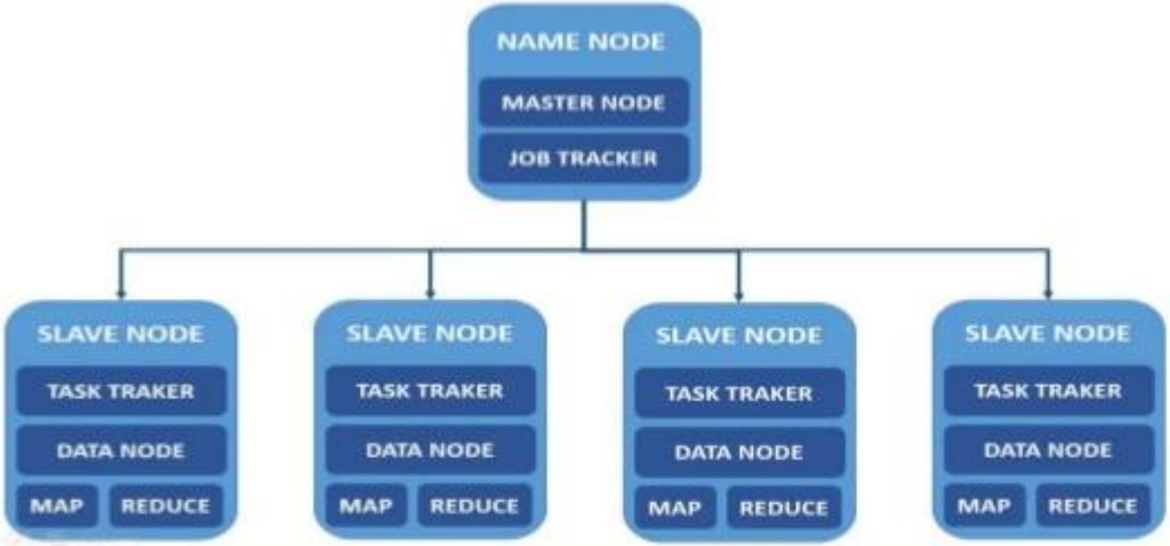
HDFS Concepts – Namenodes and Datanodes

- An HDFS cluster has two types of nodes operating in a master-worker pattern: a namenode and a number of datanodes
- The namenode manages the file system namespace stored persistently on the local disk in the form of two files: **the namespace image** and **the edit log**
- The namenode also knows the datanodes on which all the blocks for a given file are located
- A client accesses the file system through an interface on behalf of the user by communicating with the namenode and datanodes

Hadoop Architecture







HDFS Concepts – Namenodes and Datanodes

- Datanodes are the workhorses of the filesystem
- They store and retrieve blocks when they are told to (by clients or the namenode)
- Without the namenode, the file system cannot be used.
- If the machine running the namenode was obliterated, all the files on the file system would be lost

Namenode resilience to failure

Hadoop provides two mechanisms:

- Back up the files
- Run a secondary namenode

HDFS – High Availability

- The combination of replicating namenode metadata on multiple file systems and using the secondary namenode to create checkpoints protects against data loss, but it does not provide high-availability of the file system
- The namenode is still a Single Point of Failure (SPOF)
- In such an event the whole Hadoop system would effectively be out of service until a new namenode could be brought online

HDFS High-Availability

The new namenode will not be able to serve requests until it has

- loaded its namespace image into memory
- replayed its edit log, and
- received enough block reports from the datanodes to leave safe mode

On large clusters with many files and blocks, the time it takes for a namenode to start from cold can be 30 minutes or more.

HDFS High-Availability

The 2.x release series of Hadoop remedies this situation by adding support for HDFS high availability (HA)

- In this implementation there is a pair of namenodes in an **active- standby** configuration

HDFS High-Availability

A few architectural changes are needed to allow this to happen:

- The namenodes must use highly available shared storage to share the edit log
- Datanodes must send block reports to both namenodes because the block mappings are stored in a namenode's memory, and not on disk
- Clients must be configured to handle namenode failover, using a mechanism that is transparent to users
- If the active namenode fails, the standby can take over very quickly

HDFA HA - Failover and Fencing

- The transition from the active namenode to the standby is managed by a new entity in the system called the **failover controller**
- Failover controllers are pluggable
- Each namenode runs a lightweight failover controller process whose job it is to monitor its namenode for failures (using a simple heartbeating mechanism) and trigger a failover should a namenode fail.
- Failover may also be initiated manually by an administrator, for example, in the case of routine maintenance. This is known as a **graceful failover**
- In the case of an ungraceful failover, it is impossible to be sure that the failed namenode has stopped running
- The HA implements a method known as **fencing**

HDFA HA - Fencing

The system employs a range of fencing mechanisms:

- killing the namenode's process
- revoking its access to the shared storage directory
- disabling its network port via a remote management command
- the previously active namenode can be fenced with a technique known as STONITH(Shoot The Other Node In The Head)

Client failover is handled transparently by the client library. The simplest implementation uses client-side configuration to control failover

Hadoop File Systems

- Hadoop has an abstract notion of file system, of which HDFS is just one implementation.
- The Java abstract class **org.apache.hadoop.fs.FileSystem** represents a file system in Hadoop, and there are several concrete implementations, which are described in the following table:

Hadoop File Systems

Filesystem	URI scheme	Java implementation (all under org.apache.hadoop)	Description
Local	file	fs.LocalFileSystem	A filesystem for a locally connected disk with client-side checksums. Use RawLocalFileSystem for a local filesystem with no checksums. See "LocalFileSystem" on page 99.
HDFS	hdfs	hdfs.DistributedFileSystem	Hadoop's distributed filesystem. HDFS is designed to work efficiently in conjunction with MapReduce.
WebHDFS	webhdfs	hdfs.web.WebHdfsFileSystem	A filesystem providing authenticated read/write access to HDFS over HTTP. See "HTTP" on page 54.
Secure WebHDFS	swebhdfs	hdfs.web.SWebHdfsFileSystem	The HTTPS version of WebHDFS.
HAR	har	fs.HarFileSystem	A filesystem layered on another filesystem for archiving files. Hadoop Archives are used for packing lots of files in HDFS into a single archive file to reduce the namenode's memory usage. Use the hadoop archive command to create HAR files.
View	viewfs	viewfs.ViewFileSystem	A client-side mount table for other Hadoop filesystems. Commonly used to create mount points for federated namenodes (see "HDFS Federation" on page 48).
FTP	ftp	fs.ftp.FTPFileSystem	A filesystem backed by an FTP server.
S3	s3a	fs.s3a.S3AFileSystem	A filesystem backed by Amazon S3. Replaces the older s3n (S3 native) implementation.