

Hadoop-Let us Admin

facultyadvisor

Published
with GitBook



Table of Contents

1. [Preface](#)
2. [Introduction](#)
 - i. [Big Data](#)
 - ii. [Volume](#)
 - iii. [Variety](#)
 - iv. [Velocity](#)
 - v. [Bigdata challenges](#)
3. [Hadoop Eco System](#)
 - i. [Hadoop](#)
 - ii. [Hadoop HDFS](#)
 - iii. [Hadoop MapReduce](#)
 - iv. [Hadoop Hive](#)
 - v. [Hadoop Pig](#)
 - vi. [Hadoop Hbase](#)
 - vii. [Hadoop Spark](#)
 - viii. [Hadoop Platforms](#)
4. [Hadoop components](#)
 - i. [Name Node](#)
 - ii. [Secondary Name Node](#)
 - iii. [Data Node](#)
 - iv. [JobTracker](#)
 - v. [Task tracker](#)
5. [Hadoop NameSpace](#)
 - i. [Block](#)
 - ii. [Block Pool](#)
 - iii. [MetaData](#)
 - iv. [NameSpace](#)
 - v. [Name Space Volume](#)
 - vi. [NameSpace Federation](#)
 - vii. [NameNode Issues](#)
6. [Data Replication](#)
 - i. [File Placement](#)
 - ii. [Data Replication](#)
 - iii. [Block Replication](#)
 - iv. [Replication Factor](#)
7. [Communication](#)
 - i. [NameNode-DataNode](#)
 - ii. [Data Communication](#)
 - iii. [Heart Beat](#)
 - iv. [Heart Beat Contents](#)
 - v. [Block report](#)
8. [Failure Management](#)
 - i. [Check point](#)
 - ii. [FSImage](#)
 - iii. [EditLog](#)
 - iv. [Backup Node](#)
 - v. [Block Scanner](#)
 - vi. [Failure Types](#)
 - vii. [Failover Controller](#)

9. Hadoop Configuration

- i. [Core-*.xml](#)
- ii. [hdfs-*.xml](#)
- iii. [mapred-*.xml](#)
- iv. [yarn-*.xml](#)

Preface

The book is designed as an precursor to enable interested people to gain an understanding of the concepts of Bigdata. The text is structured to expose the elementary components of hadoop leading to configuration.

BigData

Big Data

Recent data management trends suggest a rapid growth in the use of large server infrastructure like data centers and cloud computing platforms for scalable services. Internet services, like Google and Yahoo!, are deploying their computing infrastructure to run applications on massive amounts of input data to compute business results. The massive volume of data being used are publicly available and is increasing year over year. The emerging future is more exciting. Success in the future will be dictated to a large extent by the organization's ability to extract value from other organization's data.

The ability to process massive volume of data is made available by Google's software and the closely related open-source Hadoop software. They are revolutionizing the Internet services community by building scalable systems infrastructure for data intensive applications.

As business "best practices" trend increasingly towards basing **decisions off data and hard facts rather than instinct and theory**, the corporate thirst for systems that can manage, process, and granularly analyze data is becoming insatiable. Venture capitalists are very much aware of this trend, and have funded no fewer than a dozen new companies in recent years that build specialized analytical data management software (e.g., Netezza, Vertica, DATAlegro, Greenplum, Aster Data, Infobright, Kickfire, Dataupia, ParAccel, and Exasol), and continue to fund them, even in pressing economic times.

BigData defined

Big data is data that **exceeds the processing capacity** of conventional database systems. **The data is too big, moves too fast, or doesn't fit the strictures of conventional database architectures** By definition, **Big Data is data that cannot be processed using the resources of a single machine**. The term Big Data applies to information that can't be processed or analyzed using traditional processes or tools. Within this data lie valuable patterns and information, previously hidden because of the amount of work required to extract them. The value of big data to an organization falls into two categories: **analytical use and enabling new products**.

With more and more businesses reporting petabyte-sized data warehouses, the system of choice for managing and analyzing massive amounts of the data (Big Data) will be the one that

- provides high performance,
- scales over clusters of thousands of heterogeneous machines
- is versatile.

The power of an organization's information can be enhanced by its trustworthiness, its volume, its accessibility, and the capability of an organization to be able to make sense of it all in a reasonable amount of time in order to empower intelligent decision making.

Volume

Data volume measures the amount of data available to an organization, which does not necessarily have to own all of it as long as it can access it. As data volume increases, the value of different data records will decrease in proportion to age, type, richness, and quantity among other factors.

Variety

Data variety is a measure of the richness of the data representation – text, images video, audio, etc. From an analytic perspective, it is probably the biggest obstacle to effectively using large volumes of data. Incompatible data formats, non-aligned data structures, and inconsistent data semantics represents significant challenges that can lead to analytic sprawl.

As data volumes increase, so do requirements for efficiently extracting value from data sets.

Velocity

Data velocity measures the speed of data creation, streaming, and aggregation. eCommerce has rapidly increased the speed and richness of data used for different business transactions (for example, web-site clicks). Data velocity management is much more than a bandwidth issue; it is also an ingest issue (extracttransform-load).

BigData challenges

Increasingly, organizations today are facing more and more Big Data challenges. Big data analytics reveals insights hidden previously by data too costly to process, such as peer influence among customers, revealed by analyzing shoppers' transactions and social and geographical data in real time.

Realtime

The meaning of **real time** can vary depending on the context in which it is used. Real-time denotes the ability to process data as it arrives, rather than storing the data and retrieving it at some point in the future. But "the present" also has different meanings to different users.

From the perspective of an online merchant, **the present** means the attention span of a potential customer. If the processing time of a transaction exceeds the customer's attention span, the merchant doesn't consider it real time. From the perspective of an options trader, however, real time means milliseconds. From the perspective of a guided missile, real time means microseconds. Real-time big data analytics is an iterative process involving multiple tools and systems.

Hardware Failure

Hardware failure is the norm rather than the exception. An HDFS instance may consist of hundreds or thousands of server machines, each storing part of the file system's data. The fact that there are a huge number of components and that each component has a nontrivial probability of failure means that some component of HDFS is always non-functional.

Hadoop Eco System

Hadoop is based on the Google paper on MapReduce published in 2004, and its development started in 2005. At the time, Hadoop was developed to support the open-source web search engine project called Nutch. Eventually, Hadoop separated from Nutch and became its own project under the Apache Foundation.

Apache Hadoop is a framework. It permits the users to store data across a cluster. It permits distributed processing of large data sets across commodity computers using a simple programming model. Each element of the cluster provides for computation and storage. It is designed to scale up from single servers to thousands of machines. It parallelizes data processing across many nodes computers in a cluster. It speeds up large computations and hides I/O latency.

At its core, Hadoop is a Java-based MapReduce framework. However, due to the rapid adoption of the Hadoop platform, there was a need to support the non-Java user community. Today Hadoop is the best-known MapReduce framework in the market. Currently, several companies have grown around Hadoop to provide support, consulting, and training services for the Hadoop software.

Machine Failure

It is understood that machine failure is the order of the day. Hence it does not make sense to rely on hardware to deliver high-availability. Hence the framework itself is designed to detect and handle failures at the application layer, thus delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.

Hadoop is a composition of various individual components creating the Hadoop eco-system.

For a very long time, Hadoop remained a system in which users submitted jobs that ran on the entire cluster. Jobs would be executed in a First In, First Out (FIFO) mode. However, this led to situations in which a long-running, less-important job would hog resources and not allow a smaller yet more important job to execute. To solve this problem, more complex job schedulers in Hadoop, such as the Fair Scheduler and Capacity Scheduler were created. But Hadoop still had scalability limitations that were a result of some deeply entrenched design decisions resulting in Hadoop 2.

Hadoop encompasses a multiplicity of tools that are designed and implemented to work together.

Hadoop

Hadoop is an open-source ecosystem used for storing, managing and analyzing a large volume of data. It is a data management system bringing together massive amounts of structured and unstructured data that touch nearly every layer of the traditional enterprise data stack, positioned to occupy a central place within a data center. It fills a gap in the market by effectively storing and providing computational capabilities over substantial amounts of data. Hadoop is different from previous distributed approaches in the following ways:

- Data is distributed in advance.
- Data is replicated throughout a cluster of computers for reliability and availability.
- Data processing tries to occur where the data is stored, thus eliminating bandwidth bottlenecks.

Hadoop ecosystem consists of several sub projects, and two of them form the very basic of Hadoop ecosystem, which are the distributed computation framework MapReduce and the distributed storage layer Hadoop Distributed File System (HDFS).

The MapReduce framework is both a parallel computation framework and a scheduling framework. The Apache MapReduce framework consists of two components:

- a single JobTracker and
- several TaskTracker

Until the Hadoop 2.x release, HDFS and MapReduce employed single-master models, resulting in single points of failure

Apache HDFS

The Hadoop Distributed File System (HDFS) is a solution to store large files across clustered machines. Hadoop and HDFS were derivative names created from Google File System (GFS) paper. It is portable across heterogeneous hardware and software platforms. It is the primary distributed storage used by Hadoop applications. It is implemented as a block-structured filesystem. It has a simple coherency model named write-once-read-many (WORM) for files and works on idea of moving computation to data.

Goal

The goal of Hadoop distributed file system is to address the issues of hardware failure, high throughput data access and process large data sets of applications. Its implementation addresses a number of problems that are present in a number of distributed filesystems.

The HDFS architecture diagram depicts basic interactions among NameNode, the DataNodes, and the clients. A HDFS cluster primarily consists of a NameNode that manages the file system metadata and DataNodes that store the actual data.

One of the requirements for such a block-structured filesystem is the capability to store, manage, and access file metadata (information about files and blocks) reliably, and to provide fast access to the metadata store

Hadoop MapReduce

MapReduce is a programming model for processing large data sets with a parallel, distributed algorithm on a cluster. Apache MapReduce was derived from Google concept of MapReduce.

MR1

Hadoop supports the MapReduce model, which was introduced by Google as a method of solving a class of petascale problems with large clusters of commodity machines. The core concept of MapReduce in Hadoop is that input can be split into logical chunks, and each chunk can be initially processed independently by a map task. The model is based on two distinct steps, both of which are custom and user-defined for an application:

- **Map:** An initial ingestion and transformation step in which individual input records can be processed in parallel. A map task can run on any compute node in the cluster, and multiple map tasks can run in parallel across the cluster. The map task is responsible for transforming the input records into key/value pairs. The output of all the maps will be partitioned, and each partition will be sorted.
- **Reduce:** An aggregation or summarization step in which all associated records must be processed together by a single entity. There will be one partition for each reduce task. Each partition's sorted keys and the values associated with the keys are then processed by the reduce task. There can be multiple reduce tasks running in parallel on the cluster.

MR2

The current Apache MapReduce version is built over Apache YARN Framework. YARN stands for “Yet-Another-Resource-Negotiator”. It is a new framework that facilitates writing arbitrary distributed processing frameworks and applications.

YARN's execution model is more generic than the earlier MapReduce implementation. YARN can run applications that do not follow the MapReduce model, unlike the original Apache Hadoop MapReduce (also called MR1). Hadoop YARN is an attempt to take Apache Hadoop beyond MapReduce for data-processing.

Hadoop Hive

Apache Hive is a data warehouse infrastructure built on top of Hadoop for providing data summarization, query and analysis. Users of MapReduce quickly realized that developing MapReduce programs is a very programming-intensive task, which makes it error-prone and hard to test. Using Hadoop was not easy for end users, especially for the ones who were not familiar with MapReduce framework. End users had to write map/reduce programs for simple tasks like getting raw counts or averages. There was a need for more expressive languages such as SQL to enable users to focus on the problem instead of low-level implementations of typical SQL artifacts.

Hive was created to make it possible for analysts with strong SQL skills (but meager Java programming skills) to run queries on the huge volumes of data to extract patterns and meaningful information. It provides an SQL-like language called HiveQL while maintaining full support for map/reduce. In short, a Hive query is converted to MapReduce tasks.

Apache Hive evolved to provide a data warehouse (DW) capability to large datasets. Users can express their queries in Hive Query Language, which is very similar to SQL. The Hive engine converts these queries to low-level MapReduce jobs transparently. More advanced users can develop user-defined functions (UDFs) in Java. Hive also supports standard drivers such as ODBC and JDBC. Hive is also an appropriate platform to use when developing Business Intelligence (BI) types of applications for data stored in Hadoop.

Hadoop Pig

Pig provides an engine for executing data flows in parallel on Hadoop. It is a platform for analyzing large data sets that consists of a high-level language for expressing data analysis programs, coupled with infrastructure for evaluating these programs. Pig runs on Hadoop.

Pig Latin

It includes a language, Pig Latin, for expressing these data flows. Pig uses MapReduce to execute all of its data processing. It compiles the Pig Latin scripts that users write into a series of one or more MapReduce jobs that it then executes. Pig Latin looks different from many of the programming languages you have seen. Pig Latin includes operators for many of the traditional data operations (join, sort, filter, etc.), as well as the ability for users to develop their own functions for reading, processing, and writing data.

The salient property of Pig programs is that their structure is amenable to substantial parallelization, which in turns enables them to handle very large data sets. It makes use of both the Hadoop Distributed File System, HDFS, and Hadoop's processing system, MapReduce.

There are no if statements or for loops in Pig Latin. This is because traditional procedural and object-oriented programming languages describe control flow, and data flow is a side effect of the program. Pig Latin instead focuses on data flow.

Although the motivation for Pig was similar to Hive, Hive is a SQL-like language, which is declarative. On the other hand, Pig is a procedural language that works well in datapipeline scenarios. Pig will appeal to programmers who develop data-processing pipelines (for example, SAS programmers). It is also an appropriate platform to use for extract, load, and transform (ELT) types of applications.

Hadoop Hbase

All the preceding projects, including MapReduce, are batch processes. However, there is a strong need for real-time data lookup in Hadoop. Hadoop did not have a native key/value store. For example, consider a Social Media site such as Facebook. If you want to look up a friend's profile, you expect to get an answer immediately (not after a long batch job runs). Such use-cases were the motivation for developing the HBase platform.

Hadoop Spark

Data analytics cluster computing framework originally developed in the AMPLab at UC Berkeley. Spark fits into the Hadoop open-source community, building on top of the Hadoop Distributed File System (HDFS). However, Spark provides an easier to use alternative to Hadoop MapReduce and offers performance up to 10 times faster than previous generation systems like Hadoop MapReduce for certain applications.

Spark is a framework for writing fast, distributed programs. Spark solves similar problems as Hadoop MapReduce does but with a fast in-memory approach and a clean functional style API. With its ability to integrate with Hadoop and inbuilt tools for interactive query analysis (Shark), large-scale graph processing and analysis (Bagel), and real-time analysis (Spark Streaming), it can be interactively used to quickly process and query big data sets. To make programming faster, Spark provides clean, concise APIs in Scala, Java and Python. You can also use Spark interactively from the Scala and Python shells to rapidly query big datasets. Spark is also the engine behind Shark, a fully Apache Hive-compatible data warehousing system that can run 100x faster than Hive.

Hadoop Platforms

Hadoop supports three different operating modes:

- Standalone mode: In this mode, Hadoop will run as a single process on a single node.
- Pseudo-distributed mode: In this mode, Hadoop will run all services in separate processes on a single node.
- Fully-distributed mode: In this mode, Hadoop will run all services in separate processes across multiple nodes

Hadoop components

Let the discussion start with Hadoop 1.x components and eventually move towards the new 2.x components. At a very high level, Hadoop 1.x has following daemons:

Name Node

The Name Node is a centralized service. It is deployed in a single node and as a cluster manager. The primary job of namenode is to manage the file system namespace. The file system tree and the metadata for all the files and directories are maintained in the namenode. It is the arbitrator and repository for all HDFS metadata. It maintains and stores the namespace tree and the mapping of file blocks to Data Nodes persistently on the local disk in the form of two files:

- the namespace image
- the edit log

All the file system metadata is stored on a metadata server. All metadata operations may be handled by a single metadata server, but a cluster will configure multiple metadata servers as primary-backup failover pairs. This includes the namespace, data location and access permissions.

Prior to Hadoop 2.0.0, the NameNode was a single point of failure (SPOF) in an HDFS cluster. With Zookeeper the HDFS High Availability feature addresses this problem by providing the option of running two redundant NameNodes in the same cluster in an Active/Passive configuration with a hot standby.

Operations

Clients contact the Name Node in order to perform common file system operations, such as open, close, rename, and delete. The Name Node does not store HDFS data itself, but rather maintains a mapping between HDFS file name, a list of blocks in the file, and the Data Node(s) on which those blocks are stored. The system is designed in such a way that user data never flows through the Name Node.

It periodically receives a Heartbeat and a Block report from each of the Data Nodes in the cluster. Receipt of a Heartbeat implies that the Data Node is functioning properly. A Block report contains a list of all blocks on a Data Node.

Namenode Format

When the NameNode is formatted a namespace ID is generated, which essentially identifies that specific instance of the distributed filesystem. When DataNodes first connect to the NameNode they store that namespace ID along with the data blocks, because the blocks have to belong to a specific filesystem.

If a DataNode later connects to a NameNode, and the namespace ID which the NameNode declares does not match the namespace ID stored on the DataNode, it will refuse to operate with the "incompatible namespace ID" error. It means that the DataNode has connected to a different NameNode, and the blocks which it is storing don't belong to that distributed filesystem.

Secondary Name Node

Secondary NameNode: This is not a backup NameNode. In fact, it is a poorly named component of the Hadoop platform. It performs some housekeeping functions for the NameNode.

The goal of the edits file is to accumulate the changes during the system operation. If the system is restarted, the contents of the edits file can be rolled into fsimage during the restart.

The role of the Secondary NameNode is to periodically merge the contents of the edits file in the fsimage file. To this end, the Secondary NameNode periodically executes the following sequence of steps:

1. It asks the Primary to roll over the edits file, which ensures that new edits go to a new file. This new file is called edits.new.
2. The Secondary NameNode requests the fsimage file and the edits file from the Primary.
3. The Secondary NameNode merges the fsimage file and the edits file into a new fsimage file.
4. The NameNode now receives the new fsimage file from the Secondary NameNode with which it replaces the old file. The edits file is now replaced with the contents of the edits. new file created in the first step.
5. The fstime file is updated to record when the checkpoint operation took place.

Data Node

In HDFS, the daemon responsible for storing and retrieving block data is called the datanode (DN). The data nodes are responsible for serving read and write requests from clients and perform block operations upon instructions from name node. Each Data Node stores HDFS blocks on behalf of local or remote clients. Each block is saved as a separate file in the node's local file system. Because the Data Node abstracts away details of the local storage arrangement, all nodes do not have to use the same local file system. Blocks are created or destroyed on Data Nodes at the request of the Name Node, which validates and processes requests from clients. Although the Name Node manages the namespace, clients communicate directly with Data Nodes in order to read or write data at the HDFS block level. A Data node normally has no knowledge about HDFS files. While starting up, it scans through the local file system and creates a list of HDFS data blocks corresponding to each of these local files and sends this report to the Name node.

Individual files are broken into blocks of a fixed size and distributed across multiple DataNodes in the cluster. The Name Node maintains metadata about the size and location of blocks and their replicas.

Hadoop was designed with an idea that DataNodes are "disposable workers", servers that are fast enough to do useful work as a part of the cluster, but cheap enough to be easily replaced if they fail.

The data block is stored on multiple computers, improving both resilience to failure and data locality, taking into account that network bandwidth is a scarce resource in a large cluster.

JobTracker

One of the master components, it is responsible for managing the overall execution of a job. It performs functions such as scheduling child tasks (individual Mapper and Reducer) to individual nodes, keeping track of the health of each task and node, and even rescheduling failed tasks. As we will soon demonstrate, like the NameNode, the Job Tracker becomes a bottleneck when it comes to scaling Hadoop to very large clusters. The JobTracker daemon is responsible for launching and monitoring MapReduce jobs.

Task tracker

The Task Tracker is a service daemon. It Runs on individual DataNodes. It is responsible for starting and managing individual Map/Reduce tasks. It communicates with the JobTracker. It runs on each compute node of the Hadoop cluster, accepts requests for individual tasks such as Map, Reduce, and Shuffle operations. which are present on every node of the cluster. The actual execution of the tasks is controlled by TaskTrackers. It is responsible to start map jobs.

Each TaskTracker is configured with a set of slots that is usually set up as the total number of cores available on the machine. When a request is received from the JobTracker to launch a task, the TaskTracker initiates a new JVM for the task. The TaskTracker is assigned a task depending on how many free slots it has (total number of tasks = actual tasks running).

The TaskTracker is responsible for sending heartbeat messages to the JobTracker. Apart from telling the JobTracker that it is healthy, these messages also tell the JobTracker about the number of available free slots.

Hadoop NameSpace

Block

A disk has a block size, which is the minimum amount of data that it can read or write. Filesystems for a single disk build on this by dealing with data in blocks, which are an integral multiple of the disk block size. Filesystem blocks are typically a few kilobytes in size, whereas disk blocks are normally 512 bytes. This is generally transparent to the filesystem user who is simply reading or writing a file of whatever length.

Having a block abstraction for a distributed filesystem brings several benefits. The first benefit is the most obvious: a file can be larger than any single disk in the network. There's nothing that requires the blocks from a file to be stored on the same disk, so they can take advantage of any of the disks in the cluster. Like in a filesystem for a single disk, files in HDFS are broken into block-sized chunks, which are stored as independent units. HDFS, too, has the concept of a block, but it is a much larger unit—64 MB by default. HDFS blocks are large compared to disk blocks, and the reason is to minimize the cost of seeks. By making a block large enough, the time to transfer the data from the disk can be significantly longer than the time to seek to the start of the block. Thus the time to transfer a large file made of multiple blocks operates at the disk transfer rate.

A file can be made up of several blocks, which are stored on different DataNodes chosen randomly on a block-by-block basis. As a result, access to a file usually requires access to multiple DataNodes, which means that HDFS supports file sizes far larger than a single-machine disk capacity. The DataNode stores each HDFS data block in a separate file on its local filesystem with no knowledge about the HDFS files themselves.

In fact, it would be possible, if unusual, to store a single file on an HDFS cluster whose blocks filled all the disks in the cluster.

The default block size and replication factor are specified by Hadoop configuration, but can be overwritten on a per-file basis. An application can specify block size, the number of replicas, and the replication factor for a specific file at its creation time.

There are tools to perform filesystem maintenance, such as `df` and `fsck`, that operate on the filesystem block level.

Block Pool

A Block Pool is a set of blocks that belong to a single namespace. Datanodes store blocks for all the block pools in the cluster. Each Block Pool is managed independently. This allows a namespace to generate Block IDs for new blocks without the need for coordination with the other namespaces. A Namenode failure does not prevent the Datanode from serving other Namenodes in the cluster.

Large blocks

HDFS blocks are large compared to disk blocks, and the reason is to minimize the cost of seeks. By making a block large enough, the time to transfer the data from the disk can be significantly longer than the time to seek to the start of the block. Thus the time to transfer a large file made of multiple blocks operates at the disk transfer rate.

Having a block abstraction for a distributed filesystem brings several benefits. The first benefit is the most obvious: a file can be larger than any single disk in the network. There's nothing that requires the blocks from a file to be stored on the same disk, so they can take advantage of any of the disks in the cluster. In fact, it would be possible, if unusual, to store a single file on an HDFS cluster whose blocks filled all the disks in the cluster.

MetaData

Because of the relatively low amount of metadata per file (it only tracks filenames, permissions, and the locations of each block), the NameNode stores all of the metadata in the main memory, thus allowing for a fast random access. The metadata storage is designed to be compact. As a result, a NameNode with 4 GB of RAM is capable of supporting a huge number of files and directories.

Modern distributed and parallel file systems such as pNFS , PVFS, HDFS, and GoogleFS treat metadata services as an independent system component, separately from data servers. A reason behind this separation is to ensure that metadata access does not obstruct the data access path. Another reason is design simplicity and the ability to scale the two parts of the system independently.

Files and directories are represented on the NameNode by inodes, which record attributes like permissions, modification and access times, namespace and disk space quotas. The NameNode maintains the file system namespace. Any change to the file system namespace or its properties is recorded by the NameNode. HDFS keeps the entire namespace in RAM.

Metadata are the most important management information replicated for namenode failover. In our solution, the metadata include initial metadata which are replicated in initialization phase and two types of runtime metadata which are replicated in replication phase. The initial metadata include two types of files: version file which contains the version information of running HDFS and file system image (fsimage) file which is a persistent checkpoint of the file system. Both files are replicated only once in initialization phase, because their replication are time-intensive processes. Slave node updates fsimage file based on runtime metadata to make the file catch up with that of primary node

The name node has an in-memory data structure called FsImage that contains the entire file system namespace and maps the files on to blocks. The NameNode stores all Metadata in a file called FsImage.

NameSpace

Traditional local file systems support a persistent name space. Local file system views devices as being locally attached, the devices are not shared, and hence there is no need in the file system design to enforce device sharing semantics.

HDFS supports a traditional hierarchical file organization. The file system namespace hierarchy is similar to most other existing file systems; one can create and remove files, move a file from one directory to another, or rename a file. The NameNode exposes a file system namespace and allows data to be stored on a cluster of nodes while allowing the user a single system view of the file system. HDFS exposes a hierarchical view of the file system with files stored in directories, and directories can be nested. The NameNode is responsible for managing the metadata for the files and directories. The current HDFS architecture allows only a single namespace for the entire cluster. This namespace is managed by a single namenode. This architectural decision made HDFS simpler to implement.

Files can be organized under Directory, which together form the namespace of a file system. A file system typically organizes its namespace with a tree-structured hierarchical organization. A distributed file system is a file system that allows access to files from multiple hosts across a network.

A user or an application can create directories and store files inside these directories.

Namespace partitioning has been a research topic for a long time, and several methods have been proposed to solve this problem in academia. These can be generally categorized into four types:

- (1) Static Subtree Partitioning
- (2) Hashing
- (3) Lazy Hybrid
- (4) Dynamic Subtree Partitioning

Name Space Volume

A Namespace and its block pool together are called Namespace Volume. It is a self-contained unit of management. When a namenode/namespace is deleted, the corresponding block pool at the datanodes is deleted. Each namespace volume is upgraded as a unit, during cluster upgrade.

NameSpace Federation

Since file metadata is stored in memory, the amount of memory in the NameNodes defines the number of files that can be available on an Hadoop cluster. To overcome the limitation of a single NameNode memory and being able to scale the name service horizontally, Hadoop has introduced HDFS Federation, which is based on multiple independent NameNodes/namespaces. Following are the main benefits of HDFS Federation:

- **Namespace scalability** — HDFS cluster storage scales horizontally, but the namespace does not. Large deployments (or deployments using a lot of small files) benefit from scaling the namespace by adding more NameNodes to the cluster.
- **Performance** — Filesystem operation throughput is limited by a single NameNode. Adding more NameNodes to the cluster scales the filesystem read/write operation's throughput.
- **Isolation** — A single NameNode offers no isolation in a multi-user environment. An experimental application can overload the NameNode and slow down production-critical applications. With multiple NameNodes, different categories of applications and users can be isolated to different namespaces.

Under federation, each namenode manages a namespace volume, which is made up of the metadata for the namespace, and a block pool containing all the blocks for the files in the namespace. Namespace volumes are independent of each other, which means namenodes do not communicate with one another, and furthermore the failure of one namenode does not affect the availability of the namespaces managed by other namenodes.

Although HDFS Federation solves the problem of HDFS scalability, it does not solve the NameNode reliability issue.

NameNode Issues

The large size of namespace catering millions of clients and billions of files and directories imposes a big challenge to provide high scalability and performance of metadata services. In such systems a structured, decentralized, self organizing and self healing approach is required.

Data Replication

File Placement

HDFS uses replication to maintain at least three copies (one primary and two replicas) of every chunk. Applications that require more copies can specify a higher replication factor typically at file create time. All copies of a chunk are stored on different data nodes using a rack-aware replica placement policy. The first copy is always written to the local storage of a data node to lighten the load on the network. To handle machine failures, the second copy is distributed at random on different data nodes on the same rack as the data node that stored the first copy. This improves network bandwidth utilization because inter-rack communication is faster than cross-rack communication which often goes through intermediate network switches. To maximize data availability in case of a rack failure, HDFS stores a third copy distributed at random on data nodes in a different rack.

HDFS uses a random chunk layout policy to map chunks of a file on to different data nodes. At file create time; the name node randomly selects a data node to store a chunk. This random chunk selection may often lead to sub-optimal file layout that is not uniformly load balanced. The name node is responsible to maintain the chunk to data node mapping which is used by clients to access the desired chunk.

Data Replication

HDFS is designed to reliably store very large files across machines in a large cluster. It stores each file as a sequence of blocks; all blocks in a file except the last block are the same size. The blocks of a file are replicated for fault tolerance. The block size and replication factor are configurable per file. An application can specify the number of replicas of a file. The replication factor can be specified at file creation time and can be changed later. Files in HDFS are write-once and have strictly one writer at any time.

Block Replication

The NameNode is responsible for block replication. The Name Node makes all decisions regarding replication of blocks. Replica placement determines HDFS reliability, availability and performance. Each replica on unique racks helps in preventing data losses on entire rack failure and allows use of bandwidth from multiple racks when reading data. This policy evenly distributes replicas in the cluster which makes it easy to balance load on component failure. However, this policy increases the cost of writes because a write needs to transfer blocks to multiple racks. The NameNode keeps checking the number of replicas.

If a block is under replication, then it is put in the replication priority queue. The highest priority is given to low replica value. Placement of new replica is also based on priority of replication. If the number of existing replicas is one, then a different rack is chosen to place the next replica. In case of two replicas of the block on the same rack, the third replica is placed on a different rack. Otherwise, the third replica is placed on a different node in the same rack as an existing replica. The NameNode also checks that all replicas of a block should not be at one rack. If so, NameNode treats the block as under-replicated and replicates the block to a different rack and deletes the old replica.

Replication Factor

The Name Node maintains the file system namespace. Any change to the file system namespace or its properties is recorded by the Name Node. An application can specify the number of replicas of a file that should be maintained by HDFS. The number of copies of a file is called the replication factor of that file. This information is stored by the Name Node.

Communication

NameNode-DataNode

Data Nodes and Name Node connections are established by handshake where namespace ID and the software version of the Data Nodes are verified. The namespace ID is assigned to the file system instance when it is formatted. The namespace ID is stored persistently on all nodes of the cluster. A different namespace ID node cannot join the cluster.

Data Communication

All HDFS communication protocols are layered on top of the TCP/IP protocol. A client establishes a connection to a configurable TCP port on the NameNode machine. It talks the ClientProtocol with the NameNode. The DataNodes talk to the NameNode using the DataNodes Protocol. A Remote Procedure Call (RPC) abstraction wraps both the Client Protocol and the DataNodes Protocol. By design, the NameNode never initiates any RPCs. Instead, it only responds to RPC requests issued by DataNodes or clients.

Heart Beat

Heartbeats carry information about total storage capacity, fraction of storage in use, and the number of data transfers currently in progress. Heartbeats also carry information about total and used disk capacity and the number of data transfers currently performed by the node, which plays an important role in the name-node's space and load balancing decisions. These statistics are used for the Name Node's space allocation and load balancing decisions. The Name Node can process thousands of heartbeats per second without affecting other Name Node operations.

Name Node considers Data Nodes as alive as long as it receives Heartbeat message (default Heartbeat interval is three seconds) from Data Nodes. If the Name Node does not receive a heartbeat from a Data Nodes in XX minutes the Name Node considers the Data Nodes as dead and stops forwarding IO request to it. The Name Node then schedules the creation of new replicas of those blocks on other Data Nodes.

The NameNode can process thousands of heartbeats per second without affecting other NameNode operations. The data nodes send regular heartbeats to the name node so the name node can detect data node failure. During normal operation DataNodes send heartbeats to the NameNode to confirm that the DataNode is operating and the block replicas it hosts are available. The default heartbeat interval is three seconds. If the name node does not receive heartbeats from data nodes for a predetermined period, it marks them as dead and does not forward any new read, write or replication requests to them. The NameNode then schedules creation of new replicas of those blocks on other DataNodes.

The heartbeat message includes the BlockReport from the data node. By design, the name node never initiates any remote procedure calls (RPCs). Instead, it only responds to RPC requests issued by data nodes or clients. It replies to heartbeats with replication requests for the specific data node. Heartbeats from a DataNode also carry information about total storage capacity, fraction of storage in use, and the number of data transfers currently in progress. These statistics are used for the NameNode's space allocation and load balancing decisions.

Heart Beat Contents

The contents of the heartbeat message are:

- Progress report of tasks currently running on sender TaskTracker.
- Lists of completed or failed tasks.
- State of resources – virtual memory, disk space, etc.
- A Boolean flag (acceptNewTasks) indicating whether the sender TaskTracker should be accept new tasks *

The NameNode does not directly call DataNodes. It uses replies to heartbeats to send instructions to the DataNodes. The instructions include commands to:

- replicate blocks to other nodes;
- remove local block replicas;
- re-register or to shut down the node;
- send an immediate block report.

Block report

The DataNode stores HDFS data in files in its local file system. The DataNode has no knowledge about HDFS files. It stores each block of HDFS data in a separate file in its local file system. The DataNode does not create all files in the same directory. Instead, it uses a heuristic to determine the optimal number of files per directory and creates subdirectories appropriately. It is not optimal to create all local files in the same directory because the local file system might not be able to efficiently support a huge number of files in a single directory. When a DataNode starts up, it scans through its local file system, generates a list of all HDFS data blocks that correspond to each of these local files and sends this report to the NameNode: this is the Blockreport.

A DataNode identifies block replicas in its possession to the NameNode by sending a block report. A block report contains the block id, the generation stamp and the length for each block replica the server hosts. The first block report is sent immediately after the DataNodes registrations. Subsequent block reports are sent every hour and provide the NameNode with an up-to date view of where block replicas are located on the cluster.

Check point

Checkpoint is an image record written persistently to disk. Name Node uses two types of files to persist its namespace:

- Fimage: the latest checkpoint of the namespace
- Edits: logs containing changes to the namespace; these logs are also called journals.

The NameNode keeps an image of the entire file system namespace and file Blockmap in memory. This key metadata item is designed to be compact, such that a NameNode with 4 GB of RAM is plenty to support a huge number of files and directories. When the NameNode starts up, it reads the Fimage and EditLog from disk, applies all the transactions from the EditLog to the in-memory representation of the Fimage, and flushes out this new version into a new Fimage on disk. It can then truncate the old EditLog because its transactions have been applied to the persistent Fimage. This process is called a checkpoint.

The Checkpoint node uses parameter **fs.checkpoint.period** to check the interval between two consecutive checkpoints. The Interval time is in seconds (default is 3600 second). The Edit log file size is specified by parameter **fs.checkpoint.size** (default size 64MB) and a checkpoint triggers if size exceeds. Multiple checkpoint nodes may be specified in the cluster configuration file.

FSImage

The entire filesystem namespace is contained in a file called the FsImage stored as a file in the NameNode's local filesystem. The image file represents an HDFS metadata state at a point in time

The entire file system namespace, including the mapping of blocks to files and file system properties, is stored in a file called the FsImage. The FsImage is stored as a file in the NameNode's local file system. Name Node creates an updated file system metadata by merging both files i.e. fsimage and edits on restart. The Name Node then overwrites fsimage with the new HDFS state and begins a new edits journal. The Checkpoint node periodically downloads the latest fsimage and edits from the active NameNode to create checkpoints by merging them locally and then to upload new checkpoints back to the active NameNode. This requires the same memory space as that of NameNode and so checkpoint needs to be run on separate machine. Namespace information lost if either the checkpoint or the journal is missing, so it is highly recommended to configure HDFS to store the checkpoint and journal in multiple storage directories.

The fsimage file is a persistent checkpoint of the filesystem metadata. However, it is not updated for every filesystem write operation, because writing out the fsimage file, which can grow to be gigabytes in size, would be very slow. This does not compromise resilience, however, because if the namenode fails, then the latest state of its metadata can be reconstructed by loading the fsimage from disk into memory, and then applying each of the operations in the edit log.

EditLog

The NameNode also uses a transaction log to persistently record every change that occurs in filesystem metadata (metadata store). This log is stored in the EditLog file on the NameNode's local filesystem. Edit log is a transactional log of every filesystem metadata change since the image file was created.

The Name Node uses a transaction log called the Edit Log to persistently record every change that occurs to file system metadata. For example, creating a new file in HDFS causes the Name Node to insert a record into the Edit Log indicating this. Similarly, changing the replication factor of a file causes a new record to be inserted into the Edit Log. The Name Node uses a file in its local host OS file system to store the Edit Log.

Backup Node

The Backup node provides the same checkpointing functionality as the Checkpoint node, as well as maintaining an in-memory, up-to-date copy of the file system namespace that is always synchronized with the active Name Node state. Along with accepting a journal stream of file system edits from the Name Node and persisting this to disk, the Backup node also applies those edits into its own copy of the namespace in memory, thus creating a backup of the namespace.

The Backup node does not need to download fsimage and edits files from the active NameNode in order to create a checkpoint, as would be required with a Checkpoint node or Secondary NameNode, since it already has an up-to-date state of the namespace state in memory. The Backup node checkpoint process is more efficient as it only needs to save the namespace into the local fsimage file and reset edits. As the Backup node maintains a copy of the namespace in memory, its RAM requirements are the same as the NameNode. The NameNode supports one Backup node at a time. No Checkpoint nodes may be registered if a Backup node is in use. Using multiple Backup nodes concurrently will be supported in the future.

The Backup node is configured in the same manner as the Checkpoint node. It is started with `bin/hdfs namenode -checkpoint`. The location of the Backup (or Checkpoint) node and its accompanying web interface are configured via the `dfs.backup.address` and `dfs.backup.http.address` configuration variables. Use of a Backup node provides the option of running the NameNode with no persistent storage, delegating all responsibility for persisting the state of the namespace to the Backup node. To do this, start the NameNode with the `-i`

Block Scanner

Each DataNode runs a block scanner that periodically scans its block replicas and verifies that stored checksums match the block data. In each scan period, the block scanner adjusts the read bandwidth in order to complete the verification in a configurable period. If a client reads a complete block and checksum verification succeeds, it informs the DataNode. The DataNode treats it as a verification of the replica.

The verification time of each block is stored in a human readable log file. At any time there are up to two files in toplevel DataNode directory, current and prev logs. New verification times are appended to current file. Correspondingly each DataNode has an in-memory scanning list ordered by the replica's verification time.

Whenever a read client or a block scanner detects a corrupt block, it notifies the NameNode. The NameNode marks the replica as corrupt, but does not schedule deletion of the replica immediately. Instead, it starts to replicate a good copy of the block. Only when the good replica count reaches the replication factor of the block the corrupt replica is scheduled to be removed. This policy aims to preserve data as long as possible. So even if all replicas of a block are corrupt, the policy allows the user to retrieve its data from the corrupt replicas

Failure Types

The primary objective of the HDFS is to store data reliably even in the presence of failures. The three common types of failures are

- NameNode failures
- DataNodes failures
- network partitions

Several things can cause loss of connectivity between name node and data nodes. Therefore, each data node is expected to send a periodic heartbeat messages to its name node. This is required to detect loss of connectivity if it stops receiving them. The name node marks data nodes as dead data nodes if they are not responding to heartbeats and refrains from sending further requests to them. Data stored on a dead node is no longer available to an HDFS client from that node, which is effectively removed from the system.

Failover Controller

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, but the first implementation uses ZooKeeper to ensure that only one namenode is active. 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, since the failover controller arranges an orderly transition for both namenodes to switch roles

Hadoop Configuration

Every node in the Hadoop cluster must have a copy of the configuration files, including the client nodes. The configurations in the file are applied as follows (in the order of highest to lowest):

1. Values specified in the JobConf or Job object when initializing the MapReduce job
2. *-site.xml on the client node
3. *-site.xml on the slave node
4. Default values from the *-default.xml, which are the same on all the nodes

Core-*.xml

core-default.xml Default core Hadoop properties. The file is located in the following JAR file: `hadoop-common-2.2.0.jar` (assuming version 2.2.0).

core-site.xml Site-specific common Hadoop properties. Properties configured in this file override the properties in the `core-default.xml` file.

`fs.defaultFS` `hdfs://localhost:9000`

The `core-default.xml` and `core-site.xml` files configure the common properties for the Hadoop system. This section explores the key properties of `core-site.xml`, including the following:

- `hadoop-tmp-dir`: Base for other temporary directories. The default value is `/tmp/hadoop- ${user.name}`. We referenced this property on a few occasions as the root directory for several properties in the `hdfs-site.xml` file.
- `fs.defaultFS`: Name of the default path prefix used by HDFS clients when none is provided. In Chapter 3, “Getting Started with the Hadoop Framework,” you configured it to be `hdfs://localhost:9000` for the pseudo-cluster mode. This property specifies the name and port for the NameNode (for example, `hdfs://:9000`). For local clusters, the value of this property is `file:///`. When the High Availability (HA) feature of HDFS is used, this property should be set to the logical HA URI. (See the Hadoop documentation for configuration of HDFS HA.)
- `io.file.buffer-size`: Chapter 2, “Hadoop Concepts,” described the mechanics of the HDFS file create and HDFS file read processes. This property is relevant to those processes. It refers to the size of buffer to stream files. The size of this buffer should be a multiple of hardware page size (4096 on Intel x86), and it determines how much data is buffered during read and write operations. The default is 4096.

- `io.bytes.per.checksum`: Hadoop transparently applies checksums to all data written to it and verifies them at the time of reading. This parameter defines the number of bytes to which a checksum is applied. The default value is 512 bytes. The applied CRC-32 checksum is 4 bytes long. Thus, by default there is an overhead of approximately 1% per 512 bytes stored (with the default setting). Note that this parameter must not be a higher value than the `io.file.buffer-size` parameter because the checksum will be calculated on the data while in the memory where streaming during the HDFS read/write process. It needs to be calculated during the read process to verify the checksums stored during the write process.
- `io.compression.codecs`: A comma-separated list of the available compression codec classes that can be used for compression/decompression. The default settings that indicate the available compression codecs are `org.apache.hadoop.io.compress.DefaultCodec`, `org.apache.hadoop.io.compress.GzipCodec`, `org.apache.hadoop.io.compress.BZip2Codec`, `org.apache.hadoop.io.compress.DeflateCodec`, and `org.apache.hadoop.io.compress.SnappyCodec`.

hdfs-*.xml

hdfs-default.xml Default HDFS properties. The file is located in the following JAR file: `hadoop-hdfs-2.2.0.jar` (assuming version 2.2.0).

hdfs-site.xml Site-specific HDFS properties. Properties configured in this file override the properties in the `hdfs-default.xml` file.

The `hdfs-default.xml` and `hdfs-site.xml` files configure the properties for the HDFS. Together with the `core-site.xml` file described next, the HDFS is configured for the cluster. As you learned in Chapter 2, NameNode and Secondary NameNode are responsible for managing the HDFS. The *hdfs-.xml files configure the NameNode and the Secondary NameNode components of the Hadoop system. The hdfs-.xml set of files are also used for configuring the runtime properties of the HDFS as well the properties associated with the physical storage of files in HDFS on the individual data nodes.* Although the list of properties covered in this section is not exhaustive, it provides a deeper understanding of the HDFS design at the physical and operational level. This section explores the key properties of the `hdfs-*.xml` file. Some of the important properties of the `hdfs-site.xml` file include the following:

- **dfs.namenode.name.dir:** Directories on the local file system of the NameNode in which the metadata file table (the `fsimage` file) is stored. Recall that this file is used to store the HDFS metadata since the last snapshot. If this is a comma-delimited list of directories, the file is replicated to all the directories for redundancy. (Ensure that there is no space after the comma in the comma-delimited list of directories.) The default value for this property is `file://${hadoop.tmp.dir}/dfs/name`. The `hadoop.tmp.dir` property is specified in the `core-site.xml` (or `core-default.xml` if `core-site.xml` does not override it).
- **dfs.namenode.edits.dir:** Directories on the local file system of the NameNode in which the metadata transaction file (the `edits` file) is stored. This file contains changes to the HDFS metadata since the last snapshot. If this is a comma-delimited list of directories, the transaction file is replicated to all the directories for redundancy. The default value is the same as `dfs.namenode.name.dir`.
- **dfs.namenode.checkpoint.dir:** Determines where the Secondary NameNode should store the temporary images to merge on the local/network file system accessible to the Secondary NameNode. Recall from Chapter 2 that this is the location where the `fsimage` file from the NameNode is copied into for merging with the `edits` file from the NameNode. If this is a comma-delimited list of directories, the image is replicated in all the directories for redundancy. The default value is `file://${hadoop.tmp.dir}/dfs/name/secondary`.
- **dfs.namenode.checkpoint.edits.dir:** Determines where the Secondary NameNode should store the `edits` file copied from the NameNode to merge the `fsimage` file copied in the folder defined by the `dfs.namenode.checkpoint.dir` property on the local/network file system accessible to the Secondary NameNode. If it is a comma-delimited list of directories, the `edits` files are replicated in all the directories for redundancy. The default value is the same as `dfs.namenode.checkpoint.dir`.
- **dfs.namenode.checkpoint.period:** The number of seconds between two checkpoints. As an interval equal to this parameter elapses, the checkpoint process begins, which merges the `edits` file with the `fsimage` file from the NameNode.
- **dfs.blocksize:** The default block size for new files, in bytes. The default is 128 MB. Note that block size is not a system-wide parameter; it can be specified on a per-file basis.
- **dfs.replication:** The default block replication. Although it can be specified per file, if not specified it is taken as the replication factor for the file. The default value is 3.
- **dfs.namenode.handler.count:** Represents the number of server threads the NameNode uses to communicate with the DataNodes. The default is 10, but the recommendation is about 10% of the number of nodes, with a minimum value of 10. If this value is too low, you might notice messages in the DataNode logs indicating that the connection was refused by the NameNode when the DataNode tried to communicate with the NameNode through heartbeat messages.

- **dfs.datanode.du.reserved**: Reserved space in bytes per volume that represents the amount of space to be reserved for non-HDFS use. The default value is 0, but it should be at least 10 GB or 25% of the total disk space, whichever is lower.
- **dfs.hosts**: This is a fully qualified path to a file name that contains a list of hosts that are permitted to connect with the NameNode. If the property is not set, all nodes are permitted to connect with the NameNode.

mapred-*.xml

mapred-default.xml Default MapReduce properties. The file is located in the following JAR file: hadoop-mapreduce-client-core-2.2.0.jar (assuming version 2.2.0).

mapred-site.xml Site-specific MapReduce properties. Properties configured in this file override the properties in the mapred-default.xml file.

yarn-*.xml

yarn-default.xml Default YARN properties. The file is located in the following JAR file: hadoop-yarn-common-2.2.0.jar (assuming version 2.2.0).

yarn.xml Site-specific YARN properties. Properties configured in this file override the properties in the mapred-default.xml file.