Handling huge volumes of data generating from billions of online activities and transactions requires continuous upgradation and evolution of Big Data. One such upcoming technology is Hadoop. People often confuse Hadoop with Big Data. Hadoop is not Big Data; however, it is also true that Hadoop plays an integral part in almost all Big Data processes. In fact, it is almost impossible to use Big Data without the tools and techniques of Hadoop. So what exactly is Hadoop?

According to Apache, "Hadoop is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures." In simple words, Hadoop is a 'software library' that allows its users to process large datasets across distributed clusters of computers, thereby enabling them to gather, store and analyze huge sets of data. Hadoop provides various tools and technologies, collectively termed as the Hadoop ecosystem, to enable development and deployment of Big Data solutions.

Hadoop Distributed File System (HDFS) is a resilient, flexible, grouped method of file management in a Big Data setup. HDFS is a data service offering unique abilities required when data variety, volumes, and velocity are beyond the controllable levels of traditional data management systems. In case of traditional data management systems, data is read many times after it is inserted into a system but written only once in its lifetime. However, in the case of systems performing continuous read-write cycles, HDFS offers excellent Big Data analysis and support. HDFS consists of a central DataNode and multiple DataNodes running on the appliance model cluster and offers the highest performance levels when the same physical rack is used for the entire cluster. HDFS is capable of managing the inbuilt environment redundancy and its design can detect failures and resolve them by running particular programs automatically on multiple servers present in the cluster. HDFS allows simultaneous application execution across multiple servers consisting of economical internal disk drives.

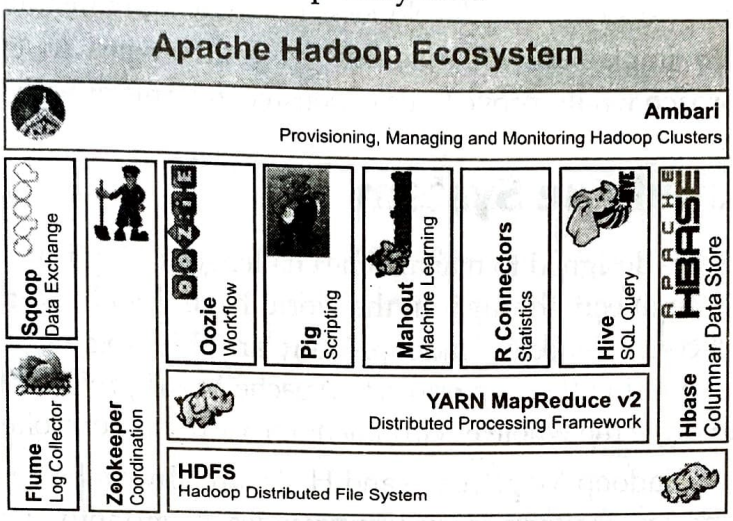
This session introduces you to the various elements of the Hadoop ecosystem, such as HDFS and its architecture, MapReduce, YARN, HBase, and Hive. You will also learn about various tools used to interact with Hadoop ecosystem including Pig and Pig Latin, Sqoop, ZooKeeper, Flume, and Oozie in later chapters. In the end, the chapter discusses the need for selecting a suitable Hadoop data organization for applications.

Hadoop Ecosystem

Hadoop ecosystem is a framework of various types of complex and evolving tools and components. Some of these elements may be very different from each other in terms of their architecture; however, what keeps them all together under a single roof is that they all derive their functionalities from the scalability and power of Hadoop. In simple words, the Hadoop ecosystem can be defined as a comprehensive collection of tools and technologies that can be effectively implemented and deployed to provide Big Data solutions in a cost-effective manner. **MapReduce** and **Hadoop Distributed File System (HDFS)** are two core components of the Hadoop ecosystem that provide a great starting point to manage Big Data; however, they are not sufficient to deal with the Big Data

challenges. Along with these two, the Hadoop ecosystem provides a collection of various elements to support the complete development and deployment of Big Data solutions.

Figure 4.1 depicts the elements of the Hadoop ecosystem:



Source: *http://blog.agro-know.com/?p=3810*

Figure 4.1: Hadoop Ecosystem

All these elements enable users to process large datasets in real time and provide tools to support various types of Hadoop projects, schedule jobs, and manage cluster resources.

Figure 4.2 depicts how the various elements of Hadoop involve at various stages of processing data:

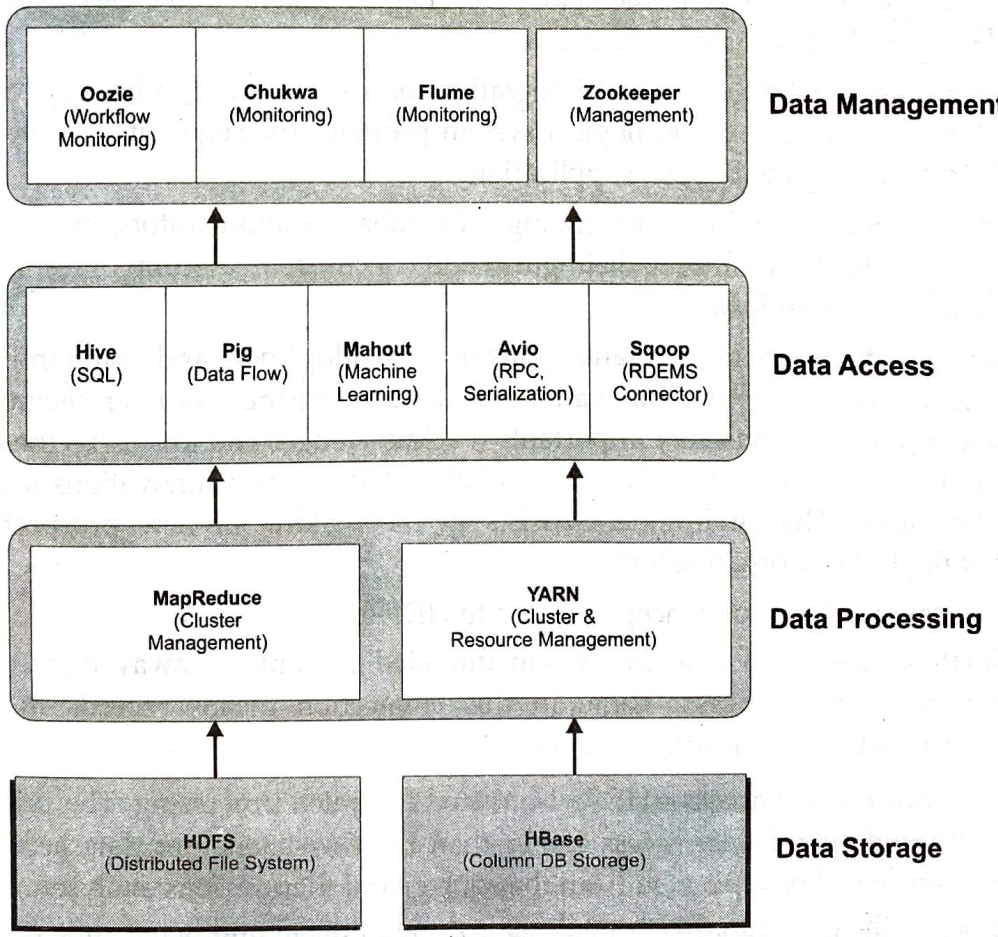


Figure 4.2: Hadoop Ecosystem Elements at Various Stages of Data Processing

MapReduce and HDFS provide the necessary services and basic structure to deal with the core requirements of Big Data solutions. Other services and tools of the ecosystem provide the environment and components required to build and manage purpose-driven Big Data applications. In the absence of an ecosystem, the developers, database administrators, system and network managers will have to implement separate sets of technologies to create Big Data solutions. However, such an approach would prove to be expensive in terms of both time and money.

Hadoop Distributed File System

File systems like HDFS are designed to manage the challenges of Big Data. For some time now, open source and commercial engineers throughout the world have been building and testing instruments to expand the role and convenience of Hadoop. Many are chipping away at the components of the Hadoop ecosystem and offering their upgrades to Apache-based projects. This steady stream of fixes and upgrades serves to drive the whole environment forward in a controlled and secure way.

Being core components, Hadoop MapReduce and HDFS are always being enhanced and hence they provide greater stability. The Hadoop ecosystem provides a constantly increasing collection of tools and technology particularly made to abridge the deployment, development and support of Big Data setups. Attempting to handle enormous information challenges without a toolkit loaded with innovation and credibility is similar to attempting to drain the sea with a spoon.

No building is steady without a solid base. Each part of the building must justify its existence. The walls, stairs, floors, electrical equipment, pipes—all need to support one another for the building to exist as one strong entity, similar to the Hadoop ecological system. In the case of Hadoop, the base consists of HDFS and MapReduce.

Both give the fundamental structure and integration services required to help the core condition of Big Data systems. The rest of the ecological system provides the segments you need to build and oversee goal-driven, real-time Big Data applications.

Without an ecosystem, it would be impending on database administrators, engineers, system and network managers, etc., to commonly distinguish and concur upon a group of technologies to create and introduce Big Data solutions.

This is routinely a situation where organizations need to adopt new and rising innovation patterns and the errand of fixing together innovations in another business is overwhelming. That is the reason the Hadoop ecosystem is very important: it acknowledges and unleashes the net capability of Big Data. It is the most comprehensive accumulation of tools and innovations available today to target Big Data tests. The environment assists in the making of new prospects for extensive deployment of Big Data by organizations.

Let's now discuss some terms or concepts related to HDFS:

- **Huge documents**—HDFS is a file system intended for putting away huge documents with streaming information access. Huge in this connection means records in the vicinity of gigabytes, terabytes or even petabytes in size.
- **Streaming information access**—HDFS is created for batch processing. The priority is given to the high throughput of data access rather than the low latency of data access. A dataset is commonly produced or replicated from the source, and then various analyses are performed on that dataset in the long run. Every analysis is done in detail and hence time consuming, so the time required for examining the entire dataset is quite high.

- **Appliance hardware**—Hadoop does not require large, exceptionally dependable hardware to run.
- **Low-latency information access**—Applications that permit access to information in milliseconds do not function well with HDFS. Therefore, HDFS is upgraded for conveying a high transaction volume of information, and this may be at the expense of idleness. HBase is at present a superior choice for small latency access.
- **Loads of small documents**—Since the NameNode holds file system data information in memory, the quantity of documents in a file system is administered in terms of the memory on the server. As a dependable guideline, each document and registry takes around 150 bytes. Thus, for instance, if you have one million documents, you would require no less than 300 MB of memory. While putting away a huge number of records is achievable, billions is past the capacity of the current equipment.

HDFS Architecture

HDFS has a master-slave architecture. It comprises a NameNode and a number of DataNodes. The NameNode is the master that manages the various DataNodes, as shown in Figure 4.3:

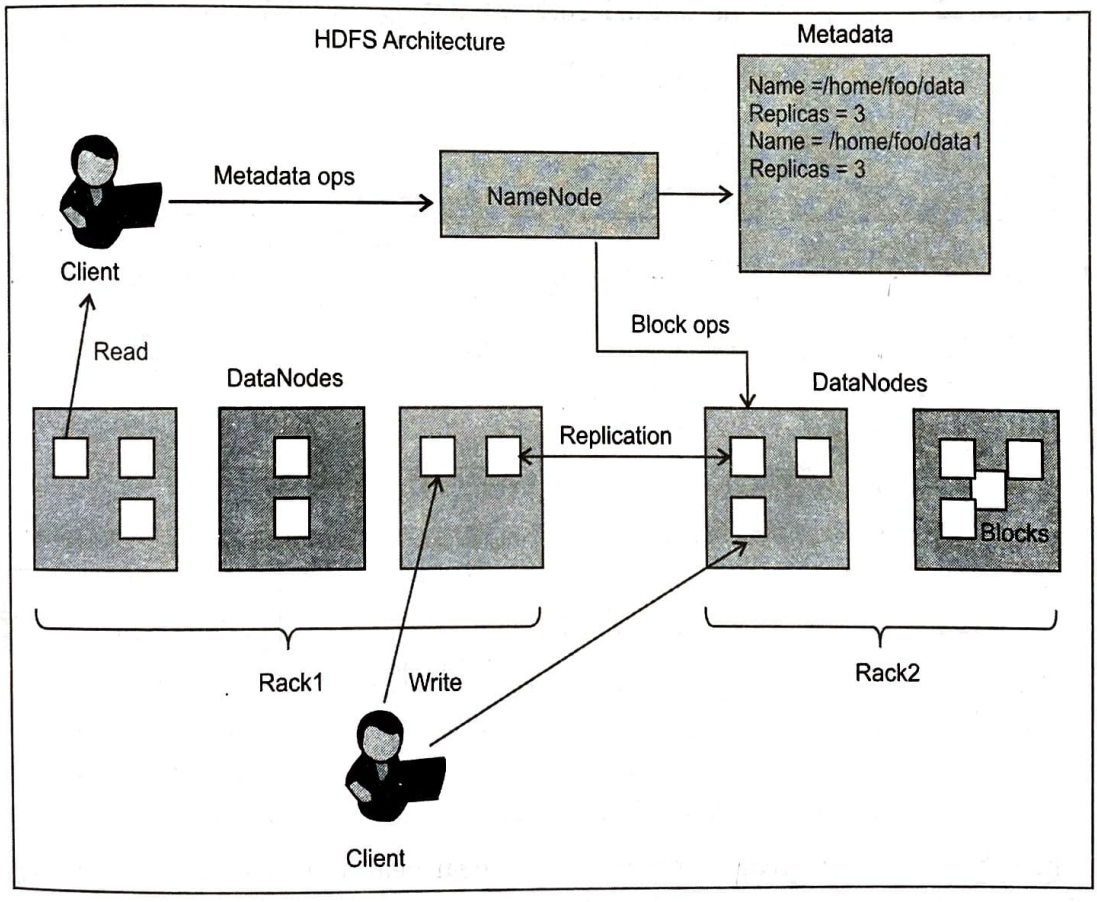


Figure 4.3: Displaying the Architecture of HDFS

The NameNode manages HDFS cluster metadata, whereas DataNodes store the data. Records and directories are presented by clients to the NameNode. These records and directories are managed on the NameNode. Operations on them, such as their modification or opening and closing them are performed by the NameNode. On the other hand, internally, a file is divided into one or more blocks, which are stored in a group of DataNodes. DataNodes read and write requests from the

clients. DataNodes can also execute operations like the creation, deletion, and replication of blocks, depending on the instructions from the NameNode.

Concept of Blocks in HDFS Architecture

A disk has a certain block size, which is the basic measure of information that it can read or compose. File systems expand by managing information in pieces, which are an indispensable part of the disk block size. File system blocks are commonly a couple of kilobytes in size, while disk blocks are regularly 512 bytes. This is by and large straightforward for the file system client that is essentially perusing or composing a record of whatever length.

HDFS blocks are huge in contrast to disk blocks because they have to minimize the expense of the seek operation. Consequently, the time to transfer a huge record made of multiple blocks operates at the disk exchange rate. A quick computation demonstrates that if the seek time is around 10ms, and the exchange rate is 100 MB/s, then to assign the seek time that is 1% of the exchange time, we have to create a block size of around 100 MB. The default size is 64 MB, although numerous HDFS installations utilize 128 MB blocks. Map tasks of MapReduce (component of HDFS) typically work on one block at once, so if you have fewer assignments (fewer than the nodes in the group), your tasks will run slower. Figure 4.4 shows the Heartbeat message of Hadoop:

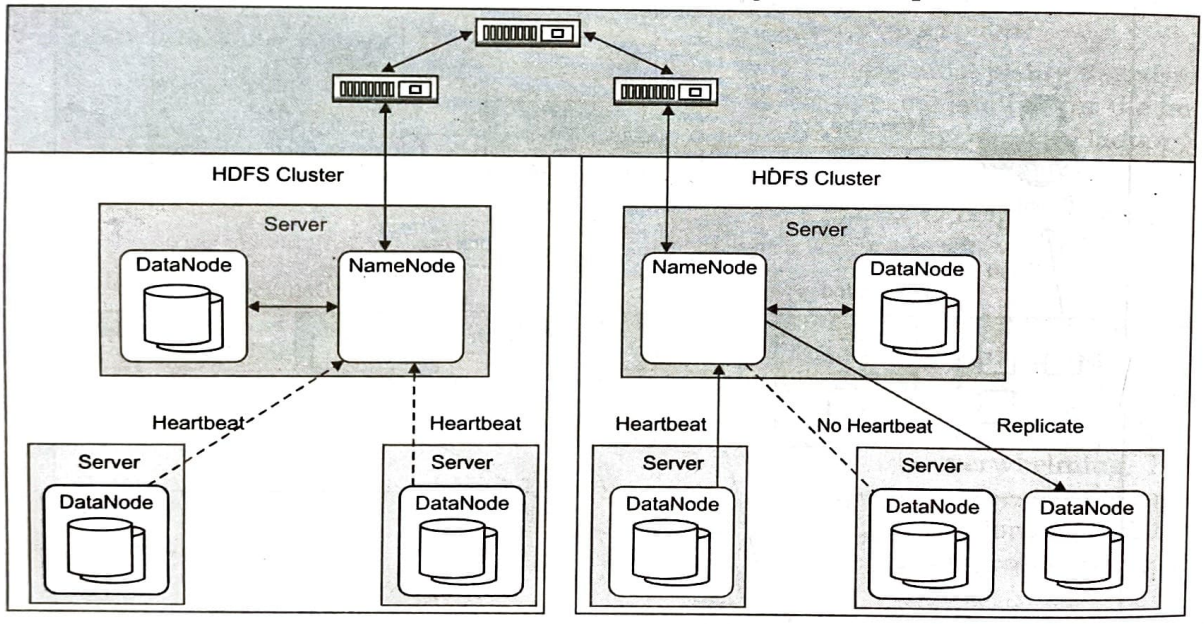


Figure 4.4: Illustration of Hadoop Heartbeat Message

When a heartbeat message reappears or a new heartbeat message is received, the respective DataNode sending the message is added to the cluster.

HDFS performance is calculated through distribution of data and fault tolerance by detecting faults and quickly recurring the data. This recovery is completed through replication and resents in a reliable file system. It results in a reliable huge file storage.

To enable this task of reliability, one should facilitate number of tasks for failure management, some of which are utilized within HDFS and others are still in a process to be implemented:

- **Monitoring**—DataNode and NameNode communicate through continuous signals ("Heartbeat"). If signal is not heard by either of the two, the node is considered to have failed and would be no longer available. The failed node is replaced by the replica and replication scheme is also changed.
- **Rebalancing**—According to this process, the blocks are shifted from one to another location where ever the free space is available. Better performance can be judged by the increase in demand of data as well as the increase in demand for replication towards frequent node failures.
- **Metadata replication**—These files are prove to failures; however, they maintain the replica of the corresponding file on the same HDFS.

There are a few advantages of abstracting a block for a distributed file system. The principal advantage is most evident: a record can be bigger than any single disk in a system. There is nothing that requires the blocks from a record to be put away on the same disk; so, they can exploit any of the disks in the cluster. It would still be possible to store a single document on a HDFS cluster with the entire cluster disks filled by its blocks.

Second, making the abstraction unit a block instead of a file improves the storage subsystem. The storage subsystem manages blocks, improving storage management (since blocks are of a fixed size, it is not difficult to compute the number of blocks that can be stored on a disk), and dispensing with metadata concerns.

To protect against corrupt blocks and disks and machine failure, each block is recreated on a few separate machines (usually three). In the event that a block gets occupied, a copy or duplicate block can be read from an alternate location in a straightforward manner to the client. A block that is no more accessible because of such issues can be replicated from its alternative location to other live machines.

Just like a disk file system, HDFS's fsck command can issue directives to blocks. For example, running the *% hadoop fsck -files -blocks* command lists the blocks that create each file in a file system.

NameNodes and DataNodes

An HDFS cluster has two node types working in a slave master design: a NameNode (the master) and various DataNodes (slaves). The NameNode deals with the file system. It stores the metadata for all the documents and indexes in the file system. This metadata is stored on the local disk as two files: the file system image and the edit log. The NameNode is aware of the DataNodes on which all the pieces of a given document are found; however, it doesn't store block locations necessarily, since this data is recreated from DataNodes.

A client accesses the file system on behalf of the user by communicating with the DataNodes and NameNode. The client provides a file system, like the POSIX interface, so the user code does not require the NameNode and DataNodes in order to execute.

DataNodes are the workhorses of a file system. They store and recover blocks when they are asked to (by clients or the NameNode), and they report back to the NameNode occasionally with a list of blocks that they store externally.

Without the NameNode, the file system cannot be used. In fact, if the machine using the NameNode crashes, all files on the file system would be lost since there would be no way of knowing how to reconstruct the files from the blocks on DataNodes. This is why it is important to make the NameNode robust to cope with failures, and Hadoop provides two ways of doing this.

The first way is to take the back up of file documents. Hadoop can be set in a way that NameNode creates its state for various file systems. The normal setup choice is to write to the local disk and to a remote NFS mount.

Another way is to run a secondary NameNode, which does not operate like a normal NameNode.

Secondary NameNode periodically reads the filesystem, changes the log, and apply them into the fsimage file. When the NameNode is down, secondary NameNode will be online but this node will only have read permissions to fsimage and editlog file.

DataNodes ensure connectivity with the NameNode by sending heartbeat messages. Whenever the NameNode ceases to receive a heartbeat message from a DataNode, it unmaps the DataNode from the cluster and proceeds with further operations.

The Command-Line Interface

We are going to view HDFS by interfacing with it from the command line. There are numerous different interfaces for HDFS; however, the command line is one of the easiest and most popular among developers. We are going to execute HDFS on a machine. For this, we need to first set up Hadoop in a distributed mode prototype.

There are two properties that we set in the distributed mode prototype setup that calls for further clarification. The principal is fs.default.name, set to `hdfs://localhost/`, which is used to set a default Hadoop file system. File systems are tagged by a URI, and here we have utilized an HDFS URI to design Hadoop. The HDFS daemons will utilize this property to focus the host and port for the HDFS NameNode. We'll be running it on localhost, on the default HDFS port, 8020. Furthermore, HDFS clients will utilize this property to figure out where the NameNode is running so they can connect with it.

Using HDFS Files

The HDFS file system is accessed by user applications with the help of the HDFS client. It is a library that reveals the interface of the HDFS file system and hides almost all the complexities that appear during the implementation of HDFS.

The user application is not required to know whether the metadata of the file system and its storage are on the same servers or have multiple replicas.

The object of a Filesystem class is created for accessing HDFS. The Filesystem class is an abstract base class for a generic file system. The code created by a user referring to HDFS must be written in order to use an object of the Filesystem class. An instance of the FileSystem class can be created by passing a new Configuration object into a constructor. Assume that Hadoop configuration files, such as hadoop-default.xml and hadoopsite.xml are present on the class path.

The code to create an instance of the FileSystem class is shown in Listing 4.1:

Listing 4.1: Creating a FileSystem Object

```
Configuration config = new Configuration();

FileSystem fsys = FileSystem.get(config);
```

Creates a configuration object config

Creates a FileSystem object fs

Path is another important HDFS object, which specifies the names of the files or directories in a file system. You can create a Path object from a string specifying the location of the file/directory on HDFS. Both the FileSystem and Path objects allow you to perform programmatic operations on HDFS files and directories.

Listing 4.2 shows the manipulation of HDFS objects:

Listing 4.2: Manipulating HDFS Objects

```
Path fp=new Path(file name);

if (fsys.exists(fp))
//statement 1
If(fsys.isFile(fp))
//statement 1
Boolean result=fsys.createNewFile(fp);
Boolean result=fsys.delete(fp);

FSDataInputStream fin=fsys.open(fp);

FSDataOutputStream fout=fsys.create(fp);
```

Creating an object for the Path class

Checking the file path

Reading from the file

Writing to the file

You must note that whenever a file is opened to perform the writing operation, the client opening this file grants an exclusive writing lease for it. Due to this, no other client can perform write operations on this file until the operation of this client gets completed. In order to ensure that a lease is held by no "runaway" clients, it gets expired periodically. The effective use of a lease ensures that no two applications can perform the write operation to a given file simultaneously.

The lease duration is bounded by a soft limit and a hard limit. In the case of the soft limit, a writer gets access to a file exclusively. In case the soft limit expires and the client fails to close the file or renew the lease, another client can preempt the lease. On the other hand, if the hard limit expires and the client fails to renew the lease, HDFS assumes that the client has quitted and closes the file automatically.

Hadoop-Specific File System Types

In addition to ordinary files, HDFS also provides several specific file system types that provide richer functionality and simplify the processing of data. The local file system of Hadoop performs the client-side checksum operation. When you write a record to a file, the client of the file system directly creates a hidden document, `.filename.crc`, in the same index containing checksums for each chunk of the document. The size of the chunk is managed by the `io.bytes.per.checksum`

property, which is 512 bytes by default. The chunk size is stocked up as metadata in the .crc document, so the file can be read correctly regardless of the changed setting for the chunk size. Checksums are checked when a record is analyzed, and if an error is found, the local file system throws a checksumexception exception.

NOTE

A checksum refers to the number of bits in a transmission unit included with the unit to enable the receiver to see whether the same number of bits has arrived.

Checksums are economical to calculate (in Java, they are executed in local code), normally adding a marginal overhead to the time to analyze or write a record. For most systems, the cost is worth the value additions it brings to the application. The disabling of checksums is possible. This is achieved by using the `Rawlocalfilesystem` class instead of the `Localfilesystem` class. Table 4.1 lists some Hadoop-specific file systems types:

Table 4.1: Hadoop-Specific File System Types

File System	URI Scheme	Java Implementation (org.apache.hadoop)	Definition
Local	file	fs.LocalFileSystem	A file system for a locally connected disk with client-side check-sums. Use `RawLocalFileSystem` for a local file system with no checksums.
HDFS	hdfs	hdfs.DistributedFile System	Hadoop's distributed file system. HDFS is designed to work efficiently with MapReduce.
HFTP	hftp	hdfs.HftpFileSystem	A file system providing read-only access to HDFS. (Despite its name, HFTP has no link with FTP.) Often used with distcp to copy data between HDFS clusters running different versions.
HSFTP	hsftp	hdfs.HsftpFileSystem	A file system providing read-only access to HDFS over HTTPS. (Again, this has no connection with FTP.)
HAR	har	fs.HarFileSystem	A file system layered on another file system for archiving files. Hadoop archives are typically used for archiving files in HDFS to reduce the NameNode's memory usage.
KFS (CloudStore)	kfs	fs.kfs.KosmosFile System	CloudStore (formerly the Kosmos file system) is a distributed file system like HDFS or Google's GFS, written in C++.
FTP	ftp	fs.ftp.FTPFileSystem	A file system backed by an FTP server.
S3 (native)	s3n	fs.s3native.NativeS3 FileSystem	A file system backed by Amazon S3.

Table 4.1: Hadoop-Specific File System Types			
File System	URI Scheme	Java Implementation (org.apache.hadoop)	Definition
S3 (blockbased)	s3	fs.s3.S3FileSystem	A file system backed by Amazon S3, which stores files in blocks (much like HDFS) to overcome S3's 5 GB file size limit.

HDFS Commands

Various shell like commands interact with HDFS directly as well as with other file systems supported by Hadoop such as Local FS, HFTP FS, S3 FS, and others. These commands are provided by the File System (FS) shell. The FS shell can be invoked by the following command:

```
bin/hadoop fs <args>
```

Most commands in the FS shell are similar to the Unix commands and perform almost similar functions. Some commonly used HDFS commands are shown in Table 4.2:

Table 4.2: HDFS Commands and their Description		
Commands	Description	Syntax
appendToFile	Used for appending a single src or multiple srcs from the local file system to the destination file system. This command also reads input from stdin and appends to the destination file system.	hdfs dfs -appendToFile <localsrc> ... <dst>
cat	Used for copying source paths to stdout. This command returns 0 on success and -1 in case of error.	Usage: hdfs dfs -cat URI [URI ...]
chmod	Used for changing the permissions of files.	hdfs dfs -chmod [-R] <MODE[,MODE]... | OCTALMODE> URI [URI ...]
chown	Used for changing the owner of files.	hdfs dfs -chown [-R] [OWNER][:[GROUP]] URI [URI]
count	Used for counting the number of directories, files, and bytes under the paths that match the mentioned file pattern.	hdfs dfs -count [-q] [-h] <paths>
cp	Used for copying files from the source to the destination. This command enables multiple sources as well in which case the destination must be a directory.	hdfs dfs -cp [-f] [-p | -p[topax]] URI [URI ...] <dest>

Table 4.2: HDFS Commands and their Description		
Commands	**Description**	**Syntax**
get	Used for copying files to the local file system.	hdfs dfs -get [-ignorecrc] [-crc] <src> <localdst>
mkdir	Used for creating directories by taking the path URI as an argument.	hdfs dfs -mkdir [-p] <paths>
mv	Used for moving files from the source to the destination.	hdfs dfs -mv URI [URI ...] <dest>
rm	Used for deleting files specified as args.	hdfs dfs -rm [-f] [-r | -R] [-skipTrash] URI [URI ...]

The org.apache.hadoop.io package

This `org.apache.hadoop.io` package provides generic I/O code to be used while reading and writing data to the network, to databases, and to files. This package provides various interfaces, classes, and exceptions. The interfaces provided by this package are listed in Table 4.3:

Table 4.3: Interfaces of the org.apache.hadoop.io Package and their Description	
Interface	**Description**
RawComparator<T>	Refers to a comparator that operates directly on byte representations of objects.
Stringifier<T>	Provides two methods for converting an object to a string representation and restore the object given in its string representation.
Writable	Refers to a serializable object that implements a simple, efficient, serialization protocol based on DataInput and DataOutput.
WritableComparable<T>	Refers to a Writable object which is also a Comparable object.
WritableFactory	Refers to a factory for a class of Writable.

Table 4.4 lists the classes of the `org.apache.hadoop.io` package:

Table 4.4: Classes of the org.apache.hadoop.io Package and their Description	
Class	**Description**
AbstractMapWritable	Refers to an abstract base class for MapWritable and SortedMapWritable. Unlike org.apache.nutch.crawl.MapWritable, this class enables the creation of MapWritable<Writable, MapWritable>.
ArrayFile	Performs a dense file-based mapping from integers to values.
ArrayPrimitiveWritable	Refers to a wrapper class.

Table 4.4: Classes of the org.apache.hadoop.io Package and their Description	
Class	**Description**
ArrayWritable	Acts as a Writable object for arrays containing instances of a class.
BinaryComparable	Refers to the interface supported by WritableComparable types that support ordering/permutation by a representative set of bytes.
BloomMapFile	Extends MapFile and provides almost the same functionality.
BooleanWritable	Refers to a WritableComparable object for booleans.
BytesWritable	Refers to a byte sequence that is usable as a key or value.
ByteWritable	Refers to a WritableComparable object for a single byte.
CompressedWritable	Refers to a base class for Writables that store themselves in a compressed form.
DataOutputOutputStream	Refers to an OutputStream implementation that wraps a DataOutput.
DefaultStringifier<T>	Refers to the default implementation of the Stringifier interface, which stringifies the objects using base64 encoding of the serialized version of the objects.
DoubleWritable	Refers to a Writable object for double values.
ElasticByteBufferPool	Refers to a simple ByteBufferPool that creates ByteBuffers as required.
EnumSetWritable<E extends Enum<E>>	Refers to a Writable wrapper for EnumSet.
FloatWritable	Refers to a WritableComparable object for floats.
GenericWritable	Refers to a wrapper for Writable instances.
IntWritable	Refers to a WritableComparable object for ints.
IOUtils	Refers to a utility class for I/O related functionality.
LongWritable	Refers to a WritableComparable object for longs.
MapFile	Performs a file-based mapping from keys to values.
MapWritable	Refers to a Writable Map.
MD5Hash	Refers to a Writable object for MD5 hash values.
NullWritable	Refers to a singleton Writable with no data.
ObjectWritable	Refers to a polymorphic Writable that writes an instance with its class name.
SequenceFile	Refers to flat files comprising binary key/value pairs.

Table 4.4: Classes of the org.apache.hadoop.io Package and their Description	
Class	**Description**
SetFile	Refers to a file-based set of keys.
ShortWritable	Refers to a WritableComparable object for shorts.
SortedMapWritable	Refers to a Writable SortedMap.
Text	Uses standard UTF8 encoding for storing text.
TwoDArrayWritable	Contains a matrix of instances of a class and is a writable for 2D arrays.
VersionedWritable	Refers to a base class for Writables that provides version checking.
VIntWritable	Refers to a WritableComparable object for integer values stored in the variable-length format.
VLongWritable	Refers to a WritableComparable for longs in a variable-length format.
WritableComparator	Refers to a comparator for WritableComparables.
WritableFactories	Refers to the factories for non-public writables.

Table 4.5 lists exceptions of the `org.apache.hadoop.io` package:

Table 4.5: Exoeptions of the `org.apache.hadoop.io` Package and their Description	
Exceptions	**Description**
MultipleIOException	Encapsulates a list of IOException into an IOException
VersionMismatchExcepti on	Occurs when the version of the object being read does not match with the current implementation version as returned by VersionedWritable.getVersion(), which throws this VersionedWritable.readFields(DataInput) exception.

HDFS High Availability

In an HDFS cluster, the NameNode was a Single Point Of Failure (SPOF) prior to Hadoop 2.0. Each Hadoop cluster contains a NameNode. The availability of an HDFS cluster depends upon the availability of the NameNode. In other words, the HDFS cluster would not be available if its NameNode cluster is not active.

This kind of situation affects the total availability of the HDFS cluster in mainly two ways:

- If an unplanned event such as a machine crash occurs, then the cluster would remain unavailable till an operator restarts the NameNode.
- Planned maintenance such as upgradation of software or hardware on the NameNode machine would result in cluster downtime.

The preceding problems are addressed by the HDFS High Availability feature which provides the facility of running two redundant NameNodes in the same cluster. These NameNodes can run in an

Active/Passive configuration which enables a fast failover to a new NameNode in case a machine crashes.

A typical HA cluster comprises two separate machines that are configured as NameNodes. At any instant, only one NameNode is in an Active state, and the other will be in the Standby state. The Active NameNode performs all the client operations in the cluster, and the Standby NameNode acts as a slave, which maintains enough state for providing a fast failover, if required.

You can deploy an HA cluster by preparing the following:

- **NameNode machines**—These are the machines on which you can run the Active and Standby NameNodes. These NameNode machines must have similar hardware configurations.
- **Shared storage**—Both NameNode machines must have read/write accessibility on a shared directory.

EXHIBIT 1: Streamlining Healthcare Connectivity with Hadoop and Big Data

Company Overview

The connectivity and information technology subsidiary of a major pharmaceutical company were created to simplify how the business of healthcare is managed while making the delivery of care safer and more efficient. As more and more of healthcare systems in the US go electronic, the company meets challenges and opportunities through an open network that supports future growth via interoperability among organizations, systems and solutions.

Business Challenges Before Cloudera

With regulations such as the Health Insurance Portability and Accountability Act of 1996 (HIPAA), healthcare organizations are required to store healthcare data for extended periods of time. This company instituted a policy of saving seven years' historical claims and remit data, but its in-house database systems had trouble meeting the data retention requirement while processing millions of claims every day.

A software engineer at the company explained, "All of our systems were maxed out. We were constantly having database issues. It was just too much data for what they were meant to handle. They were overworked and overloaded, and it started to cause problems with all of our real-time production processing."

Further, the company sought a solution that would allow users to do more than just store data. The manager of software development at the company explained, "In today's data driven world, data really is this huge asset. We wondered, 'What framework, what platform will allow us to optimize the data that we have?'"

The team set out to find a new solution. "We could have gone the SAN route, but it's expensive and cumbersome," said the software engineer. They did some searching online and came across Hadoop, MongoDB, and Cassandra. "We analyzed them and came up with a prototype for each one. In the end, we decided Hadoop was what we wanted."

Initially the company downloaded Hadoop from Apache and configured it to run on 10 Dell workstations that were already in house. Once the small Hadoop cluster showed its functionality and demonstrated value, the team decided to make a commitment to the platform, but needed support to do so. When evaluating various Hadoop distributions and management vendors, they recognized that Cloudera was different: its Hadoop distribution — CDH — is a 100% Apache open source. This allows Cloudera customers to benefit from rapid innovations in the open source community while also taking advantage of enterprise-grade support and management tools offered with the Cloudera Enterprise subscription.

When deciding to deploy CDH, the team set out to identify applications that were already seeing performance issues in production. "One of the big advantages of Hadoop has been to be able to segregate Big Data from transactional processing data and allow smoother processing of information. Basically, it allows us to offload a lot of stress from the database," said the company's manager of software development.

They quickly identified two areas that were a strong fit for Hadoop:

- Archiving seven years' claims and remit data, which requires complex processing to get into a normalized format
- Logging terabytes of data generated from transactional systems daily, and storing them in CDH for analytical purposes

Today, the company uses Flume to move data from its source systems into the CDH cluster on a 24x7 basis. The company loads data from CDH to an Oracle Online Transaction Processing (OLTP) database for billing purposes. This load runs once or twice each day via Sqoop.

Impact: Helping Providers Collect Payment Faster through Operational Efficiencies

"If you look at the margin that the average hospital has, it's between 2-3%," stated the manager of software development. "So their cash flow is very tight. Anything you can do to reduce the time to get paid is very valuable to a healthcare provider."

Since deploying Cloudera Enterprise, the company has reduced the time it takes for healthcare providers to get paid by streamlining their transfer of messages to payers. The ability to expedite this process is especially valuable when regulatory changes come into play, such as the recent conversion from HIPAA 4010 to HIPAA 5010.

"We assist with the conversion and processing of these messages," said the company's manager of software development. "For example, 4010 messages came in and we'd convert them to 5010 to allow seamless processing. The providers didn't have to upgrade any of their systems when the regulations went into effect. We gave them a bit of a buffer to implement changes. And since we do a lot of electronic processing, we can do basic sanity checks on the messages as they come in and let providers know what adjustments need to be made in order to get paid faster."

Impact: Low Cost + Greater Analytic Flexibility

Because Hadoop uses industry standard hardware, the cost per terabyte of storage is, on average, 10 times cheaper than a traditional relational data warehouse system. "One of my pet peeves is: you buy a machine, you buy SAN storage, and then you have to buy licensing for the storage in addition to the storage itself," explained the manager of software development. "You have to buy licensing for the blades, and it just becomes an untenable situation. With Hadoop you buy commodity hardware and you're good to go. In addition to the storage, you get a bigger bang for your buck because it gives you the ability to run analytics on the combined compute and storage. The solutions that we had in place previously really didn't allow for that. Even if the costs were equivalent, the benefit you get from storing data on a Hadoop type solution is far greater than what you'd get from storing it in a database."

Impact: Simple Deployment & Administration

After deciding on the Cloudera solution, "the deployment process into production with Hadoop was actually quite easy," said a software engineer at the company. "Cloudera Manager really helped us a lot. It's as easy as just clicking a few buttons, and you're up and running. It's really simple. And the support staff at Cloudera have been great. They really helped us out with a couple of issues we had along the way." Further, this company appreciates the proactive customer support offered by Cloudera Enterprise.

Source: http://hadoopilluminated.com/hadoop_illuminated/cached_reports/Cloudera_Case_Study_Healthcare.pdf

Features of HDFS

Data replication, data resilience, and data integrity are the three key features of HDFS. You have already learned that HDFS allows replication of data; thus automatically providing resiliency to data in case of an unexpected loss or damage of the contents of any data or location. Additionally, data pipelines are also supported by HDFS. A block is written by a client application on the first DataNode in the pipeline. The DataNode then forwards the data block to the next connecting node in the pipeline, which further forwards the data block to the next node and so on. Once all the data replicas are written to the disk, the client writes the next block, which undergoes the same process. This feature is supported by Hadoop MapReduce.

When a file is divided into blocks and the replicated blocks are distributed throughout the different DataNodes of a cluster, the process requires careful execution as even a minute variation may result in corrupt data.

HDFS ensures data integrity throughout the cluster with the help of the following features:

- **Maintaining Transaction Logs**—HDFS maintains transaction logs in order to monitor every operation and carry out effective auditing and recovery of data in case something goes wrong.
- **Validating Checksum**—Checksum is an effective error-detection technique wherein a numerical value is assigned to a transmitted message on the basis of the number of bits contained in the message. HDFS uses checksum validations for verification of the content of a file. The validations are carried out as follows:

1. When a file is requested by a client, the contents are verified using checksum.
2. If the checksums of the received and sent messages match, the file operations proceed further; otherwise, an error is reported.
3. The message receiver verifies the checksum of the message to ensure that it is the same as in the sent message. If a difference is identified in the two values, the message is discarded assuming that it has been tempered with in transition. Checksum files are hidden to avoid tempering.

- **Creating Data Blocks**—HDFS maintains replicated copies of data blocks to avoid corruption of a file due to failure of a server. The degree of replication, the number of DataNodes in the cluster, and the specifications of the HDFS namespace are identified and implemented during the initial implementation of the cluster. However, these parameters can be adjusted any time during the operation of the cluster.

Data blocks are sometimes, also called **block servers.** A block server primarily stores data in a file system and maintains the metadata of a block. A block server carries out the following functions:

- Storage (and retrieval) of data on a local file system. HDFS supports different operating systems and provides similar performance on all of them.
- Storage of metadata of a block on the local file system on the basis of a similar template on the NameNode.
- Conduct of periodic validations for file checksums.
- Intimation about the availability of blocks to the NameNode by sending reports regularly.
- On-demand supply of metadata and data to the clients where client application programs can directly access DataNodes.
- Movement of data to connected nodes on the basis of the pipelining model.

NOTE

A connection between multiple DataNodes that supports movement of data across servers is termed as a pipeline.

The manner in which the blocks are placed on the DataNodes critically affect data replication and support for pipelining. HDFS primarily maintains one replica of each block locally. A second replica of the block is then placed on a different rack to guard against rack failure. A third replica is maintained on a different server of a remote rack. Finally additional replicas are sent to random locations in local and remote clusters.

However, without proper supervision, one DataNode may get overloaded with data while another is empty. To address and avoid these possibilities, HDFS has a rebalancer service that balances the load of data on the DataNodes. The rebalancer is executed when a cluster is running and can be halted to avoid congestion due to network traffic. The rebalancer provides an effective mechanism; however, it has not been designed to intelligently deal with every scenario. For instance, the rebalancer cannot optimize for access or load patterns. These features can be anticipated from further releases of HDFS.

The HDFS balancer rebalances data over the DataNodes, moving blocks from over-loaded to under-loaded nodes. System administrator can run the balancer from the command-line when new DataNodes are added to the cluster.

NOTE

- *Superuser of HDFS has the capabilities to run balancer.*
- *Balancer will not balance between individual volumes on single DataNode.*

The command to run the balancer is as follows:

```
sudo -u hdfs hdfs balancer
```

MapReduce

The algorithms developed and maintained by the Apache Hadoop project are implemented in the form of Hadoop MapReduce, which can be assumed analogous to an engine that takes data input, processes it, generates the output, and returns the required answers.

MapReduce is based on the parallel programming framework to process large amounts of data dispersed across different systems. The process is initiated when a user request is received to execute the MapReduce program and terminated once the results are written back to the HDFS.

Nowadays, organizations need to quickly analyze the huge amount of gathered data and information to make effective and smart decisions. MapReduce facilitates the processing and analyzing of both unstructured and semi-structured data collected from different sources, which may not be analyzed effectively by other traditional tools.

MapReduce enables computational processing of data stored in a file system without the requirement of loading the data initially into a database. It primarily supports two operations: map and reduce. These operations execute in parallel on a set of worker nodes. MapReduce works on a master/worker approach in which the master process controls and directs the entire activity, such as collecting, segregating, and delegating the data among different workers.

Figure 4.5 depicts the working of MapReduce:

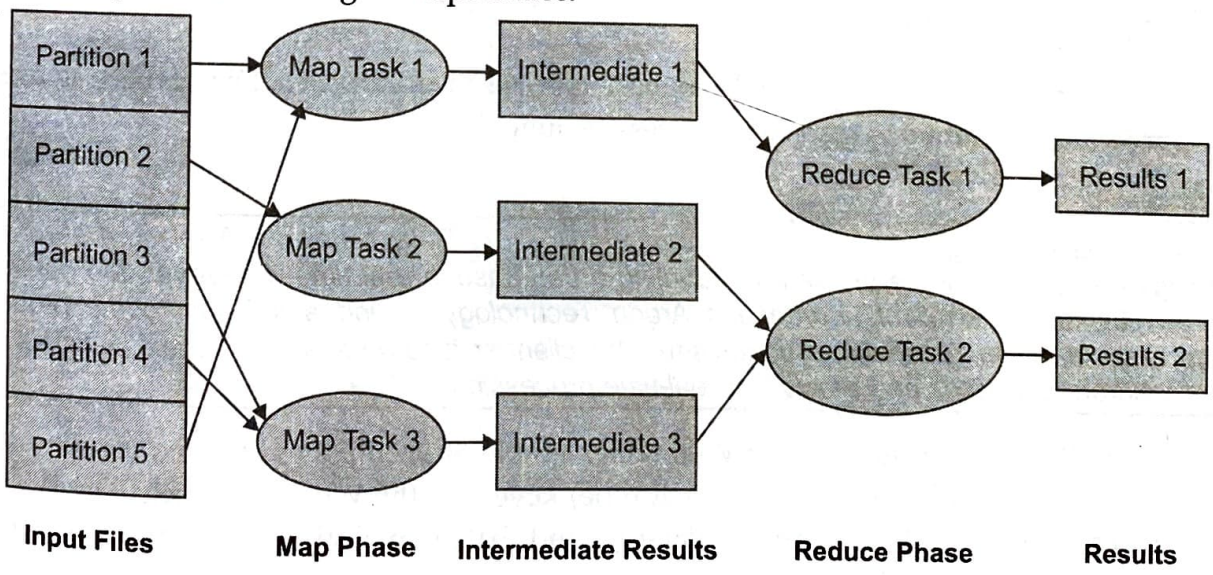


Figure 4.5: Working of MapReduce

The working of MapReduce can be summed up in the following steps:

1. A MapReduce worker receives data from the master, processes it, and sends back the generated result to the master.

2. MapReduce workers run the same code on the received data; however, they are not aware of other co-workers and do not communicate or interact with each other.
3. The master receives the results from each worker process, integrates and processes them, and generates the final output.

CASELET

Consider a case where Web logs are received as a steady stream, and then passed to various worker nodes for processing. A simple method to distribute and process incoming data blocks, in this case, is the round-robin procedure where every node receives the data packets sequentially. It can be further supported with some sort of hashing wherein data blocks are forwarded to workers on the basis of a formula so that the same worker receives similar data blocks. For example, implementing hashing on customer ID will direct all records of a particular customer to the same worker process.

Hadoop YARN

Job scheduling and tracking are primarily encoded in the functioning of Hadoop MapReduce. The early versions of Hadoop were accompanied with a rudimentary job and task-scheduling system. However, as the mix of work processed by Hadoop changed with time, the scheduler became obsolete.

The old scheduler was no more able to manage non-MapReduce jobs, and could not optimize cluster utilization. In order to address these shortcomings and provide more flexibility, efficiency, and performance boost, a new functionality was developed.

Yet Another Resource Negotiator (YARN) is a core Hadoop service that supports two major services Global resource management (ResourceManager) and Per-application management (ApplicationMaster). Global resource management (ResourceManager) and Per-application management (ApplicationMaster) have been discussed in detail in later chapters.

Introducing HBase

HBase is a column-oriented distributed database composed on top of HDFS. HBase is used when you need real-time continuous read/write access to huge datasets.

SCENARIO

Consider the case of Argon Technology, which provides Big Data solutions. A telecom client of Argon Technology has been facing a problem in handling a database in real time. Moreover, the data is getting bigger day by day. The client approaches Argon Technology to find a solution to the problem. The technical team at Argon Technology suggests that the client shift its database to HBase, which is proficient in handling large databases and allows fast real-time processing of data.

The standard HBase is considered as a Web table, a table of Web pages crawled (or accessed), and their properties (such as language and MIME type) keyed by the Web page URL. The Web table is large, containing over a billion rows. Parsing and batch analytics are MapReduce jobs that continuously run against the Web table.

These jobs derive statistics and add new columns of MIME type and parsed text content for later indexing by a search engine. Simultaneously, the Web table is accessed randomly by crawlers (search engines) running at various rates and updating random rows. The random pages in the Web table are served in real-time as users click on a website's cached-page feature.

Though there are innumerable techniques and implementations for database storage and recovery, most settings - particularly those with relational variety - are not constructed on a huge scale and allocation in mind. Most vendors offer replication and partition solutions to increase the database beyond the restrictions of a single node; yet these additional solutions are by and large an idea in retrospect and are complicated to install and maintain. Joins, triggers, complex queries, outside key requirements, and perspectives get to be incredibly expensive to run on a scaled RDBMS or do not work at all.

HBase takes the scaling issue head on and is constructed from the ground up scale just by adding nodes. HBase is not relational and thus does not support SQL, but it still has the capacity to do what an RDBMS cannot: host large, inadequately populated tables on clusters produced using appliance hardware.

HBase is one of the projects of Apache Software Foundation that is distributed under Apache Software License v2.0. It is a non-relational (columnar) database suitable for distributed environments and uses HDFS as its persistence storage. Modeled after **Google Big Table** (a way of storing non-relational data efficiently), HBase can host very large tables (billions of columns/rows). HBase facilitates reading/writing of Big Data randomly and efficiently in real time. It is highly configurable, allows efficient management of huge amount of data, and helps in dealing with Big Data challenges in many ways. It stores data into tables with rows and columns as in RDBMSs. The intersection of a row and column is called a cell. However, HBase tables have one key feature called versioning, which differentiates them from RDBMS tables. Each cell in an HBase table has an associated attribute termed as "version", which provides a timestamp to uniquely identify the cell. Versioning helps in keeping a track of the changes made in a cell and allows the retrieval of the previous version of the cell contents, if required.

HBase also provides various useful data processing features, such as scalability, persistence, sparseness, and support for distributed environments and multidimensional maps. HBase indexes are mapped with a row/column key pair and a timestamp. Each map value is represented by a continuous array of bytes. HBase also allows storage of results for later analytical processing.

HBase Architecture

Applications store information in labeled tables, and the values stored in table cells get updated from time to time. A cell's value is an unread array of bytes.

Keys in table rows are also byte arrays, so hypothetically anything can act as a row key. Table rows are categorized by the row key, which is the table's primary key. By default, the data type of the primary key is byte-ordered. All table access is via the table primary key.

Row/columns are clustered into column families. All family members of a column share a common prefix, e.g., the columns *java:android* and *java:servlets* are both members of the *java* family. The column family should consist of printable characters. The suffix can be made of any random bytes.

A table's column family members must be specified beforehand as an aspect of the table schema meaning, but new column family members can be added on a need basis. For example, a new column address:pincode can be provided by a client as part of an upgrade, and its value continued, as long as the column family exists in place on the targeted table.

Physically, all family members of a column are saved together on the file system. So, previously we described HBase as a column-oriented entity, but it would be more precise if it is described as a *column-family* oriented entity. Because storage requirements are fulfilled at the column family level, all column family members are recommended to have the same general access design and size features. In short, HBase tables are like the ones in RDBMS, with cells being versioned, sorted rows, and on-the-fly addition of columns as per the client's requirements.

Regions

Tables are automatically partitioned horizontally into regions by HBase. Each region consists of a subset of rows of a table. Initially, a table comprises a single region but as the size of the region grows, after it crosses a configurable size limit, it splits at the boundary of a row into two new regions of almost equal size, as shown in Figure 4.6:

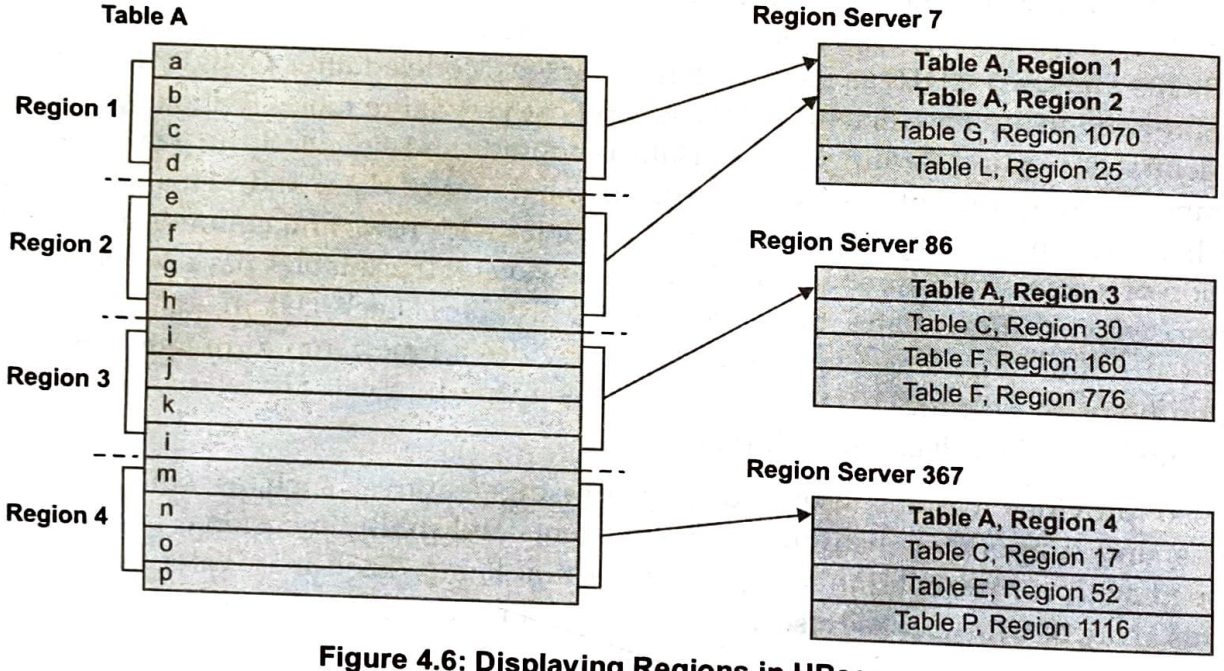


Figure 4.6: Displaying Regions in HBase

Regions are units that get spread over a cluster in HBase. Hence, a table too big for any single server can be carried by a cluster of servers with each node hosting a subset of all the regions of a table. This is also the medium by which the loading on a table gets spread. At a given time, the online group of sorted regions comprises the table's total content.

HBase persists data via the Hadoop file system API. There are multiple implementations of the file system interface—one each for the local file system, the KFS file system, Amazon's S3, and HDFS. HBase can persist to any of these implementations. By default, unless set otherwise, HBase writes into the local file system. The local file system is fine for experimenting with your initial HBase install, but thereafter, usually the first configuration made in an HBase cluster involves pointing HBase at the HDFS cluster to use.

Storing Big Data with HBase

HBase is a dispersed, non-relational (columnar) database that uses HDFS. It is designed according to Google Bigtable (a compressed, high performance proprietary data storage system built on the Google file system) and is equipped for facilitating extensive tables (billions of sections/columns) on

the grounds that it is layered on Hadoop clusters of appliance hardware. HBase gives arbitrary, continuous read/write access to enormous information. It is exceptionally configurable, giving a lot of flexibility to address immense measures of information proficiently.

In HBase, all information is put away into tables with columns and sections like relational frameworks (RDBMS). The cell is known as an intersection of a row and column. One essential distinction between HBase tables and RDBMS tables is versioning. Each cell value consists of a version property, which is just a timestamp uniquely distinguishing the cell. Versioning tracks changes in the cell and makes it possible to retrieve any version of the contents, if required. HBase stores the information in cells in descending order (utilizing the timestamp), so a read will obviously find the newer values first.

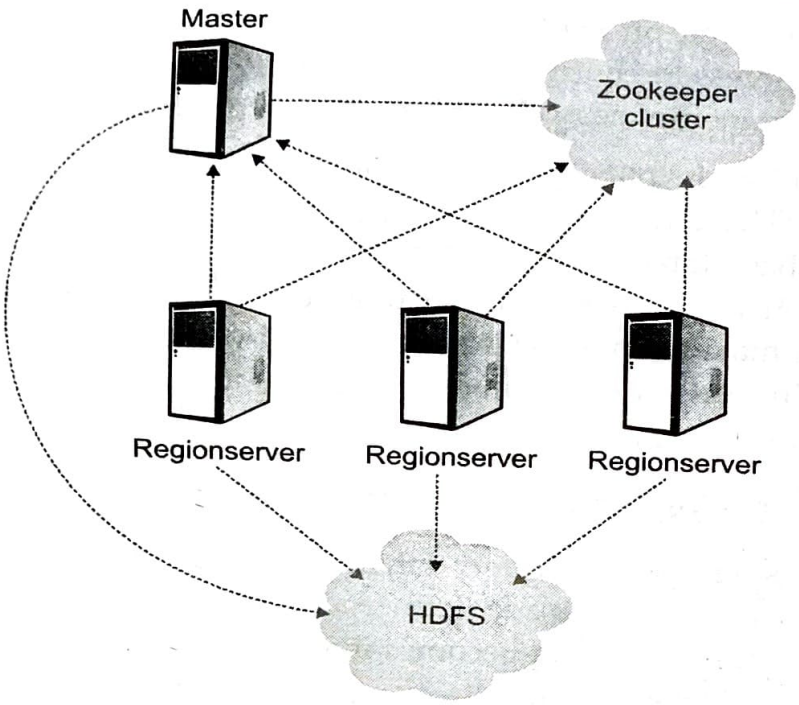
Interacting with the Hadoop Ecosystem

Writing programs or using specialty query languages is not the only way you interact with the Hadoop ecosystem. IT teams that manage infrastructures need to control Hadoop and the Big Data applications created for it. As Big Data becomes mainstream, non-technical professionals will want to try to solve business problems with it.

Hadoop-supported business distributions are always showing signs of change. New tools and technologies are presented, existing technologies are enhanced, and a few innovations are replaced by better substitutions. This is one of the key advantages of open source. An alternate is the selection of open-source innovations or applications by business organizations. These organizations upgrade the applications, making them better for everybody at an economical cost. This is how the Hadoop ecosystem has developed and why it is a good decision for serving to tackle your enormous information challenges.

HBase in Operation – Programming with HBase

HBase keeps individual catalog tables internally, called `-ROOT-` and `.META.`, within which it maintains the current list, state, recent history, and location of all regions at the top of the cluster. The `-ROOT-` table has the list of `.META.` table regions. The `.META.` table has a list of all the regions in it. Entries in these tables are keyed utilizing the region's starting row. Column keys, as noted earlier, are sorted so discovering the region that has a specific row is a matter of a lookup to discover the first entry whose key is more noteworthy than or equivalent to that of the asked for row key. Figure 4.7 displays an HBase cluster:



Source: http://www.programering.com/a/MzNwgjNwATY.html

Figure 4.7: Displaying an HBase Cluster

Fresh clients that connect to the ZooKeeper cluster need to first learn the location of the -ROOT- table. Clients consult -ROOT- to elicit the location of the .META. region, the scope of which covers the requested row. The client searches for the located .META. region to know the hosting region and its location. Thereafter, the client interacts directly with the hosting region-server.

To avoid making three round-trips per row operation, clients store all they learn while traversing -ROOT- and .META. caching locations as well as user-space region rows, so they can find hosting regions themselves without looking at the .META. table. As the clients work, they carry on using the cached entry, until there is a fault. When this happens, it means that the region has moved, and the client consults the .META. again to learn the new location. Therefore, if the contacted .META. region has moved, then -ROOT- is reconsulted.

Writes incoming at a region-server are first connected to a commit log file and then added to a cache. When the cache fills up, its content is moved to the file system. The log is stored on HDFS, so it stays available.

Combining HBase and HDFS

HBase's utilization of HDFS is different from how it is utilized by MapReduce. In MapReduce, by and large, HDFS documents are opened, their content streamed through a map task(which takes a set of data and converts it into another set of data), and are closed. In HBase, data files are opened on group startup and kept open so that the user does not have to pay the file open expenses on each access. Hence, HBase has a tendency to handle issues not commonly experienced by MapReduce clients, which are as follows:

- **File descriptors shortage**—Since we keep documents open on a loaded cluster, it doesn't take too long to run the documents on a framework. For example, we have a cluster having three hubs, each running an instance of a DataNode and region server, and we are performing an upload on

a table that presently has 100 regions and 10 column families. Suppose every column family has two file records. In that case, we will have 100 × 10 × 2 = 2000 records open at any given time. Add to this the collective random descriptors consumed by good scanners, and Java libraries. Each open record devours no less than one descriptor on the remote DataNode. The default quantity of record descriptors is 1024.

- **Not many DataNode threads**—Essentially, the Hadoop DataNode has a higher limit of 256 threads it can run at any given time. Suppose we use the same table measurements cited earlier. It is not difficult to perceive how we can surpass this figure given in the DataNode since each open connection with a record piece consumes a thread. If you look in the DataNode log, you will see an error like xceivercount 258 limit exceeds point of simultaneous xcievers 256; therefore, you need to be careful in using the threads.

  Bad blocks—The Dfsclient class in the region server will have a tendency to mark document blocks as bad if the server is heavily loaded. Blocks can be recreated only three times. Therefore, the region server will proceed onward for the recreation of the blocks. But if this recreation is performed during a period of heavy loading, we will have two of the three blocks marked as bad. In the event that the third block is discovered to be bad, we will see an error, stating, No live hubs contain current block in region server logs. During startup, you may face many issues as regions are opened and deployed. At the worst case, set the dfs.datanode.socket.write.timeout property to zero. Do note that this configuration needs to be set in a location accessible to HBase Dfsclient; set it in the hbase-site.xml file or by linking the hadoop-site.xml (or hdfssite.xml) file to your HBase conf directory.

- **UI**—HBase runs a Web server on the master to present a perspective on the condition of your running cluster. It listens on port 60010 by default. The master UI shows a list of basic functions, e.g., software renditions, cluster load, request rates, list of group tables, and participant region servers. In the master UI, click a region server and you will be directed to the Web server running the individual region server. A list of regions carried by this server and metrics like consumed resources and request rate would be displayed.

- **Schema design**—HBase tables are similar to RDBMS tables, except that HBase tables have versioned cells, sorted rows, and columns. The other thing to keep in mind is that an important attribute of the column `(-family)`-oriented database, like HBase, is that it can host extensive and lightly populated tables at no extra incurred cost.

- **Row keys**—Spend a good time in defining your row key. It can be utilized for grouping information as a part of routes. If your keys are integers, utilize a binary representation instead of a persistent string form of a number as it requires lesser space.

Now, let's create a table in Hbase, insert data in it, and then perform a cleanup.

To create a table, you must define its schema. The schema has table attributes and a table column family list. Column families have properties that you set at the time of schema definition. The schemas can be revised later by putting the table off line using the disable command (shell), making the changes using the alter command, and then restoring the table with the enable command.

You can create a table with the name `test` having a single column family `-data-` by using the following command:

```
hbase(main):007:0> create 'test', 'data' 0  row(s) in 4.3066 seconds
```

To view if the creation of the new table was successful, run the list command. This displays all the existing tables, as follows:

```
hbase(main):019:0> list
test                1 row(s) in 0.1112 seconds
```

Insert data into different rows, and columns at the `data` column family, and then list the content, by using the following commands:

```
hbase(main):021:0> put 'test', 'row1', 'data:1', 'value1'
0 row(s) in 0.0454 seconds
hbase(main):022:0> put 'test', 'row2', 'data:2', 'value2'
0 row(s) in 0.0035 seconds
hbase(main):023:0> put 'test', 'row3', 'data:3', 'value3'
0 row(s) in 0.0090 seconds hbase(main):024:0> scan 'test'
ROW                 COLUMN+CELL
row1                column=data:1, timestamp=1240148026198, value=value1
row2                column=data:2, timestamp=1240148040035, value=value2
row3                column=data:3, timestamp=1240148047497, value=value3
                    3 row(s) in 0.0825 seconds
```

To remove a table, you must first disable it, as follows:

```
hbase(main):025:0> disable 'test'
15/02/15 06:40:13 INFO client.HBaseAdmin: Disabled test
0 row(s) in 6.0426 seconds hbase(main):026:0> drop 'test'
15 /02/10 06:40:17 INFO client.HBaseAdmin: Deleted test
0 row(s) in 0.0210 seconds
hbase(main):027:0> list 0 row(s) in 2.0645 seconds
```

Shut down your HBase instance by running the following command:

```
% stop-hbase.sh
```

HBase is written in Java. Its classes and utilities are included in the `org.apache.hadoop.hbase.mapred` package, aided by using HBase in MapReduce jobs as a source. The `TableInputFormat` class splits the region borders, so maps are assigned a single region to work. The `TableOutputFormat` class writes the reduce result in HBase. The `RowCounter` class runs a map task using `TableInputFormat`, for counting rows. This class implements the Tool and HBase TableMap interfaces, as well as the `org.apache.hadoop.mapred.Mapper` interface, which sets the map inputs types passed by using the table input format. The `createSubmittableJob()` method analyses the added arguments command line configuration and estimates the table and columns we are to run using the `RowCounter` class. It also invokes the `TableMapReduceUtil.initTableMap Job()` utility method and sets the input format to `TableInputFormat`. MapReduce checks all the columns. In case some are found to be empty, it does not count the related rows; otherwise, it increments the `Counters.ROWS` property by one.

REST and Thrift

HBase contains the REST and Thrift interfaces, which are used when the interacting application is written in a language other than Java. In both cases, a Java server hosts the HBase instance, and the REST and thrift interfaces request for HBase storage. This extra work makes these interfaces slower than the Java client.

You can start REST by using the following command:

```
% hbase-daemon.sh start rest
```

This command starts a server instance, on the default port 60050, in the background, and catches any output by the server into a log file in the HBase *logs* directory.

Clients can ask for a response to be formatted as JSON or as XML, depending on how the client *HTTP Accept* header is set. You can stop the REST server by using the following command:

```
% hbase-daemon.sh stop rest
```

Similarly, you can start Thrift by running the following command:

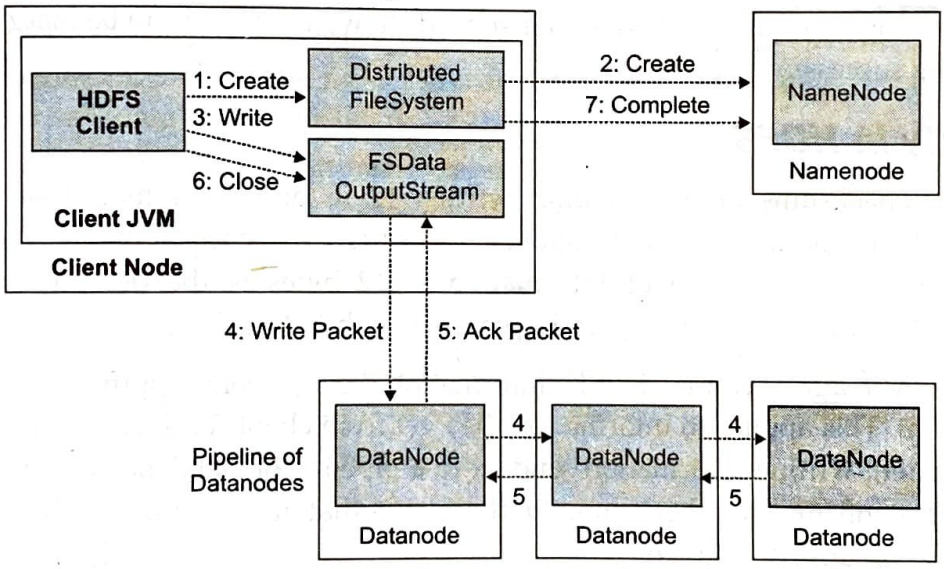
```
% hbase-daemon.sh start thrift
```

This will start the Thrift instance on the default port 9090, in the background, and catch any output by Thrift in a log file in the HBase logs directory.

You can stop Thrift by using the following command:

```
% hbase-daemon.sh stop thrift
```

Figure 4.8 shows the mechanism of writing data to HDFS:



Source: *http://hadoopguide.blogspot.in/2013/05/hadoop-hdfs-data-flow-io-classes.html*

Figure 4.8: Displaying the Process of Writing Data to HDFS

As shown in Figure 4.8, the client creates a record by calling the `make()` function on a distributed file system (step 1). The distributed file system makes a RPC call to the NameNode to make another record in the file system's namespace, with no pieces connected with it (step 2). The NameNode performs different checks to verify that the document doesn't currently exist, and that the client has the right contents to create the record. After passing these checks, the NameNode creates a record of the new document. Generally, the `Fsdataoutputstream` class encapsulates the `Dfsoutput` stream like in the read case, which manages the correspondence with the data as well as the NameNodes.

As the client composes information (step 3), `Dfsoutputstream` splits it into fragments, called the data queue. The information line is consumed by the `DataStreamer` class, whose obligation is to ask the NameNode to dispense new blocks by picking suitable DataNodes to store the copy. The DataNodes are structured in a pipeline, so there are three hubs in the pipeline. The `DataStreamer` class streams the packages in the first DataNode of the pipeline, where the packet is stored, and

advances it to the second DataNode in the pipeline. Also, the second DataNode stores the packet and advances it to the third (and last) DataNode in the pipeline (step 4). The `Dfsoutputstream` class additionally keeps an internal queue of packets that are waiting to be recognized by DataNodes, called the ack queue. A packet is expelled from the ack line just when it has been recognized by all the DataNodes in the pipeline (step 5).

In the event that a DataNode fails while information is being written to it, the pipeline is shut, and any packets in the ack line (step 5) are added to the front of the data line so that DataNodes that are downstream from the failure hub will not miss any packets. The present block of the good DataNodes is given a new identity, which is conveyed to the NameNode, so that the partial block on the failed DataNode is erased. The failed DataNode is expelled from the pipeline, and the rest of the block's information is written to the two good DataNodes in the pipeline.

At the point when the client has completed the process of writing information, on stream, it calls the `close ()` function (step 6). This activity pushes all the remaining packets to the DataNode pipeline and waits for affirmations before reaching the NameNode to signify that the work on the records is completed (step 7). The NameNode knows which blocks the document is made up of (through Data Streamer requesting block allocations), so it just needs to wait for blocks to be marginally reproduced before returning successfully.

Data Integrity in HDFS

HDFS directly checksums all information written to it, and as a matter of course, confirms checksums when perusing information. A different checksum is made for each `io.bytes.per.checksum` byte of information - 512 bytes is the default, and since CRC-32 checksum is of 4 bytes, the capacity operating cost is less than 1%.

DataNodes are in charge of confirming the information they get before putting away the information and its checksum. This applies to information they get from clients and from different DataNodes in replication. A client writing information sends it to a pipeline of DataNodes, and the last DataNode in the pipeline confirms the checksum. In the event that it recognizes a lapse, the client gets `Checksumexception`, a subclass of `Ioexception`.

When clients read information from DataNodes, they check checksums also, looking at them with the ones put away at the DataNode. Every DataNode keeps a determined log of checksum confirmations, so it knows when each of its blocks was last confirmed. At the point when a client effectively confirms a block, it tells the DataNode, which overhauls its log. This action is profitable in detecting bad disks.

Every DataNode runs a Datablockscanner in a background thread that occasionally checks all the blocks put away on the DataNode. This is to prepare for debasement because of "bit decay" in the physical storage media.

Since HDFS stores imitations of pieces, it can "repair" corrupted blocks by replicating one of the good copies to deliver another, uncorrupted copy. The way this works is that if a client identifies an error when examining a piece, the client reports the bad block in the related file system before throwing the `Checksumexception` exception. The NameNode marks the block copy as corrupt, so it doesn't forward clients to it. It then schedules the block to be duplicated on an alternate DataNode.

It is possible to disable checksum verifications by passing false to the setverify `Checksum()` method on the related file system, before utilizing the `open()` system to peruse a record. The same impact is possible from the shell by utilizing the `-ignorecrc` choice with the -get or equal `-copytolocal` charge. This feature is useful when you have a corrupt document that you need to assess so you can choose what to do with it.

NOTE

HBase stores data in column-oriented fashion instead of the record storage pattern as done in RDBMS.

Features of HBase

Some of the main features of HBase are:

- **Consistency**—In spite of not being an ACID implementation, HBase supports consistent read and write operations. This makes HBase suitable for high-speed requirements where RDBMS-supported extra features, such as full transaction support or typed columns, are not required.
- **Sharding**—HBase allows distribution of data using an underlying file system and supports transparent, and automatic splitting and redistribution of content.
- **High availability**—HBase implements region servers to ensure the recovery of LAN and WAN operations in case of a failure. The master server at the core monitors the regional servers and manages all the metadata for the cluster.
- **Client API**—HBase supports programmatic access using Java APIs.
- **Support for IT operations**—HBase provides a set of built-in Web pages to view detailed operational insights about the system.

NOTE

ACID stands for Atomicity, Consistency, Isolation, and Durability. It is a set of properties that ensures reliable processing of database transactions.

HBase implementations are most suitable in situations where:

- Gathering and processing of high volume, incremental data is required
- Real-time information processing and exchange take place
- Content changes at frequent intervals

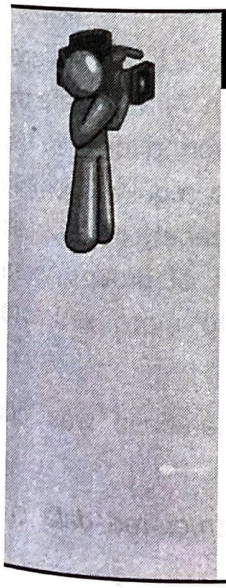


EXHIBIT 2: Nokia: Using Big Data to Bridge the Virtual & Physical Worlds

Company Overview

Nokia has been in business for more than 150 years, starting with the production of paper in the 1800s and evolving into a leader in mobile and location services that connects more than 1.3 billion people today. Nokia has always transformed resources into useful products — from rubber and paper, to electronics and mobile devices — and today's resource, which is data.

Nokia's goal is to bring the world to the third phase of mobility: leveraging digital data to make it easier to navigate the physical world. To achieve this goal, Nokia needed to find a technology solution that would support the collection, storage, and analysis of virtually unlimited data types and volumes.

Use Case

Effective collection and use of data have become central to Nokia's ability to understand and improve users' experiences with their phones and other location products. "Nokia differentiates itself based on the data we have," stated Amy O'Connor, Senior Director of Analytics at Nokia. The company leverages data processing and complex analyses in order to build maps with predictive traffic and layered elevation models, to source information about points of interest around the world, to understand the quality of phones, and more.

To grow and support its extensive use of Big Data, Nokia relies on a technology ecosystem that includes a Teradata Enterprise Data Warehouse (EDW), numerous Oracle and MySQL data marts, visualization technologies, and at its core: Hadoop. Nokia has over 100 terabytes (TB) of structured data on Teradata and petabytes (PB) of multi-structured data on the Hadoop Distributed File System (HDFS), running on Dell PowerEdge servers. The centralized Hadoop cluster that lies at the heart of Nokia's infrastructure contains .5 PB of data. Nokia's data warehouses and marts continuously stream multi-structured data into a multi-tenant Hadoop environment, allowing the company's 60,000+ employees to access the data. Nokia runs hundreds of thousands of Scribe processes each day to efficiently move data from, for example, servers in Singapore to a Hadoop cluster in the UK data center. The company uses Sqoop to move data from HDFS to Oracle and/or Teradata. And Nokia serves data out of Hadoop through HBase.

Business Challenges before Hadoop

Prior to deploying Hadoop, numerous groups within Nokia were building application silos to accommodate their individual needs. It didn't take long before the company realized it could derive greater value from its collective data sets if these application silos could be integrated, enabling all globally captured data to be cross-referenced for a single, comprehensive version of truth.

"We were inventorying all of our applications and data sets," O'Connor noted. "Our goal was to end up with a single data asset."

Nokia wanted to understand at a holistic level how people interact with different applications around the world, which required them to implement an infrastructure that could support daily, terabyte-scale streams of unstructured data from phones in use, services, log files, and other sources. Leveraging this data also requires complex processing and computation to be consumable and useful for a variety of uses, like gleaning market insights, or understanding collective behaviors of groups; some aggregations of that data also need to be easily migrated to more structured environments in order to leverage specific analytic tools.

However, capturing petabyte-scale data using a relational database was cost prohibitive and would limit the types of data that could be ingested.

"We knew we'd break the bank trying to capture all this unstructured data in a structured environment," O'Connor said.

Because Hadoop uses industry standard hardware, the cost per terabyte of storage is, on average, 10 times cheaper than a traditional relational data warehouse system. Additionally, unstructured data must be reformatted to fit into a relational schema before it can be loaded into the system. This requires an extra data processing step that slows ingestion, creates latency and eliminates elements of the data that could become important down the road.

Various groups of engineers at Nokia had already begun experimenting with Apache Hadoop, and a few were using Cloudera's Distribution Including Apache Hadoop (CDH). The benefits of Hadoop were clear — it offers reliable, cost-effective data storage and high performance parallel processing of multi-structured data at petabyte scale — however, the rapidly evolving platform and tools designed to support and enable it are complex and can be difficult to deploy in production. CDH simplifies this process, bundling the most popular open source projects in the Apache Hadoop stack into a single, integrated package with steady and reliable releases.

After experimenting with CDH for several months, the company decided to standardize the use of the Hadoop platform to be the cornerstone of its technology ecosystem. With limited Hadoop expertise in-house, Nokia turned to Cloudera to augment their internal engineering team with strategic technical support and global training services, giving them the confidence with expertise necessary to deploy a very large production Hadoop environment in a short timeframe.

Impact

In 2011, Nokia put its central CDH cluster into production to serve as the company's enterprise-wide information core. Cloudera supported the deployment from start to finish, ensuring the cluster was successfully integrated with other Hadoop clusters and relational technologies for maximum reliability and performance.

Nokia is now using Hadoop to push the analytics, creating 3D digital maps that incorporate traffic models that understand speed categories, recent speeds on roads, historical traffic models, elevation, ongoing events, video streams of the world, and more.

"Hadoop is absolutely mission critical for Nokia. It would have been extremely difficult for us to build traffic models or any of our mapping content without the scalability and flexibility that Hadoop offers," O'Connor explained. "We can now understand how people interact with the apps on their phones to view usage patterns across applications. We can ask things like, 'Which feature did they go to after this one?' and 'Where did they seem to get lost?' This has all been enabled by Hadoop, and we wouldn't have gotten our Big Data platform to where it is today without Cloudera's platform, expertise and support."

Source: http://hadoopilluminated.com/hadoop_illuminated/cached_reports/Cloudera_Nokia_Case_Study_Hadoop.pdf

Hive

Hive is a data-warehousing layer created with the core elements of Hadoop (HDFS and MapReduce) to support batch-oriented processes. It exposes a simple SQL-like implementation called HiveQL for easy interaction along with access via mappers and reducers. In this manner, Hive provides the best of both worlds with an SQL-like access to structured data and sophisticated analysis of Big Data using MapReduce. Hive looks somewhat similar to traditional database. Hive uses map-reduce operations over Hadoop ecosystem, and there are many differences between Hive and traditional SQL. Hive has not been designed for quickly addressing queries, like other data warehouses. In fact, query processing and execution in Hive may take several hours depending on the complexity involved. On the other hand, Hive is used effectively for data mining and intense analysis, which does not involve real-time processing. Since Hive works on underlying Hadoop functions, it is scalable, resilient, and extensible.

Pig and Pig Latin

The official website of Hadoop, *http://hadoop.apache.org/*, defines Pig as "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. 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."

Pig is used as an ELT tool for Hadoop. It makes Hadoop more approachable and usable for non-technical persons. It opens an interactive and script-based execution environment for non-developers with its language, Pig Latin. Pig Latin loads and processes input data using a series of operations and transforms that data to produce the desired output.

Sqoop

Sqoop (SQL-to-Hadoop) is a tool used for data transfer between Hadoop and relational databases. Critical processes are employed by MapReduce to move data into Hadoop and back to other data sources.

Similar to Pig, Sqoop is also a command-line interpreter, which sequentially executes Sqoop commands. Sqoop can be effectively used by non-programmers as well and relies on underlying technologies like HDFS and MapReduce.

ZooKeeper

Hadoop works by the divide-and-conquer approach. Once a problem is divided, it is approached and processed by using distributed and parallel processing techniques across Hadoop clusters. In case of Big Data problems, traditional interactive tools do not provide enough insight or timelines required to take business decisions. In that case, Big Data problems are approached with distributed applications. ZooKeeper helps in coordinating all the elements of the distributed applications.

Flume

Apache Flume aids in transferring large amounts of data from distributed resources to a single centralized repository. It is robust and fault tolerant, and efficiently collects, assembles, and transfers data. Flume is used for real-time data capturing in Hadoop. It can be applied to assemble a wide variety of data such as network traffic, data generated via social networking, business transaction data, and emails. The simple and extensible data model of Flume facilitates fast online data analytics.

As an implied tool, Flume covers a wide range of common requirements for transferring data into HDFS. It is a very popular system for moving large amounts of streaming data into HDFS. A common case where Flume can be applied is of collecting the log data from different systems, and collecting the aggregated data into HDFS for later analysis. Flume is also used for collecting data from various social media websites such as Twitter and Facebook.

Oozie

Oozie is an open-source Apache Hadoop service used to manage and process submitted jobs. It supports the workflow/coordination model and is highly extensible and scalable. Oozie is a dataware service that coordinates dependencies among different jobs executing on different platforms of Hadoop, such as HDFS, Pig, and MapReduce.

Hive, Pig and Pig Latin, Sqoop, ZooKeeper, Flume, and Oozie have been discussed in detail in further chapters.

Summary

In this chapter, you have learned about the Hadoop ecosystem and explored its major components. We have discussed various aspects of data storage in Hadoop. In the beginning, it has introduced HDFS and its architecture. After that, it has described HDFS files and Hadoop-specific file types. Next, the chapter has explained HBase and its architecture. The chapter has also shown the procedure of installing HDFS in a system. In addition, you have learned how to combine HBase with HDFS. We have learned the other components like MapReduce, YARN, HBase, and Hive. The chapter has also discussed various tools used to interact with the Hadoop ecosystem including, Pig and Pig Latin, Sqoop, ZooKeeper, Flume, and Oozie.

Quick Revise

Multiple-Choice Questions

Q1. **Which of the following terms is used to denote the small subsets of a large file created by HDFS?**

a. NameNode
b. DataNode
c. Blocks
d. Namespace

Ans. The correct option is c.

Q2. What message is generated by a DataNode to indicate its connectivity with NameNode?

a. Beep
b. Heartbeat
c. Analog pulse
d. Map

Ans. The correct option is b.

Q3. Which of the following defines metadata?

a. Data about data
b. Data from Web logs
c. Data from government sources
d. Data from market surveys

Ans. The correct option is a.

Q4. Which of the following is managed by the MapReduce environment?

a. Web logs
b. Images
c. Structured data
d. Unstructured data

Ans. The correct option is d.

Q5. Which of the following services is provided by YARN?

a. Global resource management
b. Record reader
c. MapReduce engine
d. Data mining

Ans. The correct option is a.

Q6. In an HDFS cluster, _____ manages cluster metadata.

a. NameNode
b. DataNode
c. Inode
d. Namespace

Ans. The correct option is a.

Q7. Which of the following commands of HDFS can issue directives to blocks?

a. fcsk
b. fkcs
c. fsck
d. fkcs

Ans. The correct option is c.

Q8. Which of the following file systems provides read-only access to HDFS over HTTPS?

a. HAR
b. HDFS
c. HFTP
d. HSFTP

Ans. The correct option is d.

Subjective Questions

Q1. Write a short note on the Hadoop ecosystem.

Ans. The Hadoop ecosystem is a framework of various types of complex and evolving tools and components. Some of these elements may be very different from each other in terms of their architecture; however, what keeps them all together under a single roof is that they all derive their functionalities from the scalability and power of Hadoop. In simple words, the Hadoop

ecosystem can be defined as a comprehensive collection of tools and technologies that can be effectively implemented and deployed to provide Big Data solutions in a cost-effective manner.

MapReduce and **Hadoop distributed File System (HDFS)** are two core components of the Hadoop ecosystem that provide a great starting point to manage Big Data; however, they are not sufficient to deal with the Big Data challenges. Along with these two, the Hadoop ecosystem provides a collection of various elements to support the complete development and deployment of Big Data solutions.

Q2. What is metadata? What information does it provide?

Ans. Metadata refers to data about data. It acts as a template that provides the following information:

- The time of creation, last access, modification, and deletion of a file
- The storage location of the blocks of files on a cluster
- The access permissions to view and modify a file
- The number of files stored on a cluster
- The number of DataNodes connected to form a cluster
- The location of the transaction log on a cluster

Q3. How does HDFS ensure data integrity in a Hadoop cluster?

Ans. HDFS ensures data integrity throughout the cluster with the help of the following features:

- **Maintaining Transaction Logs**—HDFS maintains transaction logs in order to monitor every operation and carry out effective auditing and recovery of data in case something goes wrong.
- **Validating Checksum**—Checksum is an effective error-detection technique wherein a numerical value is assigned to a transmitted message on the basis of the number of bits contained in the message. HDFS uses checksum validations for verification of the content of a file. The validations are carried out as follows:
  1. When a file is requested by a client, the contents are verified using checksum.
  2. If the checksums of the received and sent messages match, the file operations proceed further; otherwise, an error is reported.
  3. The message receiver verifies the checksum of the message to ensure that it is the same as in the sent message. If a difference is identified in the two values, the message is discarded assuming that it has been tempered in transition. Checksum files are hidden to avoid tempering.
- **Creating Data Blocks**—HDFS maintains replicated copies of data blocks to avoid corruption of a file due to failure of a server. The degree of replication, the number of DataNodes in the cluster, and the specifications of the HDFS namespace are identified and implemented during the initial implementation of the cluster. However, these parameters can be adjusted any time during the operation of the cluster.

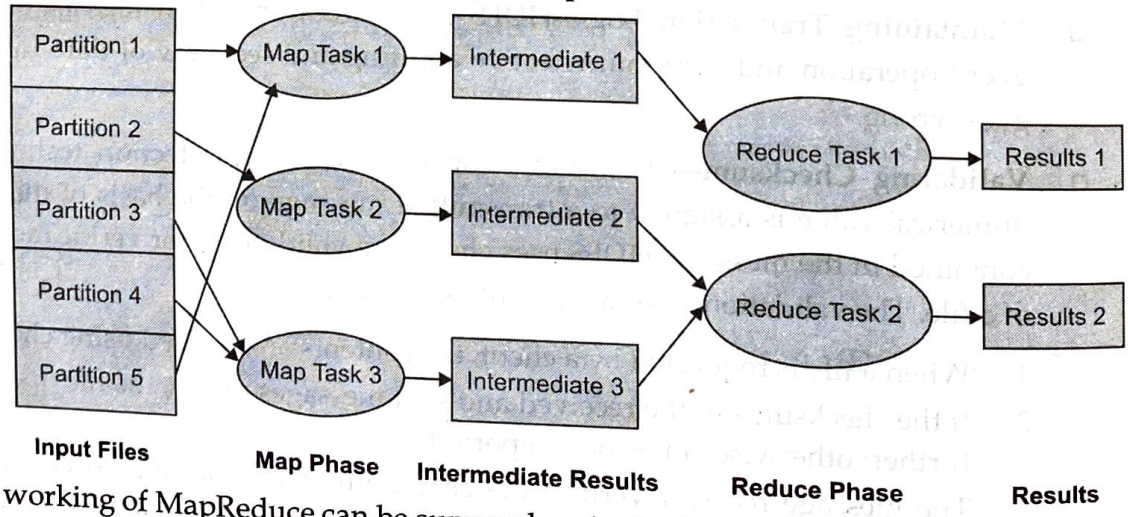
Data blocks are sometimes, also called block servers. A block server primarily stores data in a file system and maintains metadata of a block. A block server carries out the following functions:

- Storage (and retrieval) of data on a local file system. HDFS supports different operating systems and provides similar performance on all of them.
- Storage of metadata of a block on the local file system on the basis of similar template on the NameNode.
- Conduction of periodic validations for file checksums.
- Intimation about the availability of blocks to the NameNode by sending reports regularly.
- On-demand supply of metadata and data to the clients where client application programs can directly access DataNodes.
- Movement of data to connected nodes on the basis of the pipelining model.

Q4. Discuss the working of MapReduce.

Ans. MapReduce enables computational processing of data stored in a file system without the requirement of loading the data initially into a database. It primarily supports two operations: map and reduce. These operations execute in parallel on a set of worker nodes. MapReduce works on a master/worker approach in which the master process controls and directs the entire activity, such as collecting, segregating, and delegating the data among different workers.

The following figure depicts the working of MapReduce:



The working of MapReduce can be summed up in the following steps:

1. A MapReduce worker receives data from the master, processes it, and sends back the generated result to the master.
2. MapReduce workers run the same code on the received data; however, they are not aware about other co-workers and do not communicate or interact with each other.
3. The master receives the results from each worker process, integrates the results received, processes the integrated result, and generates the final output.

Q5. **Discuss the following terms:**

- **Streaming information access**
- **Low-latency information access**

Ans. **Streaming information access**—HDFS is created for batch processing. The priority is given to high throughput to data access rather than the low latency of data access. A dataset is commonly produced or replicated from the source, and then various analyses are performed on that dataset in the long run. Every analysis is done in detail and hence time consuming, so the time required for examining the entire dataset is quite high.

Low-latency information access—Applications that permit access to information in milliseconds do not function well with HDFS. Therefore, HDFS is upgraded for conveying a high transaction volume of information, and this may be at the expense of idleness. HBase is at present a superior choice for low latency access.

Q6. **What is the role of NameNode in an HDFS cluster?**

Ans. HDFS cluster has two node types working in a slave master design: a NameNode (the master) and various DataNodes (slaves). The NameNode deals with the file system. It stores the metadata for all the documents and indexes in the file system. This metadata is stored on the local disk as two files: the file system image and the edit log. The NameNode is aware of the DataNodes on which all the pieces of a given document are found; however, it doesn't store block locations necessarily, since this data is recreated from DataNodes.

A client accesses the file system on behalf of the user by communicating with the DataNodes and the NameNode. The client provides a file system, like the POSIX interface, so the user code does not require the NameNode and DataNodes in order to execute.

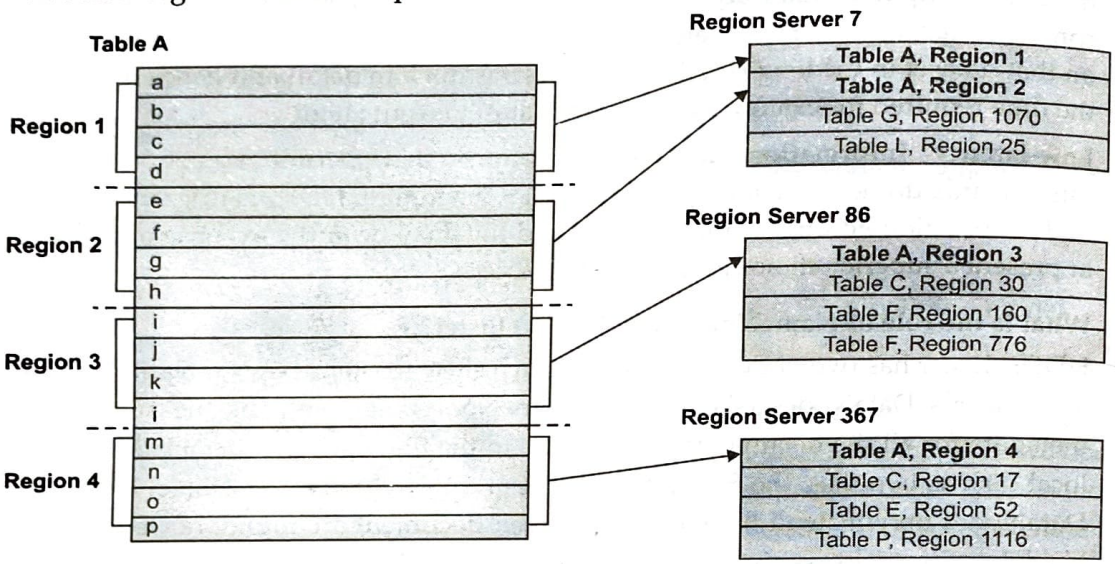
Q7. **What is HBase?**

Ans. HBase is a column-oriented distributed database composed on top of HDFS. HBase is the Hadoop application used when you need real-time continuous read/write access to huge datasets. Consider a case of Argon Technology, which provides Big Data solutions. A telecom client of Argon Technology has been facing a problem in handling a database in real time. Moreover, the data is getting bigger day by day. The client approaches Argon Technology to find a solution to the problem. The technical team at Argon Technology suggests that the client shift its database to HBase, which is proficient in handling large databases and allows fast real-time processing of data.

The standard HBase is considered as a Web table, a table of Web pages crawled (or accessed) and their properties (such as language and MIME type) keyed by the Web page URL. The Web table is large, containing over a billion rows. Parsing and batch analytics are MapReduce jobs that continuously run against the Web table. These jobs derive statistics and add new columns of the MIME type and parsed text content for later indexing by a search engine. Simultaneously, the Web table is accessed randomly by crawlers (search engines) running at various rates and updating random rows. The random pages in the Web table are served in real-time as users click on a website's cached-page feature.

Q8. Discuss the concept of regions in HBase.

Ans. Tables are automatically partitioned horizontally into regions by HBase. Each region consists of a subset of rows of a table. Initially, a table comprises a single region but as the size of the region grows, after it crosses a configurable size limit, it splits at the boundary of a row into two new regions of almost equal size, as shown in the following figure:



Regions are units that get spread over a cluster in HBase. Hence, a table too big for any single server can be carried by a cluster of servers with each node hosting a subset of the total regions of a table. This is also the medium by which the loading on a table gets spread. At a given time, the online group of sorted regions comprises the table's total content.