Big Data Using Hadoop™ and Hive™

N. Kumar

Big Data Using Hadoop™ and Hive™

LICENSE, DISCLAIMER OF LIABILITY, AND LIMITED WARRANTY

By purchasing or using this book (the "Work"), you agree that this license grants permission to use the contents contained herein, but does not give you the right of ownership to any of the textual content in the book or ownership to any of the information or products contained in it. *This license does not permit uploading of the Work onto the Internet or on a network (of any kind) without the written consent of the Publisher.* Duplication or dissemination of any text, code, simulations, images, etc. contained herein is limited to and subject to licensing terms for the respective products, and permission must be obtained from the Publisher or the owner of the content, etc., in order to reproduce or network any portion of the textual material (in any media) that is contained in the Work.

Mercury Learning and Information ("MLI" or "the Publisher") and anyone involved in the creation, writing, production, accompanying algorithms, code, or computer programs ("the software"), and any accompanying Web site or software of the Work, cannot and do not warrant the performance or results that might be obtained by using the contents of the Work. The author, developers, and the Publisher have used their best efforts to ensure the accuracy and functionality of the textual material and/or programs contained in this package; we, however, make no warranty of any kind, express or implied, regarding the performance of these contents or programs. The Work is sold "as is" without warranty (except for defective materials used in manufacturing the book or due to faulty workmanship).

The author, developers, and the publisher of any accompanying content, and anyone involved in the composition, production, and manufacturing of this work will not be liable for damages of any kind arising out of the use of (or the inability to use) the algorithms, source code, computer programs, or textual material contained in this publication. This includes, but is not limited to, loss of revenue or profit, or other incidental, physical, or consequential damages arising out of the use of this Work.

The sole remedy in the event of a claim of any kind is expressly limited to replacement of the book and only at the discretion of the Publisher. The use of "implied warranty" and certain "exclusions" vary from state to state, and might not apply to the purchaser of this product.

Big Data Using Hadoop™ and Hive™

Nitin Kumar

Mercury Learning and Information
Dulles, Virginia
Boston, Massachusetts
New Delhi

Publisher: David Pallai
Mercury Learning and Information
22841 Quicksilver Drive
Dulles, VA 20166
info@merclearning.com
www.merclearning.com
800-232-0223

Nitin Kumar. *Big Data Using Hadoop™ and Hive™*.
ISBN: 978-1-68392-645-0

Library of Congress Control Number: 2021934303

212223321 This book is printed on acid-free paper in the United States of America.

To my wife Sarika, my children, Shaurya and Irima,
and to my parents

CONTENTS

PREFACE

Big Data Using Hadoop and Hive is the essential guide for developers, architects, engineers, and anyone who wants to start leveraging Hadoop to build a distributed, scalable, and concurrent applications. It is a concise guide to get started with Hadoop and Hive. It provides an overall understanding of Hadoop and how it works while giving the sample code to speed up development with minimal effort. It refers to simple concepts and examples, as they are likely to be the best teaching aids. It will explain the logic, code, and configurations needed to build a successful, distributed, concurrent application, and the reason behind those decisions.

Intended Audience

The primary audience of this book is the developers who would like to start development using Hadoop and Hive, architects who want to get an architect's understanding of Hadoop technologies, and application designers looking for a detailed understanding of what Hadoop can do for them.

What this book covers

Big Data Using Hadoop and Hive covers Hadoop 3 exclusively. The Hadoop 3.3.x release series is the current active release series and contains the most stable version of Hadoop. This edition also covers the latest Hive 3.x version with all-new features. It contains four main parts: Chapters 1 through 5 are an introduction to Hadoop and the architecture around it. Chapters 6 to 8 discuss architecture and HDFS, MapReduce, YARN, and Mesosphere. In Chapters 9 and 10, we cover Hive and its architecture. Chapter 11 discusses various file formats and the compression methodology used in the Hadoop environment.

Chapter 1: This chapter explains Big Data and how organizations face difficulties with it. This chapter explores the key benefits of Big Data and how Big Data is utilized. It also shines light on how Hadoop solves the challenges of Big Data.

Chapter 2: This chapter covers Hadoop history and answers questions regarding the key benefits of Apache Hadoop. It also provides a brief introduction to the Hadoop ecosystem and explains Hadoop architecture.

Chapter 3: This chapter covers the Hadoop Distributed File System (HDFS) and its architecture. It also includes the HDFS core components and how Hadoop reads/writes with the HDFS.

Chapter 4: This chapter explains the basic Hadoop set-up and essential Hadoop configuration and optimization. It also covers the command-line interface to access the HDFS.

Chapter 5: This chapter covers the JAVA API and WebHDFS REST API to access HDFS files.

Chapter 6: This chapter provides a detailed understanding of MapReduce jobs and how the Hadoop Core framework executes MapReduce jobs.

Chapter 7: This chapter explains the challenges of Hadoop v1 and how YARN solves these challenges. This chapter explains YARN's basic architecture understanding and various failover scenarios on YARN. It also describes the Hadoop Timeline Server introduced in Hadoop v3.

Chapter 8: This chapter covers Hive and its architecture with various storage types. It also includes the Hive query language.

Chapter 9: This chapter covers the step-by-step Hive set-up. It includes different types of tables and explains how to query the table data. It also emphasizes various join and builds in utility functions.

Chapters 10 and 11: These chapters explain various file formats, such as Avro, ORC, and Parquet. They also cover different compression mechanisms.

Nitin Kumar
March 2021

CHAPTER 1

BIG DATA

There has been an explosive growth of social interactions, including online relationships, collaboration, information sharing, and feedback. Social networks such as Twitter, Facebook, LinkedIn, YouTube, and WhatsApp are connecting millions of users, allowing them to share vast amounts of information. Consumers invest more time on the Internet, which enables organizations to collect a massive amount of data. Every new generation of devices is connected and sharing massive amounts of sensor data. These data could be anything, such as user transaction logs, sales logs, user history, machine-generated data, video streaming, and sensor data.

These large amounts (terabytes, petabytes, and exabytes) of structured and unstructured data are known as *Big Data*. Today, businesses have endless opportunities to transform these structured and unstructured data into practical information for decision-making, predictive analysis, recommendation systems, and analytic platforms.

Organizations collect and transform data into actionable insights that can be used to contribute to revenue generation. There is an enormous demand for professionals in these areas to build models, extract features and patterns, and perform statistical analysis to transform the data into meaningful information.

Various fields were spawned from Big Data, such as artificial intelligence (AI), machine learning, and deep learning, which apply patterns to predict outcomes and make recommendations. For example, video streaming services keep your interest based on your previous activity and recommend the same type of video for future viewing. Online shopping sites provide recommendations based on your interests, trends, and events.

BIG DATA CHALLENGES FOR ORGANIZATIONS

Internet users are doing everything online, from business communications to shopping and social networking. Billions of connected devices and embedded systems create, collect, and share a wealth of data analytics every day, all over the world. According to IDC, 75% of the world's population will be interacting with online data every day by 2025. The amount of digital data is expected to more than double.

Organizations are collecting a large amount of data, but struggling to manage and utilize it. This leads to a new challenge regarding how to effectively manage, process, and analyze vast amounts of data. The challenge is not only performing data extraction but also managing that expanding amount of data.

Traditional RDBMS-based databases were not capable of handling this massive amount of unstructured data, and so organizations were not able to use it. If data is archived on tapes, it is expensive to recover for further use. There was a need for technologies that store and process massive amounts of data.

HOW WE ARE USING BIG DATA

Big Data isn't only about the quantity of information, but how we utilize it to improve statistical and computational process. Organizations are adopting new technologies to visualize new patterns that could improve their strategic decision-making. Big Data helps organizations drive innovation by providing new insights about their customers. An organization can build an understanding of a customer based on that customer's transaction history and provide improved recommendations to them about products or services. Behavioral analytics is important, which visualizes business sentiment and accelerates business outcomes.

BIG DATA: AN OPPORTUNITY

Big Data has brought about a new era in the economy, which is all about information and converting it into revenue. Organizations need to understand how to deal with all this data. Their growth depends on the quality of their

predictive mechanisms and how they leverage machine learning. In one survey, 97% of companies responded that they plan to increase their spending in analytic endeavors, with estimates that companies will drive $432 billion in IT spending through 2025. By 2023, the demand for jobs related to Big Data are estimated to soar 38%.

HADOOP: A BIG DATA SOLUTION

Astronomical unstructured data is a challenge to store, process, and extract information. Traditional RDBMS systems are not compatible with handling such large data sets.

Apache Hadoop is the solution for such types of data sets. Hadoop is an open-source project started by Doug Cutting. It was initiated based on papers published by Google, describing how their engineers were dealing with the challenge of storing and processing massive amounts of data. Yahoo! and other IT companies have driven the development of Hadoop.

BIG DATA IN THE REAL WORLD

Hadoop provides highly scalable, reliable, and distributed data processing computing platforms. Below are some of the critical business use cases for Big Data.

- **Large Data Transformation:** In big organizations, we always face challenges regarding how to process extensive data from one source to another source.
- **Market Trends:** Organizations need to analyze market trends based on the feedback they received. Big Data provides the best-optimized market trends.
- **Machine Learning:** Data can be obtained from different sources. Developers can build a system that can learn from data, such as Artificial Intelligence (AI), speech recognition, or understanding human behaviors.
- **Making Recommendations:** Organizations utilize user-buying patterns to identify recommendations for proactive sales.
- **Decision-making:** The organization uses data to predict the future and make decisions.

- **Campaigning:** Big Data processing provides a way to customize the marketing campaigns based on user activity and behavior.
- **User Behavior:** Organizations are interested in how their customers use their product, and Big Data provides a window to this behavior.
- **Predictive Mechanism:** Organizations get extensive data and ways to extrapolate and make predictions using it.

CHAPTER 2

WHAT IS APACHE HADOOP?

Hadoop provides the capabilities to store a massive amount of data in a distributed environment and process it effectively. It's a distributed data processing system that supports distributed file systems, and it offers a way to parallelize and execute programs on a cluster of machines. It could be installed on a cluster using a large number of commodities hardware, which optimizes overall solution costs.

Apache Hadoop has been adopted by technology giants such as Yahoo, Facebook, Twitter, and LinkedIn to address their data needs, and it's making inroads across all industrial sectors.

The Apache Hadoop software library is a distributed framework for processing massive data sets in batches and streams across clusters. It allows for scaling servers to thousands of machines to support large data set computation with a robust failure handling mechanism. Each device on a Hadoop cluster offers local computation and storage, as well as failure detection and handling.

HADOOP HISTORY

Doug Cutting originally developed Nutch, an open-source Web search engine, which is part of Lucene. However, it is a challenge to scale up this solution in a distributed environment. The Nutch Distributed Filesystem (NDF) was based on a paper published in 2003 [*http://research.google.com/archive/gfs.html*].

In 2005, MapReduce with NDFS was implemented in Nutch. Later, Doug Cutting joined Yahoo!, which provided a dedicated team to build Hadoop on top of Lucene based on Nutch's NDFS. Yahoo! later announced that a 10,000

core Hadoop cluster generated its production search index. In 2008, Yahoo released Hadoop as an Apache open source project (Apache Software Foundation). In 2009, Hadoop successfully evaluated a petabyte of data in less than 17 hours and handled billions of searches. In 2011, Doug Cutting joined Cloudera and spread Hadoop to other organizations. In the same year, the Apache foundation released Apache Hadoop version 1.0; Hadoop version 2 was released in 2013, and Apache Hadoop version 3.0 was released in 2017.

HADOOP BENEFITS

Hadoop is one of the best solutions for the distributed storage and processing of a vast dataset (with terabytes or petabytes of information) in a cluster environment. It is fault-tolerant, scalable, and easy to use.

It divides files into small parts and distributes them into multiple parallel processing nodes to accelerate the processing time. Hadoop leverages clusters of machines to provide ample storage and processing power at a price that businesses can afford. Hadoop provides a scalable and reliable mechanism for processing large amounts of data over cluster environments. It offers novel analytic techniques that enable the sophisticated processing of multi-structured data. Data replicate in multiple nodes for reliability and higher availability. Hadoop processes data in the local node before aggregating it the remote node, which minimizes bandwidth bottlenecks.

The following are some of the salient features of Hadoop:

- **Massive data processing:** Apache Hadoop can process massive amounts of data.
- **Cost:** Apache Hadoop is a low-cost solution for high-performance computing applications.
- **Easily scalable:** Apache Hadoop distributes data into small chunks across clusters that can run independently. It provides flexibility to scale storage without impacting the application.
- **Distributed processing:** Hadoop breaks large datasets into smaller, fixed chunks and distributes those chunks across the clusters. It allows users to access and process data in a distributed environment.
- **Reliability:** Reliability is a big challenge in a distributed cluster environment. Hadoop was designed so it can detect failures and retries by replicated processing to other nodes.

- **Separation of concerns:** Hadoop maintains its business functionality processing within the overall infrastructure, and offers fault tolerance, high availability, parallel processing, and storage.
- **Fast processing:** It breaks the data processing task into multiple smaller jobs, which run in parallel in distributed cluster environments to reduce the overall data processing time.
- **Fault tolerance & high availability:** HDFS replicates data across the clusters, which gives it the power to handle failure and protect data loss in case of a node crash. In overall processing, if any node failed, HDFS efficiently allocates data from replicas. Since each task is independent, MapReduce easily re-processes the failed task.
- **Highly configurable:** Apache Hadoop is highly configurable and provides a default configuration.
- **Status monitoring:** Apache Hadoop builds on Web servers to make it easy to monitor the clusters and jobs through a Web application.

HADOOP'S ECOSYSTEM: COMPONENTS

Various components have been developed for Apache Hadoop to assist with Big Data solutions, and these form the Hadoop ecosystem.

The Hadoop Distributed File System (HDFS) is the core component for storing and accessing large file systems. MapReduce is the heart of Hadoop, and provides the capability to process files stored in the HDFS. HBase assists with real-time data processing. Hive allows for a data query without using the MapReduce program. New components are always being added to the Apache Hadoop ecosystem.

We can visualize the Hadoop Ecosystem as a Hadoop platform, a combination of various components stitched together to provide a Big Data solution.

- **HDFS:** The Hadoop Distributed File System partitions data and stores it across cluster nodes. HDFS is used to store a massive amount of data over a distributed environment. HDFS stores metadata information from the file and data separately. Data stored in the HDFS is written once, but read many times. It provides a base for other tools, such as Hive, Pig, HBase, and MapReduce, to process data.
- **YARN:** It was introduced in Hadoop 2 and is available in higher versions. It decouples the functionalities of resource management and job scheduling/monitoring into separate daemons.

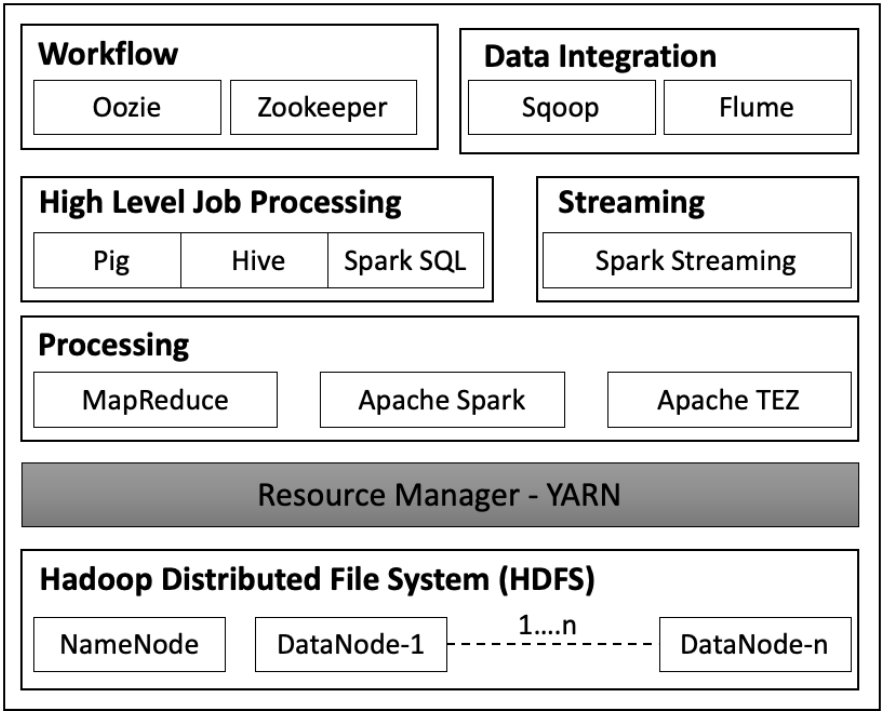

FIGURE 2.1 Key Hadoop ecosystem components and the layered architecture

- **MapReduce:** MapReduce is Hadoop's key component for processing a massive amount of data in parallel. It provides mechanisms to handle large datasets as a batch in highly reliable, available, and fault-tolerant environments. MapReduce breaks the data into independent parts, which are processed in parallel by the map task and pass the <Key, value> to a reducer that aggregates them before storing them into the HDFS.
- **Apache Spark:** Apache Spark provides fast in-memory data processing for the Hadoop environment, as well as support for a wide range of processing, including ETL, machine learning, stream processing, and graph computation.
- **Apache Tez:** Apache Tez is an alternative of MapReduce in Hadoop 2 used to process HDFS data in both batch and interactive ways based on the Database Availability Group (DAG).
- **Zookeeper:** Zookeeper is Apache Hadoop's coordination service designed to manage Hadoop operations.

- **Oozie:** Oozie is a workflow system for MapReduce designed to manage multiple MapReduce job flows.
- **Pig:** Pig provides a scripting language (*Pig Latin*) to analyze datasets. Pig Latin makes it easy to create a sequence of MapReduce programs.
- **Hive:** Apache Hive provides a SQL-like language to retrieve data stored in Hadoop. Developers can write a SQL-like query, which gets translated into MapReduce jobs in Hadoop. Hive is more useful to developers who are familiar with SQL.
- **Sqoop:** Sqoop is an integration framework used to transfer data from the relational database to Hadoop and vice versa. Sqoop uses the database to describe the schema and MapReduce for parallelization operation and fault tolerance.
- **Flume:** Apache Flume collects, aggregates, and transfers extensive data from multiple machines to HDFS. It provides a distributed, reliable and highly available service to transfer data from various devices to Hadoop.

Beyond the core components, the Apache Hadoop ecosystem includes other tools to address particular needs, as explained below.

- **Whirr:** Whirr is a set of libraries that allow using the Hadoop cluster on top of Amazon EC2, Rackspace, or any virtual infrastructure.
- **Mahout:** This is a machine learning and data-mining library to provides MapReduce implementation for popular algorithms used for analyzing data and modeling.
- **BigTop:** This is a framework for the packaging and testing of Hadoop's sub-projects and related components.
- **HBase:** HBase is a schema-based database built on top of the HDFS to read and write HDFS files.
- **HCatalog:** HCatalog is a metadata abstraction layer for referencing data without using the underlying file names or formats. It insulates users and scripts from how and where the data is physically stored.
- **Ambari:** A Web-based tool for provisioning, managing, and monitoring Apache Hadoop clusters, including support for Hadoop HDFS, Hadoop MapReduce, Hive, HCatalog, HBase, ZooKeeper, Oozie, Pig, and Sqoop.
- **Avro™:** A component for data serialization.

HADOOP CORE COMPONENT ARCHITECTURE

Hadoop stores a massive amount of data and provides a framework to process and manage it. Hadoop stores datasets in the HDFS cluster environment, which breaks files, stores them in multiple fixed blocks, and replicates them across the cluster. Its replication and cluster capability enhance the reliability and scalability for extensive data storage management.

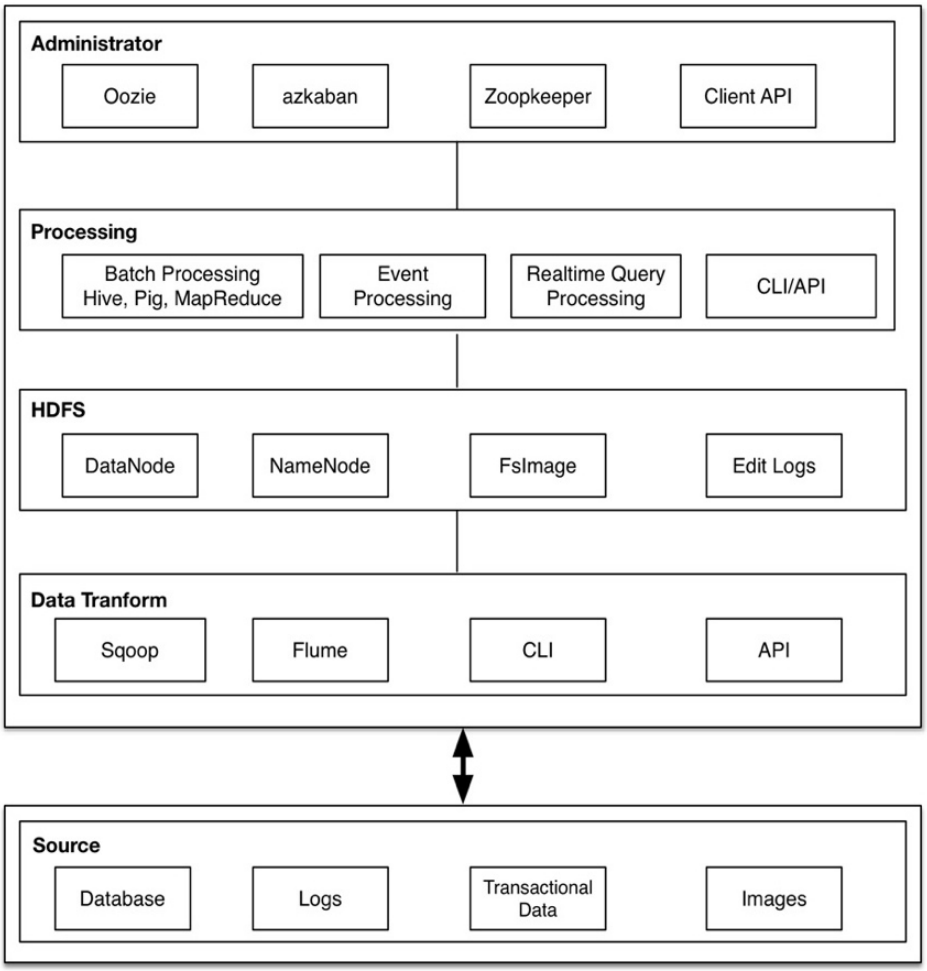

FIGURE 2.2 Hadoop's core components

Hadoop works on a multi-layer architecture where it receives data from various sources, processes them, and then optimizes them for further use. Hadoop uses batch processes (MapReduce, Hive, and Pig) to process large datasets and store the aggregated results into the HDFS, NoSQL, or imported to OLTP. The source data are transferred to the Hadoop HDFS via various mechanisms, such as FTP, Flume, Sqoop, messaging, logs, and applications. Oozie is a workflow that manages and schedules Hadoop jobs. The HDFS stores data across the clusters (DataNode), where NameNode controls the meta detail information. Fsimage and EditLogs synchronizes data states across the NameNode and DataNode.

Various processing frameworks allow processing stored HDFS data, such as batch processing, event processing, and streaming.

ZooKeeper manages Hadoop operation and provides high availability.

However, due to the amount of data stored in Hadoop, real-time access is a challenge for data transactions; HBase could be useful on partial data stored on the cluster.

SUMMARY

This chapter has provided a high-level overview of Hadoop and its ecosystem (core components). We have briefly discussed each component and their utilization as part of the platform and how each component fits together to solve Big Data challenges.

CHAPTER 3

THE HADOOP DISTRIBUTION FILESYSTEM

The HDFS is Hadoop's distributed filesystem storage system, designed to hold a large amount of data and provide access to the file system's namespace in a distributed and clustered environment. The HDFS abstracts the network complexity from storing and processing data files, and therefore it is easy to use it without thinking about the network complexity.

The HDFS is a block-structured filesystem where individual files are broken into multiple smaller chunks of a fixed size and stored across Hadoop clusters. The HDFS uses low-cost hardware to store large datasets. It supports high streaming read performance, but doesn't advocate frequent updates on the file. It is based on the concept of "write once and read many times."

Below are the salient features of the HDFS:

- ***Hardware Failure:*** The HDFS runs on a distributed environment with commonly-used hardware, so there is a fair chance for hardware failure. Therefore, quick automatic failure recovery is the primary goal of HDFS architecture.
- ***Large Datasets:*** The HDFS supports large data files (gigabytes, terabytes, and petabytes of data) by providing a large number of nodes in a cluster environment.
- ***High Throughput:*** The HDFS uses a "write-once-read-many" model, which results in high throughput data access.
- ***Data Locality:*** Data processed locally in the HDFS minimizes network congestion and increases the throughput of the system. It is always preferable to process the data where they are placed.

- ***Parallel Processing:*** The HDFS breaks big files into small blocks that support parallel data processing.
- ***Data Fault Tolerance:*** The HDFS replicates the block in multiple nodes to minimize data loss during a failure.
- ***API:*** The HDFS provides various APIs to access the data, such as WebHDFS access through the RESTful API, HDFS Java API, and the command line.

HDFS CORE COMPONENTS

The HDFS stores a file's metadata information on a single dedicated node server called NameNode and stores data on other distributed nodes called DataNodes. The HDFS NameNode abstracts and manages the file system's namespace and keeps a repository of the file's meta details. DataNode stores files in the form of fixed blocks and replicates them into other distributed DataNodes to handle fault tolerance.

The HDFS clients approach NameNode to access the file system's Name-Space information, and the NameNode point to the specific blocks on the DataNode. NameNode tracks individual DataNodes by receiving a heartbeat from each DataNode. If NameNode doesn't receive a response from any particular DataNode, then it fails that DataNode.

Figure 3.1 depicts the overall interaction between the Client, NameNode, and DataNode.

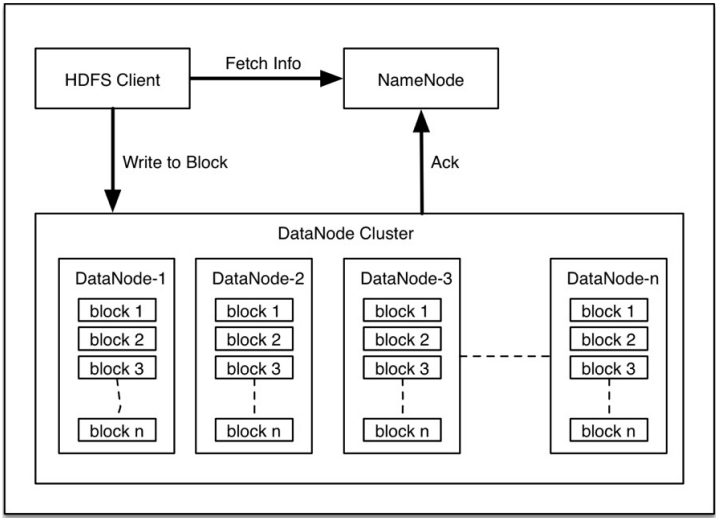

FIGURE 3.1 Client interaction with NameNode and DataNode

FileSystem Namespace: FileSystem NameSpace is similar to a traditional file system that supports all the file system's key operations such as creating a file, removing a file, moving to other directors, and re-naming a file. It promotes a traditional tree-like hierarchical directory structure.

NameNode manages and maintains FileSystem NameSpace, and it keeps all the NameSpace property and information. In case of a change to the file system, the namespace will get updated inside NameNode.

Blocks: Files in the HDFS are broken into small, fixed-size chunks called blocks, and these blocks are stored into DataNode. Its default size is 128 MB, which is much larger than the standard file block size, i.e., 512 bytes. A fixed-size block replicates for fault tolerance and availability. Using small blocks means that the small size creates a large number of files that requires considerable interaction between NameNode and DataNode, which can create overhead for the whole process.

NameNode and DataNode: The HDFS works as a master/slave architecture. HDFS clusters consist of a single NameNode, a master server that manages the file's meta information, and regulates access to data. Several DataNodes, usually one per node in the cluster, manage storage attached to the nodes that they run on. A file splits into one or more blocks, and these blocks are stored in a set of DataNodes.

The NameNode executes file system operations like opening, closing, and renaming files and directories. It also determines the mapping of blocks to DataNodes. The DataNodes are responsible for serving read and write requests from the clients. The DataNodes also perform block creation, deletion, and replication upon instructions from the NameNode.

HDFS ARCHITECTURE

The HDFS stores files across clustered machines. It stores the data as a sequence of blocks; all blocks in a file except the last block contain the same data size. The blocks replicate to other data nodes for fault tolerance and are configured by the replication factor. The HDFS provides a provision to configure the block size. The replication factor can be specified at file creation time and can be changed later. Files in the HDFS are write-once and have strictly one writer at any time.

NameNode periodically receives a *heartbeat* and a block report from each of the DataNodes in the cluster. Receipt of a heartbeat implies that the DataNode is functioning properly. It also contains a block report of all blocks on a DataNode.

NameNode maintains the file system tree and the metadata information in two files named "image log" and "edits log." NameNode is a core component of HDFS and keeps all the information of the data. If NameNode got lost, there is no way to reconstruct the file from DataNodes. NameNode could be a single point failure and should be resilient to failure by either persisting file state information or using a secondary NameNode, which operates as a replica and becomes primary.

Figure 3.2 depicts the high-level HDFS architecture.

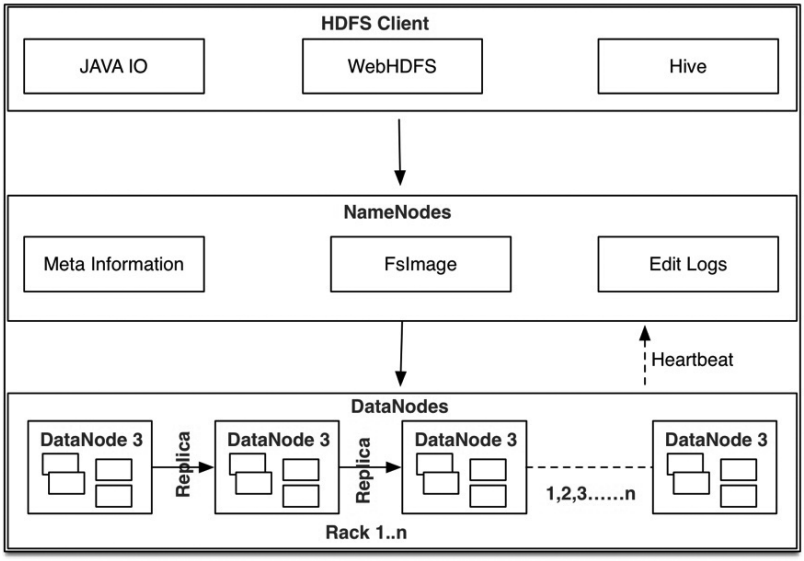

FIGURE 3.2 HDFS architecture

A fixed-size block is stored in DataNodes. However, DataNode doesn't have the visibility of the block's meta details, meaning, which block belongs to which file. NameNode is a single master node machine that stores all the files' metadata information and multiple DataNodes store files' blocks.

The HDFS uses the master/slave architecture, where the master is a NameNode that keeps track of the slave node, i.e., DataNodes. NameNode stores metadata information in the main memory that allows fast access. NameNode persists the filesystem's details in a local file called "fsimage." It also stores all transaction logs in a file called "edit log".

HDFS keeps the block size large (the default is 128 MB) to hold large amounts of disk data to ensure fast data streaming. The HDFS replicates each

block of data on more than one node (by default, three blocks) to support a high availability. It keeps the first node on the local node and replicates it to other distributed machine nodes.

DATA REPLICATION

HDFS stores blocks of data in the first local node, and once the data block is full, it communicates to NameNode to fetch the next DataNode. First, DataNode stores the data block and forwards control to the next DataNode. The blocks replicate to different DataNodes to provide fault tolerance. The replication factor is configurable, and we can specify it during file creation. NameNode uses the default replication factor (the default replication factor is three) if no replication factors are defined during file creation.

For data replication, the occupied size multiplies to the replication factor in different DataNodes. That means if a file with the size of 128 MB is stored in the HDFS with a replication factor of three, it will end up holding a space of (128*3) 384 MB, whereas each replica of the file is stored in different DataNodes. Since the files are stored as a block, therefore, blocks of files get replicated into DataNode.

NameNode keeps getting the block status from DataNode through regular heartbeat intervals and maintains the replication factor. Therefore, whenever a block gets lost and is unable to send a heartbeat, NameNode re-creates a new replica of the other DataNode.

A large cluster environment uses many DataNodes that are distributed across multiple racks. Hadoop NameNode keeps the rack information so it can place the replicated block on a different rack. There is a chance that the whole rack goes down; this kind of failure scenarios is handled efficiently by copying blocks onto a separate rack. Replication onto different racks provides high data availability and stable fault tolerance. NameNode places a unique ID on the replicas to avoid having the same two copies of a block on the same rack. Hadoop rack awareness distributes replicas across racks; unfortunately, this increases the I/O costs because of the transfer of blocks across the racks.

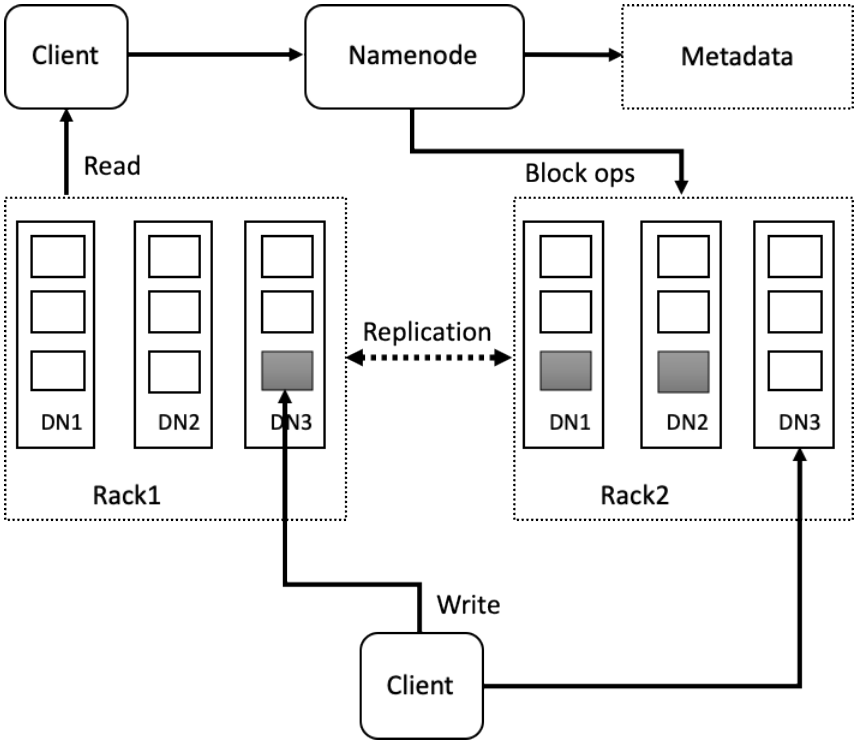

FIGURE 3.3 Data replicated on different racks

Figure 3.3 shows that when the replication factor is three, NameNode places the first block on the local DataNode and another two replicas on a different remote rack. In general, the two replicas are placed on the same rack, which reduces inter-traffic costs and improves the writing performance. This policy enhances data availability; however, it doesn't guarantee the data are evenly distributed across the rack.

NameNode also considers storage policies along with the rack awareness during replica placement. NameNode uses rack awareness to check that the candidate node has the storage required by the policy associated with the file.

DATA LOCALITY

The Hadoop execution paradigm works using the data locality, meaning it brings processing tasks to the data. Hadoop stores large datasets into many fixed blocks that can execute independently. When a MapReduce process is launched, it spawns multiple tasks to utilize each block's data in parallel. Data

locality allows for executing tasks on the node where the block resides, reducing unnecessary I/O transfers of data to the pipeline task. Hadoop deals with massive Data sets; hence, if the data moves across nodes, it degrades the overall pipeline execution performance.

Hadoop runs a MapReduce task first to execute the individual task on the same DataNode where data block resides. Hadoop prefers to perform mapper tasks on the same DataNode where the data is stored. However, that is not always possible for many scenarios. In that case, Hadoop uses different data locality policy levels with an order of priority (DataNode level locality, Rack-level Locality, and Off-rack-level locality). That means if the task not able to get free slots on the same DataNode, then the mapper task will try to find loose slots on the same rack. If the Mapper task doesn't see any open slots on the same rack, it will try to find different locations on the same Off-rack.

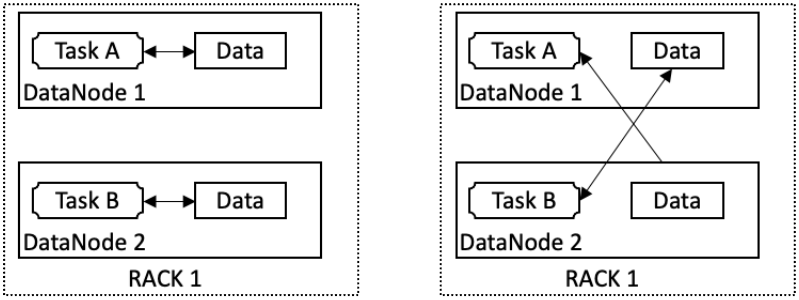

FIGURE 3.4 Data locality vs. rack locality

Figure 3.4 shows that a task will try to find free slots on the same rack on the same DataNode. If it gets an open slot on the same DataNode, it will launch a task there. If it cannot find a free slot on the same DataNode, the map task will try to find on the same rack as shown on the right in Figure 3.5. If it gets free, the slots will launch a task on the same rack.

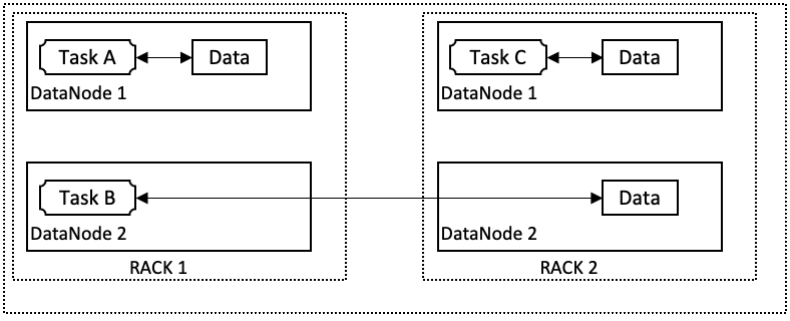

FIGURE 3.5 Rack locality vs. Off-rack locality

Figure 3.5 shows that Hadoop, not being able to find free slots on the same rack, tries to find an open slot on the Off-rack. Hadoop is always aware of the rack topology, meaning that Hadoop has full knowledge of how the data placed on DataNode. Hadoop uses this knowledge to allocate task slots near to data. Data locality gives a better latency, as data network bandwidth can be very high on the same rack.

DATA STORAGE

To keep up with the growing amount of data, the HDFS needed better data storage management. There are different categories of data, like archive data or frequent data, and so storage also required such categories. The latest Hadoop storage policy supports various storage types for HDFS data.

Hadoop's data storage policy decoupled storage data from the compute capacity. We can keep cold data that is less often used or archival in cold storage, whereas very frequently used data can go into hot storage. This strategy improves the efficiency of HDFS storage. We can have access to data in RAM and SSD, which are very responsive. The data storage policy is applied during the data replication process.

Storage Types

The HDFS works with various physical storage media, such as hard drives (DISK), SSD, and RAM, where DISK is the default storage type.

- ARCHIVE – An archival storage type for rarely used data. This storage is cheaper than a regular hard drive, where the read/write performance is not that good.
- DISK – This is the hard disk storage type. It is moderately inexpensive, and provides sequential I/O performance.
- SSD – Solid-state drives exhibit a high read/write performance. However, it is more expensive than a hard disk and the archive storage type. It is the perfect choice to store frequently used computational data with I/O-intensive applications.
- RAM_DISK – This in-memory storage type is used to accelerate low-durability, single-replica writes.

Storage Policies

The storage policy allows the HDFS to store files and the directory in different storage types.

The HDFS uses six key storage policies:

- Hot Storage: Hot storage is used for frequent data access. Data is stored on a disk and is quickly accessed for compute process. All replicas are stored on the disk.
- Cold Storage: Cold storage is used when the data are not frequently accessed and stored for archival purposes. Replica data are stored in the archive when the policy is set as "cold."
- Warm Storage: This type of storage is in between cold and hot storage. It means some replicas are stored on the disk as hot data and other replicas are stored inside the archival storage as cold data. In general, one replica is stored inside the disk and the other is stored in the archive.
- ALL_SSD: For ALL_SSD, all replicas are stored on the SSD.
- One_SSD: For this type, one replica is stored on the SSD and the rest of the replicas are stored on the disk.
- Lazy_Persist: For this type, the data are first stored in RAM, and after some time, they are stored on a persistent disk.

We can choose different storage types to optimize data usage. We define the storage policy, which explains the storage type to apply when storing the file/directory. A storage policy manages the information, such as the policy ID, policy name, list of storage types, and fallback storage type, to apply it correctly during the creation of a file/directory.

A storage policy consists of the following:

1. Policy ID
2. Policy name
3. Storage types list
4. Fallback storage types list
5. Fallback storage types for replication

Storage policies use the following criteria to assign storage types:

1. If there is sufficient space and there is no out of space issue, then the data replicas use the storage type listed in Point #3.
2. If the system is out of space on the listed storage type in Point #3, then the fallback storage type of Points #4 and #5 is used.

FAILURE HANDLING ON THE HDFS

DataNode Failure

The DataNode keeps sending health checks to NameNode when DataNode fails and cannot address health checks. NameNode removes the block from the pipeline and re-replicates it to a different DataNode.

NameNode Failure (SPOF)

The HDFS architecture uses the master/slave model, which causes a single point of failure. If DataNode fails, we can get data from a replica DataNode, but if NameNode fails, there is no standard way to recover the data automatically. There are various mechanisms to handle a NameNode failure, and these are discussed in the next few sections.

Solution 1: Secondary NameNode Failover

A partial recovery is possible using secondary NameNode. The secondary NameNode stores the state of the HDFS NameNode, i.e., the image file and transaction history of DataNodes (the edit log files). If NameNode fails, the secondary NameNode reconstructs the current state by reading the image file and edit log. However, the drawback is that it is only used for the checkpoint, not as a back-up for NameNode, and if NameNode fails, it needs to be started manually. Figure 3.6 shows how the NameNode failure works in these scenarios.

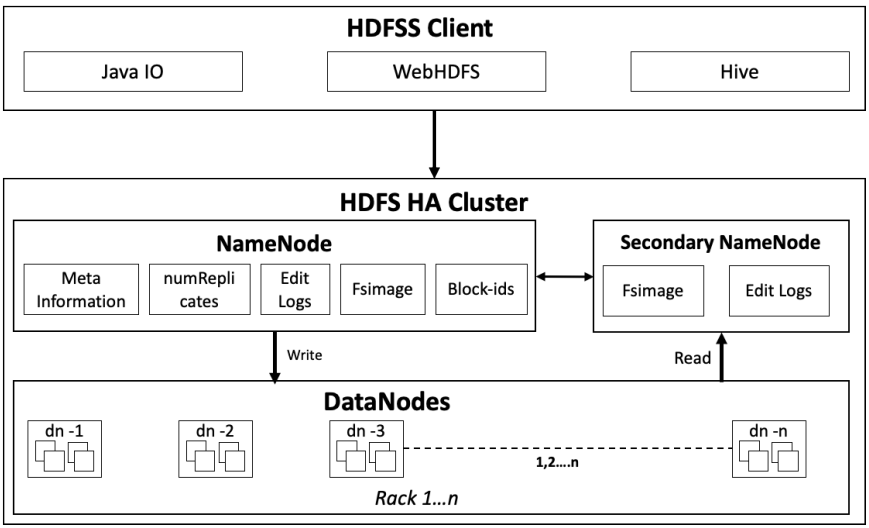

FIGURE 3.6 A NameNode failure

Solution 2: Active/Passive Failover (High Availability)

Hadoop 1.x has a single NameNode; that means if NameNode crashes, then the entire cluster is unavailable until NameNode is either re-started or brought up on a new host. In Hadoop 2 and Hadoop 3, high availability is achieved by two NameNodes, one active and other passive or on standby. The active NameNode serves all the requests, whereas the standby NameNode keeps up to date by the active NameNode with the same states. In case of failure, the standby NameNode takes over as the active NameNode to manage the requests.

Apache Hadoop uses the Quorum Journal Manager (QJM) to provide a high availability (HA) feature, which addresses the SPOF issue by providing a redundant NameNode in the same cluster/passive configuration. The QJM allows for a high availability on NameNode. QJM runs on each NameNode and communicates with the JournalNode daemon. JournalNode runs on the distributed environment with a machine in a cluster.

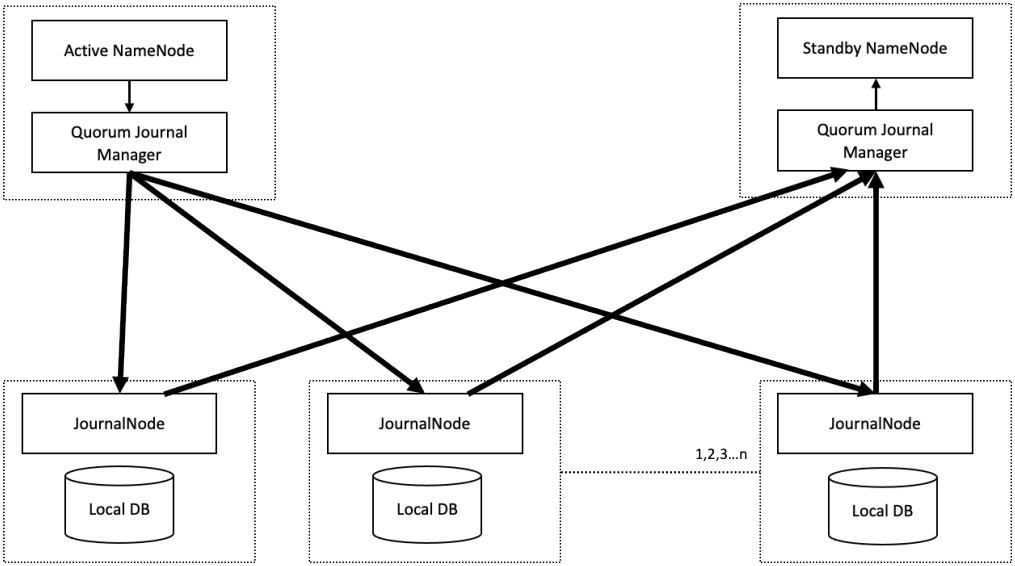

FIGURE 3.7 Quorum Journal Manager

Figure 3.7 shows the edit log's change state in one of the local disk's storage through JournalNode. The standby NameNode reads these changes from JournalNode and updates its Fsimage to keep its state in sync with the active NameNode. In the case of a failure, the standby NameNode synchronizes all latest states from JournalNode and converts into active NameNode.

QJM writes to JournalNode exclusively (only one writes at a time), making sure that no other writers are writing to the edit logs. This way, QJM guarantees that if two NameNodes are trying to edit logs, only one NameNode will be allowed to write to the JournalNode.

A Quorum JournalManager sends the RPC request to JournalNode for the write confirmation. Once Quorum JournalManager receives an acknowledgment, it starts writing the next message.

Note: In Hadoop 3, we can keep more than two NameNodes, and the DataNode will send information to all of those NameNodes. In this way, QJM helps in achieving a high availability.

Figure 3.8 depicts how the QJM synchronizes the state between the active NameNode and standby NameNode.

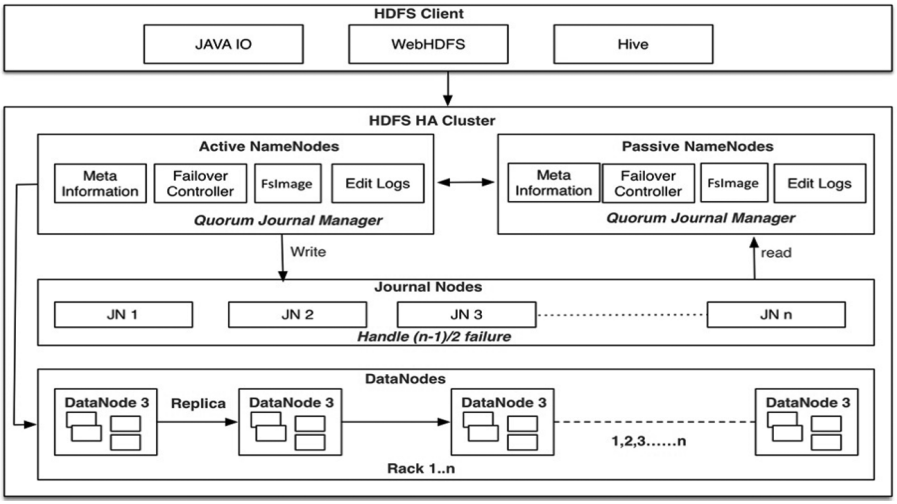

FIGURE 3.8 Quorum Journal Manager state synchronization

The active and passive (standby) NameNodes communicate to a group of daemons called JournalNodes (JNs) to synchronize their states. Active NameNode logs transaction records on JournalNodes and standby NameNode is constantly watching the JournalNodes. If it sees any change, it updates to its NameNode via the Journal Manager. The active and standby NameNodes store the latest DataNodes' information. This allows for a fast transfer to a new NameNode in the case a machine crashes or an administrator-initiated failsafe for the purpose of planned maintenance.

Journal Nodes also make sure only active NameNodes allow writing on JournalNodes. During a failover, the NameNode, which is to become active,

simply takes over the role of writing to the JournalNodes, which effectively prevents the other NameNode from continuing in the active state.

ERASURE CODING (EC)

HDFS uses RAID(1) to replicate and provide an effective way to handle disk failure at the cost of disk storage. The HDFS stores data into fixed-size blocks across the DataNode. Data get replicated to multiple copies. By default, HDFS keeps three copies of each data block. The primary purpose of keeping replicated data is to handle failure scenarios; however, it puts a greater burden on the overall HDFS storage space. To handle failure scenarios, we are keeping multiple replicas of cold storage, which is expensive. For example, suppose you have 1 TB of data and the replication factor 3, which means you require 3 TB of space to store that data and its replicas. Imagine if you have 1 PB of HDFS data for your organization, and it keeps growing day by day. You have to support 2 PB more space for the replica as cold storage that can only be utilized in the case of a failure.

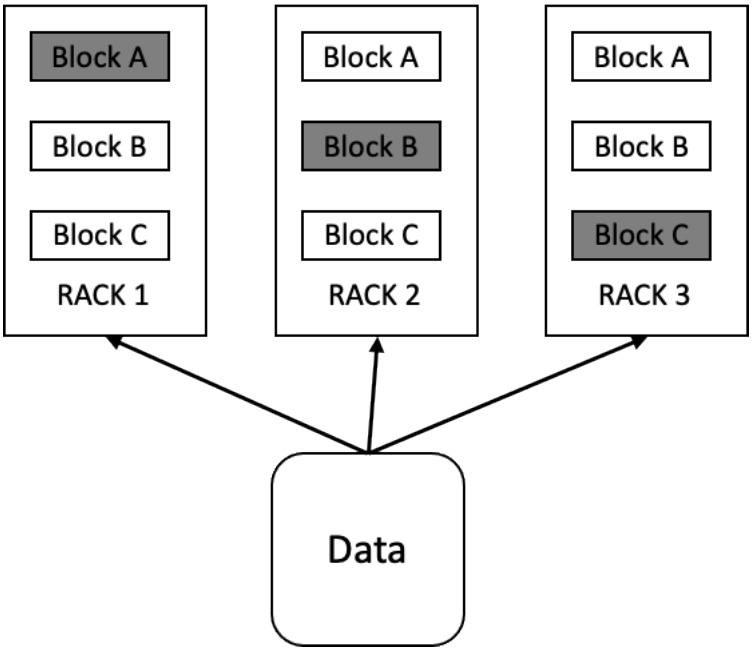

FIGURE 3.9 Data replication-caused storage overhead

Let's look at Figure 3.9. Assume one data file breaks into three blocks stored into three different DataNodes in a separate rack. To save one data file with three blocks requires nine blocks with three replication factors.

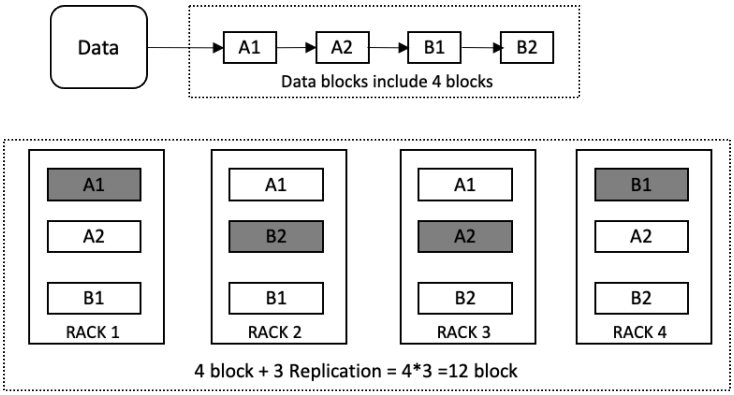

FIGURE 3.10 Utilized blocks without EC

The replication factor is an effective way to handle failure at the cost of storage overhead. N-times replication can handle n-1 failures; for example, three replication factors can simultaneously handle two failures, but at the expense of 200% storage overhead.

To solve replicated storage problem, Hadoop 3.0 introduced erasure coding.

Erasure coding (EC) reduced the replication factor from 3 to about 1.4. By using this, we can reduce the storage overhead by 50% while maintaining the same durability guarantees. It's one of the most significant changes introduced in Hadoop 3.x.

Erasure encoding uses the XOR algorithm to retrieve data in case of disk failure. Let's assume we have 3 data cells, x, y, and z, where block z stores the parity bit of x and y (x⊕y). A data file splits and stores data into blocks x and y, whereas block z stores the parity bit of x and y (x⊕y). In the case of disk failure for blocks x and y, we can easily re-construct the information using block z and other non-failure disks.

Table 3.1 Erasure Coding

x	y	z(x⊕y)
0	0	0
0	1	1
1	0	1
1	1	0

Note: EC generates only one parity bit. If any bit is lost, it can create by the remaining data cells and parity bits since it produces only one parity bit so that the XOR operation can tolerate only one failure with N group size. Still, we benefit from better storage efficiency by using the XOR algorithm.

Erasure Coding & Decoding

Erasure coding creates parity cells based on uniform input data cells. The group of parity cells and data are called an *erasure coding group*.

In RAID, data gets stripped into sequential smaller chunks (such as bits, bytes, or blocks) and stored onto different disks. Theses striping chunks are called *striping cells*. For each strip of original data cells, a certain number of parity cells are saved onto a separate disk—a parity cell called the erasure encoding. The group of parity cells and data is called an *erasure coding group*. Erasure decoding generates lost cells by decoding the remaining cells (surviving cells and parity cells).

EC improves the disk storage efficiency with a similar data durability as provided on traditional Hadoop (Hadoop 1.x and 2.x). For example, a file stored into 4 blocks will require 12 (4 * 3, for a replication factor of 3). But with EC, it only needs 6 blocks of disk space (4 data blocks and 2 parity blocks).

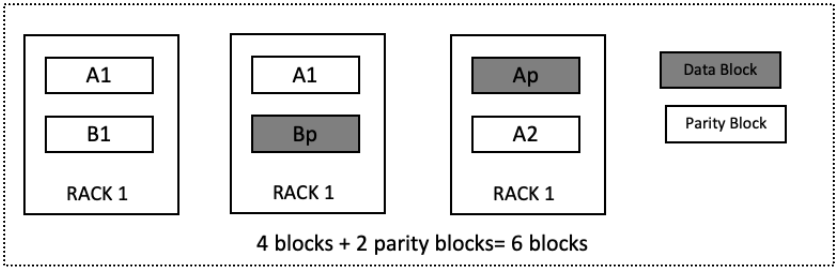

FIGURE 3.11 Erasure Coding optimized occupied blocks

Let's say the size of one block is 128 MB to store 4 blocks with a replication factor of 3 in Hadoop 2.x, and so it requires 1536 MB of space (128*4*3). With EC, it requires 768 MB of space (128*4+128*2). EC optimized this amount of space by about 50% less than the area acquired by the Hadoop 2.x replication method.

HDFS DISK BALANCER

Data is stored as a block into the HDFS using the round-robin (default) or the available space policy. The available space policy allocates new blocks based

on the free space on the disk while the round-robin policy allocates blocks evenly on each drive.

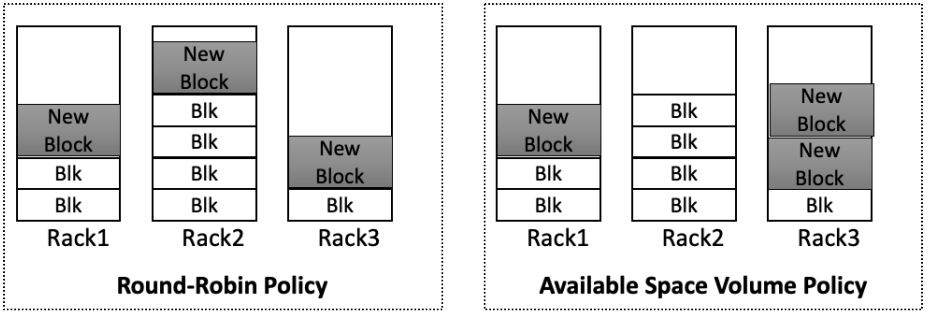

FIGURE 3.12 HDFS disk balancer

DataNode uses the preferred policies to allocate new blocks. However, many incidents happen in long-running durations, such as large amounts of data deletion or added new disk data allocation, which cause a data volume imbalance across the cluster. An uneven volume distribution impacts the overall HDFS performance in terms of efficient disk I/O.

Hadoop 3.x introduced the intra-node disk balancer to balance the data across each data node's disk. Hadoop's disk balancer improved write performance to run the process on the drive. The disk balancer is a command-line tool (CLI) that distributes data evenly on all disks in the cluster. This tool runs on a DataNode and moves blocks from one drive to another.

The disk balancer requires that you create a plan and then execute that prepared plan. A plan is a set of statements that describe how much data should move between two disks. Below is the command used to create a plan:

```
hdfs diskbalancer -plan node1.mycluster.com
```

This plan command writes two output files. They are `<nodename>.before.json`, which captures the cluster's state before the disk balancer, and `<nodename>.plan.json` after the disk balancer. A plan is composed of multiple move steps. A move step has a source disk, destination disk, and several bytes to move. Once the plan is created, the execute command takes the plan and runs it on the DataNode.

```
hdfs diskbalancer -execute
/system/diskbalancer/nodename.plan.json
```

HDFS FEDERATION

The current architecture uses a single NameNode and NameSpace on a cluster to manage files, directories, and blocks. This single NameNode architecture is simple and straightforward, but it can be challenging to scale. NameNode can become a bottleneck when it comes to scaling.

HDFS ARCHITECTURE AND ITS CHALLENGES

The HDFS clusters can scale horizontally by adding DataNode; however, NameSpace can only run on a single NameNode. That means NameNode uses a single node to store the entire file system, directories, and blocks details. It has limitations on NameNode, and the performance degrades in a large cluster environment.

The HDFS architecture is a master-slave topology where NameNode works as a master daemon responsible for managing slave nodes called DataNodes. NameNode keeps the NameSpace to maintain the file system tree and the metadata for all the tree files and directories. NameNode also stores and holds the metadata details of the blocks, files, and directories. NameNode maintains and manages the DataNodes and assigns a task to them.

As shown in Figure 3.13, the HDFS architecture has two layers: the NameSpace layer and block storage layer.

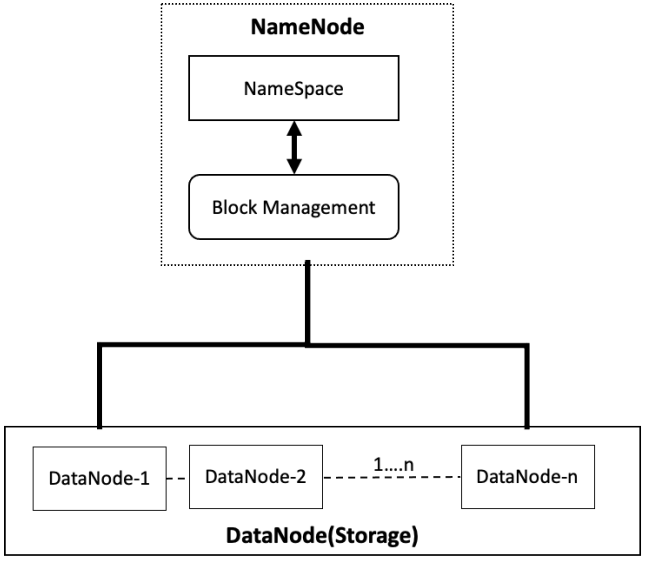

FIGURE 3.13 HDFS layered architecture

NameSpace layer: The NameSpace layer manages directories, files, and blocks. This layer supports essential files operation such as listing, creation, the deletion of files, and folders.

Block storage layer: This layer has two parts: block management and physical storage.

Block Management (In NameNode): Block management supports block operations such as block creation, deletion, and modification. Block management registers block membership and performs periodic heartbeats to validate the health of the DataNodes. It also supports the block replication specific operation.

Storage: This stores blocks on the DataNode and provides read/write access on it.

In the HDFS architecture, only a single NameSpace is allowed, and a single NameNode manages that NameSpace. If NameNode fails, the whole cluster goes down until NameNode restarts.

In a Hadoop environment, a cluster is deployed in a multi-tenant environment shared by many organizations. A single NameNode doesn't provide any isolation. The nodes internally depend on each other without knowing whether they will cause trouble to the others. Not only are the nodes interdependent, but in the case of a single NameNode fail, the entire cluster will fail.

HADOOP FEDERATION: A RESCUE

The HDFS Federation overcomes existing HDFS architecture limitations by supporting many NameSpaces and NameNodes. These NameNodes are federated, meaning the NameNodes are independent and do not coordinate with each other. The DataNodes store all blocks and each DataNode registers with all the NameNodes in the cluster.

The HDFS Federation solves challenges on the existing HDFS architecture through a clear separation between the NameSpace and Storage layers. It allows more than one namespace in the cluster environment to improve scalability, performance, availability, and isolation. Each NameNode manages its NameSpace and works independently. It doesn't coordinate to any of the others.

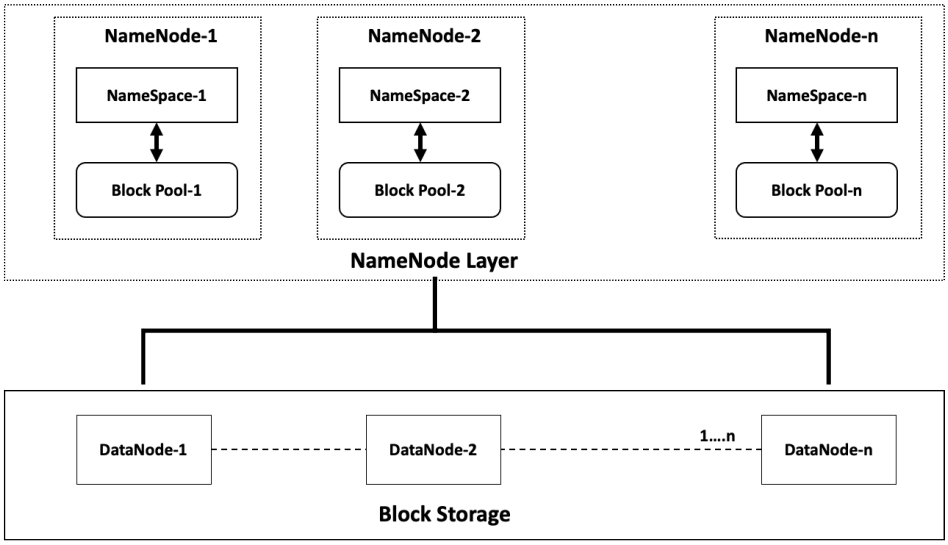

FIGURE 3.14 HDFS Federation

The HDFS Federation architecture uses many federated NameNodes/NameSpaces to scale NameNode horizontally. DataNode stores blocks and registers all the NameNodes. Each NameNode maintains its NameSpace and keeps its block pool. For example, NameNode 1 keeps NameSpace 1 and preserves its block pool that is stored in DataNode 1. The state of NameNode1 is not aware of DataNode details other than DataNode1. The HDFS Federation architecture is the most crucial feature to scale the NameNode. Along with the NameNode scalability, it also is backwards-compatible, meaning a single NameNode configuration also works without any changes.

HDFS federated NameNode/NameSpace manages its DataNode and data indexes located on the DataNode. Each NameNode's indexed data flows through towards it. For example, let's assume there are two NameNodes (NameNod1 and NameNode2) with supported data (/Data1/ and /Data2, respectively). If the data flows through NameNode1, it goes to /Data1/, and if the data flows through NameNode2, it goes to /Data2/. However, if a client connects to NameNode1 and tries to access DataNode2, it will throw an exception.

BENEFITS OF THE HDFS FEDERATION

Scalability

The HDFS Federation scales multiple NameNodes horizontally. It helps on a vast scale to distribute the load across numerous NameNodes.

Isolation

HDFS Federation categorizes different types of storage and isolates it to different NameSpaces by using many NameNodes. It segregates the data volume for users and applications and improves isolation.

Modular

Multiple NameNodes with isolation support the modular Hadoop architecture.

Performance

The NameNode keeps NameSpace and meta details in its memory.

Adding more NameNodes scales the meta details across the cluster to improve the file system's read/write throughput.

HDFS PROCESSES: READ AND WRITE

HDFS keeps files in multiple blocks and replicates blocks in different DataNodes to handle failovers on data. NameNode points out the sequence of blocks to the HDFS client. The HDFS client uses the FileSystem API to stream the file into the input stream and output stream.

Here, NameNode's responsibility is to point out the block location to the client, and afterward, the client takes care of reading and writing. It uses DataOutputStream and DataInputStream to write and read into HDFS files.

The HDFS uses FileSystem, which is similar to java.io.File but uses a large fixed size (e.g., 64 Mb or 124 Mb). The FileSystem is a crucial interface and has different implementations, such as DistributedFileSystem and LocalFileSystem.

The HDFS client accesses FileSystem, which in turn invokes NameNode to get the block information. FileSystem uses FSDataInputStream and FSDataOutputStream to read and write, respectively, from a sequence of blocks. FSDataInputStream and FSDataOutputStream are wrappers on DataInputStream and DataOutputStream, which provide extra features, such as retrieving specific positions, seeking, buffering, status, and synchronization.

Figure 3.15 depicts how the client uses NameNode and DataNode to read and write the HDFS file.

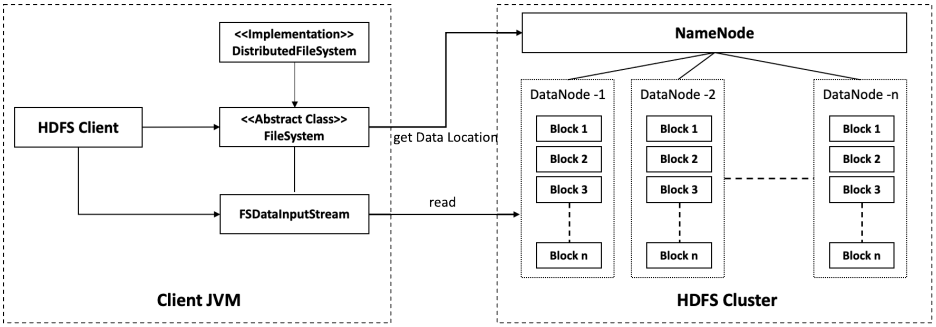

FIGURE 3.15 HDFS read and write

- Client calls `open()` on FileSystem to read the file, which creates an instance of DistributedFileSystem.
- DistributedFileSystem gets the DataNode location from NameNode.
- NameNode keeps the location of DataNode where the data are stored.
- Based on the DataNode location, DistributedFileSystem returns FSDataInputStream, an input stream to the client.
- The client calls `read()` on the DataNode addresses and connects to the first DataNode for the first block in the file.
- The data is streamed from the DataNode back to the client by calling the read method.
- Once the DataNode reaches the end, FSDataInputStream closes the connection for DataNode, then finds the next DataNode for the block.
- FSDataInputStream keeps calling the NameNode to retrieve the DataNode location for the next block.
- Once the client finishes reading, it calls close on FSDataInputStream

Figure 3.16 depicts how the new file creates and writes data into it.

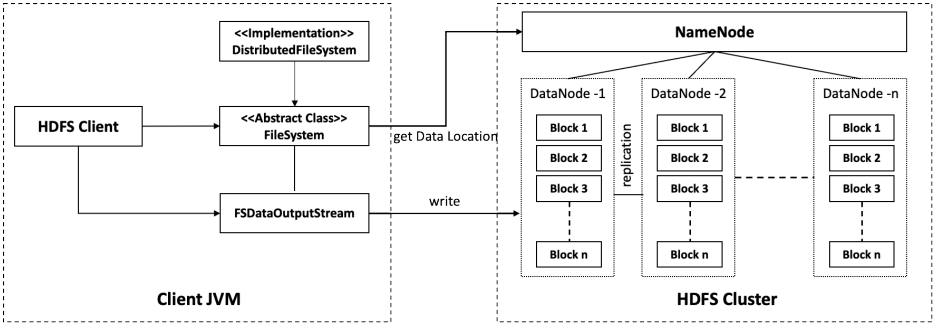

FIGURE 3.16 HDFS file actions: create, write, and read

The client creates a file by calling `create()` on DistributedFileSystem.

- DistributedFileSystem makes a call to the NameNode to create a new file in the filesystem's namespace.
- The NameNode performs various checks, such as whether the client has the right permissions or the file doesn't already exist, before creating a new file.
- The DistributedFileSystem returns an FSDataOutputStream to the client to start writing data.
- Client call `write()` on FSDataOutputStream, which splits the data and writes into a data queue.
- Data Streamer invokes NameNode to allocate new blocks from DataNode and stores the data from the data queue.
- Each pipeline contains maximum N DataNodes, so it allows N numbers of the replica for that DataNode.
- The Data Streamer writes the data packets to the first DataNode in the pipeline, which stores data packets and forwards it to the second DataNode in the pipeline.
- Similarly, the second data node forwards it to the third DataNode and so on.

FAILURE HANDLING DURING READ AND WRITE

If any error occurred during reading, then DFSInputStream would try the next closet block. DFSInputStream would also remember the failed DataNode so it would not retry again. This DataNode-NameNode design allows the HDFS to scale to a large number of concurrent clients in a cluster environment.

If a DataNode fails while data writing, the HDFS follows the steps below to handle the failure.

- The pipeline will close, and the packets in the Ack (Acknowledgment) queue add to the data queue.
- The remaining DataNode in the pipeline is reassigned and stores unique IDs into NameNode.
- The failed DataNode is removed from the pipeline, and the rest of the data is written to the remaining good DataNodes.
- Once the data is finished writing, the client calls `close()` on the stream to flush all the remaining data packets to the DataNode pipeline and waits for an acknowledgment before informing NameNode of the completion.

CHAPTER 4

GETTING STARTED WITH HADOOP

There are three ways to setup Hadoop:

- **Local Mode:** The local mode is for when you need to run a Hadoop job locally on a single machine. We can set up Hadoop on a computer by unzipping the Hadoop distribution to the local Hadoop set-up. The local mode is useful for testing a Hadoop job.
- **Pseudo Distribute Mode:** This simulates a distributed cluster environment on a single machine. It is useful for testing and validating a distributed environment on a single device.
- **Distributed Mode:** This mode is best suited for the production environment. Hadoop can be set up on a cluster of distributed machines from a few nodes to thousands of nodes. Distributed mode is for the production environment.

Hadoop Setup in Single Node Cluster

Installing Hadoop is simple. We need to untar the Hadoop tar on the cluster nodes. Below is a step-by-step explanation of how to set up Hadoop 3.x on a **single-node cluster.**

Prerequisites

- Java 8+ installed
- Dedicated user for Hadoop
- SSH configured

Platform

Hadoop can be used with MacOS, and we would follow the same steps to install it on a Linux machine. To install it on Windows, we would need to install Cygwin to support the shell.

Download

- Download the tar from *http://hadoop.apache.org/releases.html*
- Extract the tar into */Application/hadoop-3.x*

Set Up the Environment

```
$ export HADOOP_HOME=/Application/hadoop-3.x
$ export PATH=$HADOOP_HOME/bin:$PATH
$ export PATH=$HADOOP_HOME/sbin:$PATH
```

Note: We could also add to the above the command bash profile to avoid repeating above steps.

Create Directories

Create the NameNode and DataNode directories:

- `$ mkdir -p $ HADOOP_HOME/data/hdfs/namenode`
- `$ mkdir -p $ HADOOP_HOME/data/hdfs/datanode`

Change in core-site.xml

Change the following in */Application/hadoop-3.x/etc/hadoop/core-site.xml*:

```
<!-- Put site-specific property overrides in this file. -->
<configuration>
<property>
  <name>fs.defaultFS</name>
  <value>hdfs://localhost:9000/</value>
 </property>
</configuration>
```

Change in hdfs-site.xml

Change the following in */Application/hadoop-3.x/etc /hadoop/hdfs-site.xml*:

```
<configuration>
    <property>
        <name>dfs.name.dir</name>
```

```
        <value>/Users/joe/data/ndn</value>
    </property>
    <property>
        <name>dfs.data.dir</name>
        <value>/Users/joe/data/dnd</value>
    </property>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
    <property>
        <name>dfs.block.size</name>
        <value>134217728</value>
    </property>
</configuration>
```

We can configure MapReduce in *mapred-site.xml*:

```
<configuration>
    <property>
        <name>mapred.job.tracker</name>
        <value>localhost:9001</value>
    </property>
   <property>
        <name>mapred.local.dir</name>
        <value>/Users/joe/data/local</value>
    </property>
    <property>
        <name>mapred.system.dir</name>
        <value>/Users/joe/data/system</value>
    </property>
    <property>
        <name>mapred.child.java.opts</name>
        <value>-Xmx1024M</value>
    </property>
</configuration>
```

For configuring Hadoop YARN, we have to configure the *mapred-site.xml* and *yarn-site.xml* files:

mapred-site.xml

Change the following in */Application/hadoop-3.x/etc/hadoop/mapred-site.xml*. If it is not available, then create one:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl"
href="configuration.xsl"?>
<!-- Put site-specific property overrides in this file. -->
<configuration>
<property>
   <name>mapreduce.framework.name</name>
   <value>yarn</value>
</property>

</configuration>
```

yarn-site.xml

Change the following in */Application/hadoop-3.x/etc/hadoop/yarn-site.xml*:

```
<configuration>
<!-- Site specific YARN configuration properties -->
<property>
   <name>yarn.nodemanager.aux-services</name>
   <value>mapreduce_shuffle</value>
</property>
<property>
   <name>yarn.nodemanager.env-whitelist</name>
   <value>JAVA_HOME,HADOOP_COMMON_HOME,HADOOP_
   HDFS_HOME,HADOOP_CONF_DIR,CLASSPATH_PREPEND_
   DISTCACHE,HADOOP_YARN_HOME,HADOOP_MAPRED_HOME</value>
</property>
</configuration>
```

Logging

Update the *conf/log4j.properties* file to customize the Hadoop logging configuration. Hadoop uses the Apache *log4j* via the Apache Commons Logging framework.

Format namenode

```
$ hadoop namenode -format
```

The above command yields the following result:

```
/************************************************
******
```

```
SHUTDOWN_MSG: Shutting down NameNode at
username.local/xx.yyy.zz.aa
************************************************
*****/
```

The format namenode command should be performed once to create the NameNode and metadata. The format command initializes the new file system and creates an empty fsimage and edit log. If you re-format it again, it will flush the existing block data and metadata.

Start the HDFS server

Run the jps command using the following:

```
$jps
  782
912 Jps
```

This means that as of now, the HDFS NameNode and DataNode did not start yet.

Start the NameNode and DataNode Daemons

```
$ sbin/start-dfs.sh
$ jps
    1305 Jps
    1238 DataNode
    1201 NameNode
```

Start Resource Manager

Use the following command to start the ResourceManager daemon and NodeManager daemon:

```
$ sbin/start-yarn.sh
```

Below is the default URL for the Web interface for the ResourceManager:

`ResourceManager - http://localhost:8088/`

Now we can run the MapReduce job. To stop the daemons, use the command below:

```
$ sbin/stop-yarn.sh
```

HADOOP CONFIGURATION

io.file.buffer.size

The buffer size (`io.file.buffer.size`) defined in *core-default.xml* is to set the buffer size according to the read and write operation. By default, its size is 4096 bytes, but we can increase the file buffer size using the `io.file.buffer.size` configuration parameter. More data can be transferred efficiently, but increasing the buffer consumes more memory.

dfs.block.size

The block size (`dfs.block.size`) defined in *hdfs-default.xml* is to set the block size for newly created HDFS files. By default, its value is 128 MB. The code `dfs.block.size` doesn't affect files that already exist with different block sizes. It is client-specific and has no impact on the NameNode and DataNode, except for the MapReduce job.

```
<configuration>
    <property>
        <name>dfs.blocksize</name>
        <value>134217728</value>
    </property>
</configuration>
```

dfs.datanode.du.reserved

Reserving space in bytes on a disk (`dfs.datanode.du.reserved`) is defined in *hdfs-default.xml* for non-dfs applications, such as a MapReduce job. By default, no space is reserved; however, it is advisable to reserve some space in each node:

```
dfs.datanode.du.reserved=<<size in bytes>>.
```

dfs.namenode.handler.count

The thread pool to handle RPC calls across the cluster environment (`dfs.namenode.handler.count`) is defined in *hdfs-default.xml*. The default value is 10, but we can increase it for a large cluster environment.

dfs.datanode.failed.volumes.tolerated

We can set the threshold for failure (`dfs.datanode.failed.volumes.tolerated`) parameter in *hdfs-default.xml*, which is treated as the threshold

to fail the entire data node. The default value is zero: if any disk fails, the entire data node fails.

Hadoop replicates the data from the DataNode for recovery at the time of disk failure. However, we need to set limits or thresholds to fail the node; otherwise, it will keep retrieving the data from a replicate node and take a considerable amount of time before failing the DataNode.

dfs.hosts

The list of hosts that allow connecting to the NameNode (`dfs.hosts`) is defined in *hdfs-default.xml*. The value is a file name with a full path, which contains the list of hosts. If there is no value, then all hosts are allowed. This property provides an extra security feature to restrict the connection to a limited number of hosts.

dfs.hosts.exclude

The property used to exclude a list of hosts to connect to the NameNode explicitly (`dfs.hosts.exclude`) is defined in *hdfs-default.xml*. The value is a filename with a full path that contains a list of hosts to be eliminated. It is also used to decommission the DataNode.

fs.trash.interval

The trash is managed using `fs.trash.interval`; it stores deleted HDFS files for a certain amount of time and is defined in *core-default.xml*. The Hadoop HDFS doesn't have an independent thrash folder, so there is no way to recover deleted files from the HDFS. Fortunately, we can define a temporary trash in the HDFS, which allows us to keep a deleted file in the */.trash/* folder for specified intervals. This definition enables us to move the deleted files and folder into the *trash* folder for time interval defined. Data moved to the trash folder provides users a chance to recover the accidentally-deleted data. The default value of zero indicates that the trash feature is disabled.

COMMAND-LINE INTERFACE

The command-line interface is the easiest way to access and manage HDFS stored files. Hadoop commands are invoked by bin/Hadoop scripts. Using Hadoop's command-line interface, we can do all file-related operations, such as reading a file, creating directories, moving/copying/deleting files, and listing files/directories.

```
Usage: hadoop [--config confdir] [COMMAND]
[GENERIC_OPTIONS] [COMMAND_OPTIONS]
```

Note: The HDFS can be referenced as `dfs`.

Hadoop has an optional parsing framework that employs parsing generic options as well as running classes.

Table 4.1 Command Line Interface

COMMAND_OPTIONS	Description
--config confdir	Overwrites the default Configuration directory. Default is ${HADOOP_HOME}/conf.
GENRIC_OPTION	The common set of options supported by multiple commands.
COMMAND COMMAND_OPTIONS	Various commands with their options are described in the following sections. The commands have been grouped into User Commands and Administration Commands

Note: Hadoop provides a file system shell utility to access the HDFS files using Unix-like commands such as delete, move, and copy. If you want to use a similar type of command in Java, there is class **FsShell** available that provides the same kinds of methods in Java by using Object.

GENERIC FILESYSTEM CLI COMMAND

The file system (FS) shell includes various shell-like commands that directly interact with the Hadoop Distributed File System (HDFS) as well as other file systems that Hadoop supports, such as Local FS, HFTP FS, S3 FS, and others. The FS shell is invoked by

```
hadoop fs –command <args>
```

All FS shell commands mentioned below take path URIs as arguments.

Table 4.2 Hadoop CLI Commands

Command	Description	Usage
appendToFile	Appends single/multiple src from local file system to the destination file system	`hadoop fs -appendToFile <local file system> <destination file system>`
Cat	Views source file	`hadoop fs -cat URI [URI …]`
checksum	Gets checksum info of a file	`hadoop fs -checksum [-v] URI`
Chgrp	Changes group association of files	`hadoop fs -chgrp [-R] GROUP URI [URI …]`
Chmod	Changes the permissions of files	`hadoop fs -chmod [-R] <MODE[,MODE]... | OCTALMODE> URI [URI …]`
Chown	Changes the owner of files	`hadoop fs -chown [-R] [OWNER] [:[GROUP]] URI [URI ]`
copyFromLocal	Copies from the local destination	`14 adoop fs -copyFromLocal <localsrc> URI`
copyToLocal	Copies from destination to the local destination	`Hadoop fs -copyToLocal [-ignorecrc] [-crc] URI <localdst>`
count	Counts the number of directories, files, and bytes under the paths that match the specified file pattern	`hadoop fs -count [-q] <paths>`
cp	Copies files from the source to the destination	`hadoop fs -cp URI [URI …] <dest>`

(continued)

Command	Description	Usage
du	Displays sizes of files and directories contained in the given directory or the length of a file in case it is just a file	`hadoop fs -du [-s] [-h] URI [URI …]`
dus	Displays a summary of the file lengths	`hadoop fs -dus <args>`
expunge	Empties the trash	`hadoop fs -expunge`
find	Similar to Unix, finds files based on patterns and applies an action	`hadoop fs -find <path> ... <expression> ...`
Get	Copies files to the local file system	`hadoop fs -get [-ignorecrc] [-crc] <src> <localdst>`
Getmerge	Takes a source directory and a destination file as input and concatenate files in the src into the destination's local file	`hadoop fs -getmerge <src> <localdst> [addnl]`
ls	For the directory, it returns list of its direct children as in Unix; For a file, it returns the status on the file	`hadoop fs -ls <args>`
lsr	Recursive version of ls. Similar to Unix ls -R.	`hadoop fs -lsr <args>`
mkdir	Takes path URIs as arguments and creates directories	`hadoop fs -mkdir <paths>`
moveFromLocal	Similar to the put command, except that the source localsrc is deleted after it is copied.	`Hadoop fs -moveFromLocal <localsrc> <dst>`
moveToLocal	Displays a "Not implemented yet" message.	`hadoop fs -moveToLocal [-crc] <src> <dst>`
mv	Moves files from source to destination	`hadoop fs -mv URI [URI …] <dest>`

Command	Description	Usage
put	Copies single src, or multiple srcs from local file system to the destination file system	`hadoop fs -put <localsrc> ... <dst>`
rm	Deletes files specified as args	`hadoop fs -rm [-skipTrash] URI [URI …]`
rmr	Recursive version of delete	`hadoop fs -rmr [-skipTrash] URI [URI …]`
setrep	Changes the replication factor of a file	`hadoop fs -setrep [-R] <path>`
Stat	Returns the status of the path	`hadoop fs -stat URI [URI …]`
tail	Displays the last kilobyte of the file to stdou	`hadoop fs -tail [-f] URI`
test	Options: (-e) check to see if the file exists. Returns 0 if true. (-z) check to see if the file is zero length. Returns 0 if true. (-d) check to see if the path is directory. Returns 0 if true.	`hadoop fs -test -[ezd] URI`
text	Takes a source file and outputs the file in text format	`hadoop fs -text <src>`
touchz	Create a file of zero length.	`hdfs dfs -touchz URI [URI …]`

DISTRIBUTED COPY (DISTCP)

The distribute copy command (`distcp`) is the Hadoop utility for copying large files across or within clusters. It uses a MapReduce job to accomplish the file movements. It distributes files and directories into individual map tasks, each of which copies the files in parallel.

```
Usage: hadoop distcp <srcurl> <desturl>
Example
hadoop distcp hdfs://nn1:8020/local/files \
hdfs://nn2:8020/desitonal/files
```

Update and Overwrite

update: This command copies to a target if the files don't exist or differ from the target version.
overwrite: This command overwrites target-files that exist at the target.

Let's assume that these are source files:

```
hdfs://nn1:8020/source/sourcefiles/a 64
hdfs://nn1:8020/source/sourcefiles/b 64
hdfs://nn1:8020/source/second/c 32
```

The destinations and sizes are as shown below:

```
hdfs://nn2:8020/target/a 64
hdfs://nn2:8020/target/b 32
```

If we apply `distcp` as shown below,

```
distcp hdfs://nn1:8020/source/sourcefiles \
hdfs://nn2:8020/target
```

then the output will be as follows:

```
hdfs://nn2:8020/target/a 64
hdfs://nn2:8020/target/b 64
hdfs://nn2:8020/target/c 32
```

update: In the case of an update, "a" is skipped because the file-length and contents match. "b" is overwritten because the contents don't match. "c" is copied because it's not in the destination.
overwrite: In the case of an overwrite, "a" will also get overwritten.

HADOOP'S OTHER USER COMMANDS

jar

This command runs a `jar` file. Users can bundle their Map Reduce code into a `jar` file and execute it using this command.

```
Usage: hadoop jar <jar> [mainClass] args...
```

Note: Use `yarn jar` to launch YARN applications instead.

Version

This command prints the version.

```
Usage: hadoop version
```

Envvars

This command displays computed Hadoop environment variables.

```
Usage: hadoop envvars
```

classname

This Hadoop script can be used to invoke any class.

```
Usage: hadoop CLASSNAME
```

classpath

The `classpath` command is used to print the class path needed to get the Hadoop jar and the required libraries.

```
Usage: hadoop classpath [--glob |--jar <path> |-h |--
help]
```

Here, `-glob` is for the wildcard , `--jar path` is the write `classpath` as manifested in the `jar` named `path` and it helps to print help.

distch

The `distch` command is used to change the ownership of objects,

```
Usage: hadoop distch [-f urilist_url] [-i] [-log
logdir] path:owner:group:permissions
```

where

- `-f`: List of files/folder to change ownership
- `-I`: Ignore failures
- `-log`: Directory to log output

HDFS PERMISSIONS

The HDFS also provides a permission model for files and directories like POSIX. Each file and directory links with the file owner and group. There is a separate permission strategy for the user and the associated group.

There are three types of permissions:

- The read permission (r): To read files or list files of a directory
- The write permission (w): To write or create/delete a file or folder, write, and append it
- The execute permission (x): To access the child of the directory. It doesn't support an execute command on the HDFS like POSIX.

The HDFS also supports Access Control Lists for POSIX. The HDFS POSIX ACL setting is an optional setting. It provides a two-phase identity check:

- The owner permission applies when the name matches with the group owner.
- The group permission is used when the name matches a group from the group list.
- Other permissions are used when the name doesn't match either of the above two scenarios.
- The client operation fails if none of the permissions can be applied.

HDFS QUOTAS GUIDE

We can set up a quota based on the size or count the HDFS directories. Both the name and space quotas work separately in parallel. If the allowances exceed the file and directory creations, then the attempts fail. Please consider replication carefully because each replication block is counted against the quota limits.

HDFS Space Quotas

HDFS space quotas is the number of bytes allowed to create a file or directory. If the quota is exceeded, the HDFS fails to allocate space to write the block. Each replica of the file counts against the quota. For example, 10 MB of files with three replication factors results in 30 MB of HDFS space consumed. In case set quotas, less than 30 MB will fail to write an HDFS file. Quotas also apply to the rename operation, and the rename operation fails in case the quota space limits are exceeded.

HDFS Name Quotas

The HDFS name quota is limited to the number of files included in the directory. If the name quota is exceeded, the HDFS fails to allocate space to write

the files and directory. The HDFS name quotas apply for the rename operation, and the rename operation fails if the HDFS name quota is exceeded. The quota is also applicable to the replication factor, meaning each replica of a file counts against the quota.

Note: The HDFS Space and Name quotas persist with the fsimage.

Storage Type Quotas

The storage type quota provides restrictions on a directory that allows a specific storage type, such as SSD, DISK, and ARCHIVE. We can use the storage type quota, name quota, and space quota to allow for a fine-grained control over the cluster storage.

HDFS Quota CLI Commands

- `hdfs dfsadmin -setQuota <N> <directory>...<directory>`

This command sets the name quota to N for each directory.

- `hdfs dfsadmin -clrQuota <directory>...<directory>`

This command removes the name quota rule on the listed directories.

- `hdfs dfsadmin -setSpaceQuota <N> <directory>...<directory>`

This command sets the space quota by N bytes for the listed directory.

- `hdfs dfsadmin -clrSpaceQuota <directory>...<directory>`

This command removes the space quotas on the listed directories.

- `hdfs dfsadmin -setSpaceQuota <N> -storageType <storagetype> <directory>…<directory>`

This command sets the storage type, along with the space quota, on the listed directories.

- `hdfs dfsadmin -clrSpaceQuota -storageType <storagetype> <directory>...<directory>`

This command removes the storage type and space quota for the listed directories.

Remove the Storage Type Quota

The storage type quota with the space quota limits a total storage type space on the given directories. For example, 1 MB of data with a replication

factor of 5 and SSD storage type consume 5 MB of the SSD quota on a given directory.

```
hadoop fs -count -q [-h] [-v] [-t [comma-separated list of
storagetypes]] <directory>...<directory>
```

This command gets reports on the HDFS quota values and current name count and bytes used on listed directories

HDFS SHORT-CIRCUIT LOCAL READS

HDFS short-circuit local reads allow the client to read data directly if the client is collocated with the data. This bypassing of the DataNode mechanism is called *short-circuit reads*. Short-circuit reads improve the reads performance. We can configure short-circuit reads by setting below parameters in *hdfs-default.xml*.

- `dfs.client.read.shortcircuit` : (default false)

This configuration parameter enables short-circuit local reads

- `dfs.domain.socket.path`

This is a path to a Unix domain socket that is used for communication between the DataNode and local HDFS clients.

- Before setting the properties, make sure to enable the native libraries:

  `libhadoop.so`

OFFLINE EDITS VIEWER GUIDE

An offline edits viewer tool is used to parse the edit log file into XML and binary format. XML allows for a human-readable format. The following input formats are supported:

- **binary:** native binary format that Hadoop uses internally
- **xml:** XML format, as produced by xml processor, used if the file name has the .xml (case insensitive) extension

The offline edit viewer provides below output processor:

- **binary:** native binary format that Hadoop uses internally

- **xml:** XML format
- **stats:** prints out statistics; this cannot be converted back to the edit file

XML Processor Usage

The XML processor creates an XML file that contains the edit log information. Users can specify input and output files via `-i` and `-o` in the command line.

```
$ bin/hdfs oev -p xml -i edits -o edits.xml
$ bin/hdfs oev -i edits -o edits.xml
```

Binary Processor Usage

The binary processor is the reverse of the XML processor. It uses XML input and creates binary output.

```
$ bin/hdfs oev -p binary -i edits.xml -o edits
```

Stats Processor Usage

The stats processor command aggregates the counts of the operation codes (opcodes) contained in the edit log file.

```
$ bin/hdfs oev -p stats -i edits -o edits.stats
```

OFFLINE IMAGE VIEWER GUIDE

The offline image viewer tool converts the HDFS fsimage file into a human-readable format and provides the WebHDFS API to analyze and examine a Hadoop cluster.

Web Processor

This provides the HTTP REST API to investigate the namespace. The CLI command below includes the address by using the `-addr` option.

```
$ bin/hdfs oiv -i fsimage
```

The CLI command to get the information about fsimage is as follows:

```
$ bin/hdfs dfs -ls webhdfs://127.0.0.1:5978/
```

The CLI command allows for the recursive use of the fsimage information:

```
$ bin/hdfs dfs -ls -R webhdfs://127.0.0.1:5978/
```

XML Processor

The XML processor converts fsimage to an XML document and includes all the information within the fsimage file. The command below allows us to create that XML file from the content of fsimage:

```
$ bin/hdfs oiv -p XML -i fsimage -o fsimage.xml
```

ReverseXML Processor

The ReverseXML processer converts XML files into fsimage, and we use the command to accomplish this:

```
$ bin/hdfs oiv -p ReverseXML -I fsimage.xml -o fsimage
```

FileDistribution Processor

The FileDistribution processor analyzes NameSpace file sizes. Users can specify the maxSize (128 GB, by default) and step (2 MB, by default) in bytes via `-maxSize` and `-step` command-line.

```
bin/hdfs oiv -p FileDistribution -maxSize maxSize -step
size -i fsimage -o output
```

We can use `-format` to make it more readable:

```
$ bin/hdfs oiv -p FileDistribution -maxSize maxSize -step
size -format -i fsimage -o output
```

Delimited Processor

This generates comma delimited text, whereas the default delimiter is `\t`.

```
 $ bin/hdfs oiv -p Delimited -delimiter delimiterString
-i fsimage -o output
```

DetectCorruption Processor

This generates a text representation of the errors in the fsimage:

```
$ bin/hdfs oiv -p DetectCorruption -delimiter
delimiterString -t temporaryDir -i fsimage -o output
```

CHAPTER 5

INTERFACES TO ACCESS HDFS FILES

Hadoop was developed using a Java application, and hence the Java API is very useful for accessing the Hadoop file system. However, various interfaces are available to access the Hadoop file system, such as WebHDFS and libhdfs.

HttpFS: Apache Hadoop HttpFS is a server that provides HTTP protocols to access the HDFS file system operations, such as read and write. It also communicates with the WebHDFS REST HTTP API.
WebHDFS: WebHDFS REST HTTP API is used to access the HDFS file system from outside of the cluster environment using HTTP REST protocols such as GET, POST, PUT, and DELETE.
JAVA API: Hadoop provides a file system similar to the Java file system to access and process HDFS files. It allows the Configuration to be set using configuration property or explicitly using a set method.
C API libhdfs: libhdfs is a native C API that provides non-Java programs to access the HDFS.

WEBHDFS REST API

WebHDFS is a framework used to access the HDFS file system based on industry-standard RESTful protocols. In traditional Java, the API uses one instance per JVM; however, the WebHDFS leverages Hadoop's cluster environment to access a parallel environment. WebHDFS is used to access the HDFS file system from outside of the cluster.

The HTTP REST API supports not only the complete FileSystem interface for HDFS, but also all kinds of HDFS user operations, such as reading, writing, creating a directory, and changing permissions. WebHDFS uses RESTful protocols and, therefore, it is language-neutral. WebHDFS leverages Hadoop's Kerberos and tokens to provide security.

Configuration

We can use the below configuration setting for WebHDFS in *hdfs-default.xml*.

dfs.web.authentication.kerberos.principal: Hadoop-Auth uses the HTTP Kerberos principal in the HTTP endpoint. It t is required when WebHDFS and security are enabled.
dfs.web.authentication.kerberos.keytab: Hadoop Auth uses the Kerberos keytab file for the HTTP Kerberos principle in the context of the HTTP endpoint. For example, `/etc/security/spnego.service.keytab`
dfs.webhdfs.socket.connect-timeout: This is the timeout time for a unit (for example, "2m" for 2 minutes or "30s" for 30 seconds) to wait to get the connection before timeout. The default value is 60s.
dfs.webhdfs.socket.read-timeout: This is how long the wait is for data to arrive before failing. The defaults is 60s.

FILESYSTEM URIS

The WebHDFS URI follows the same pattern as the HTTP URI with the operations GET, POST, DELETE, and PUT.

```
http://<HOST>:<HTTP_PORT>/webhdfs/v1/<PATH>?op=
```

The operations (op) vary based on the required operations.

Note: When WebHDFS is secured with SSL, then we should use the following:

```
swebhdfs://<HOST>:<HTTP_PORT>/<PATH>
```

Table 5.1 HTTP Operations

Op	URL	Description
GET	`url?op=OPEN [&offset=<LONG>] [&length=<LONG>] [&buffersize=<INT>]"`	Open and Read a File

Op	URL	Description
GET	`url?op= GETFILESTATUS`	Status of a File/ Directory
GET	`url? op=LISTSTATUS`	List a Directory
GET	`url? op=LISTSTATUS_ BATCH&startAfter=<CHILD>`	Iteratively List a Directory
GET	`url? op=GETCONTENTSUMMARY`	Get Content Summary of a Directory
GET	`url? op=GETFILECHECKSUM`	Get File Checksum
GET	`url? op=GETHOMEDIRECTORY`	Get Home Directory
GET	`url? op= op=GETDELEGATION TOKEN[&renewer=<USER>] [&service=<SERVICE>] [&kind=<KIND>]`	Get Delegation Token
GET	`url? op=GETTRASHROOT`	Get Trash Root
GET	`url?op=GETXATTRS&xattr.name= <XATTRNAME>&encoding=<ENCODING>`	Get an XAttr
GET	`url? op= LISTXATTRS`	List all XAttrs
GET	`url?op=CHECKACCESS &fsaction=<FSACTION>`	Check access
GET	`url? op=GETALLSTORAGEPOLICY`	Get all Storage Policies
GET	`url? op= GETSTORAGEPOLICY`	Get Storage Policy
GET	`url? op=GETSNAPSHOTDIFF&oldsn apshotname=<SNAPSHOTNAME>&sna pshotname=<SNAPSHOTNAME>`	Get Snapshot Diff
PUT	`uri<PATH>?op=CREATE[&ov erwrite=<true |false>] [&blocksize=<LONG>] [&replication=<SHORT>] [&permission=<OCTAL>] [&buffersize=<INT>] [&noredirect=<true|false>]`	Create and Write to a File

(continued)

Op	URL	Description
PUT	uri<PATH>?op=MKDIRS[&permission=<OCTAL>]	Make a Directory
PUT	uri<PATH>?op=CREATESYMLINK&destination=<PATH>[&createParent=<true|false>]	Create a Symbolic Link
PUT	uri<PATH>?op=SETREPLICATION[&replication=<SHORT>]	Set Replication Factor
PUT	uri<PATH>?op=SETOWNER[&owner=<USER>][&group=<GROUP>]	Set Owner
PUT	uri<PATH>?op= SETPERMISSION [&permission=<OCTAL>]	Set Permission
PUT	?op=SETTIMES [&modificationtime=<TIME>] [&accesstime=<TIME>]	Set Access or Modification Time
PUT	?op= RENEWDELEGATIONTOKEN&token=<TOKEN>	Renew Delegation Token
PUT	?op= CANCELDELEGATIONTOKEN&token=<TOKEN>	Cancel Delegation Token
PUT	?op=CREATESNAPSHOT[&snapshotname=<SNAPSHOTNAME>]	Create Snapshot
PUT	?op=RENAMESNAPSHOT&oldsnapshotname=<SNAPSHOTNAME>&snapshotname=<SNAPSHOTNAME>	Rename Snapshot
PUT	?op=SETXATTR&xattr.name=<XATTRNAME>&xattr.value=<XATTRVALUE>&flag=<FLAG>	Set XAttr
PUT	op=REMOVEXATTR&xattr.name=<XATTRNAME>	Remove XAttr
PUT	SETSTORAGEPOLICY&storagepolicy=<policy>	Set Storage Policy
POST	APPEND[&buffersize=<INT>] [&noredirect=<true|false>]	Append to a File
POST	CONCAT&sources=<PATHS>	Concat File(s)
POST	op=TRUNCATE&newlength=<LONG>	Truncate a File

Op	URL	Description
POST	op=UNSETSTORAGEPOLICY	Unset Storage Policy
Delete	op=DELETESNAPSHOT&snapshotname=<SNAPSHOTNAME>	Delete Snapshot
Delete	op=DELETE[&recursive=<true|false>]	Delete a File/Directory

ERROR RESPONSES

In case of service request fails, the server may throw an exception. The RemoteException is used for the JSON schema error response.

The table below shows the mapping from exceptions to HTTP response codes.

Table 5.2 HTTP Error Response Codes

Exceptions	HTTP Response Codes
IllegalArgumentException	400 Bad Request
UnsupportedOperationException	400 Bad Request
SecurityException	401 Unauthorized
IOException	403 Forbidden
FileNotFoundException	404 Not Found
RumtimeException	500 Internal Server Error

Table 5.3 HTTP Curl operations

Action	Curl
Read a File	curl -i -L "http://$<Host_Name>:$<Port>/webhdfs/usr/joe/employee?op=OPEN"
List directory status	curl -i "http://$<Host_Name>:$<Port>/webhdfs/usr/joe/?op=LISTSTATUS"
Status of file	curl -i "http://$<Host_Name>:$<Port>/webhdfs/v1/foo/bar?op=GETFILESTATUS"

(continued)

Action	Curl
Write file	`curl -i -X PUT -L "http://$<Host_ Name>:$<Port>/ webhdfs//usr/joe/ newFile?op=CREATE" -T newFile`
Rename file	`curl -i -X PUT "http://$<Host_ Name>:$<Port>/ webhdfs//usr/joe/ newFile?op=RENAME&destination=/usr/joe/ john"`
New Directory	`curl -i -X PUT "http://$<Host_Name>:$<Port>/ webhdfs/usr/employer?op=MKDIRS&permissi on=711"`

AUTHENTICATION

An HTTP token and Kerberos SPENGO provide authentication security. If the token is not available, then by default, it authenticates by Kerberos SPNEGO. If security is disabled, then we can specify the username in the "user.name" query parameter.

Below are examples using the curl command tool:

```
curl --negotiate -u:anyUser
"http://$<Host_Name>:$<Port>/
webhdfs/usr/joe?op=OPEN
curl --negotiate -u:anyUser -b ~/cookies.txt
-c ~/cookies.txt
http://$<Host_Name>:$<Port>/webhdfs/usr/joe?op=OPEN
```

where

- The `--negotiate` option enables SPNEGO in curl.
- The `-u:anyUser` option is mandatory when the user name is not specified. Instead, a Kerberos established user (via kinit) is used.
- The `-b` and `-c` options are used for storing and sending HTTP cookies

Below are the curl-based authentication requests:

- Authentication when security is off:

```
curl -i
http://<HOST>:<PORT>/webhdfs/v1/<PATH>?[user.
name=<USER>&]op=...
```

- Authentication using Kerberos SPNEGO when security is on:

```
curl -i --negotiate -u :
"http://<HOST>:<PORT>/webhdfs/v1/<PATH>?op=..."
```

- Authentication using the Hadoop delegation token when security is on:

```
curl -i
"http://<HOST>:<PORT>/webhdfs/v1/<PATH>?delegatio
```

JAVA FILESYSTEM API

Hadoop's `org.apache.hadoop.fs.FileSystem` is a generic class to access and manage HDFS files/directories in Java. The file's contents are stored inside data nodes in multiple equal blocks (e.g., 128 MB), and NameNode keeps the information of those blocks and meta information. FileSystem reads and streams by accessing blocks sequentially. FileSystem initially gets blocks of information from NameNode and then opens, reads, and closes them one by one. It opens the first block, and once it complete, closes and opens the next block. The HDFS replicates the block to give a higher reliability and scalability, and if the client is one of the DataNodes, it tries to access the block locally. If it fails to locate the block on the same DataNode, it moves to other cluster data nodes.

FileSystem uses FSDataOutputStream and FSDataInputStream to write and read the contents. Hadoop has various implementations of FileSystem.

- **DistributedFileSystem:** To access the HDFS File in a distributed environment
- **LocalFileSystem:** To access the HDFS file on the local system
- **FTPFileSystem:** To access the HDFS file FTP client
- **WebHdfsFileSystem:** To access the HDFS file over the Web

URI AND PATH

Hadoop's URI locates file location in the HDFS. It uses the `hdfs://host:port/location` to access a file through FileSystem.

The below code shows how to create URI.

```
hdfs://localhost:9000/user/joe/TestFile.txt
URI uri=URI.create ("hdfs://host: port/path");
```

The host and port are defined in *core-site.xml*:

```
<property>
<name>fs.default.name</name>
<value>hdfs://localhost:9000</value>
</property>
```

Please refer to the Hadoop Setup section for more details.

A path that consists of URI with normalized OS dependency in URI, e.g., Windows uses \\path, whereas Linux uses //. It also resolves the parent-child dependency. It could be created as shown below:

```
Path path=new Path (uri); //It constitutes the URI
```

Configuration

The configuration class passes the Hadoop configuration information to FileSystem. It loads the core-site and *core-default.xml* through class loader and keeps Hadoop configuration information such as `fs.defaultFS` and `fs.default.name`. The configuration class can be created as below:

```
Configuration conf = new Configuration ();
```

You can also set the configuration parameter explicitly:

```
conf.set("fs.default.name", "hdfs://localhost:9000")
conf.set("fs.default.name", "hdfs://localhost:9000")
```

org.apache.hadoop.fs.FileSystem

The following code describes how to create Hadoop's FileSystem:

- `public static FileSystem get(Configuration conf)`
- `public static FileSystem get(URI uri, Configuration conf)`
- `public static FileSystem get(URI uri, Configuration conf, String user)`

FileSystem uses the NameNode to locate the DataNode and then directly accesses the DataNodes' blocks in sequential order. FileSystem uses the Java IO FileSystem interface, mainly DataInputStream and DataOutputStream, for IO operation.

To get a local filesystem, we can directly use the getLocal method:

```
public static LocalFileSystem getLocal(Configuration conf)
```

FSDATAINPUTSTREAM

`FSDataInputStream` wraps the `DataInputStream` and implements `Seekable`, a position-readable interface that provide methods like `getPos()` and `seek()` to provide Random Access on the HDFS file. FileSystem has the `open()` method, which returns the `FSDataInputStream`:

```
URI uri = URI.create ("hdfs://host: port/file path");
Configuration conf = new Configuration ();
FileSystem file = FileSystem.get (uri, conf);
FSDataInputStream in = file.open(new Path(uri));
```

In the above method, we get `FSDataInputStream` with a default buffer size of 4096 bytes, i.e., 4 KB. We can also define the buffer size while creating the input stream:

```
public abstract FSDataInputStream open(Path path, int
sizeBuffer)
```

`FSDataInputStream` implements the `seek(long pos)` and

`getPos()` method of a seekable interface.

```
public interface Seekable {
void seek(long pos) throws IOException;
long getPos() throws IOException;
   boolean seekToNewSource(long targetPos) throws
IOException;
}
```

The `seek()` method looks for the file to the given offset from the start of the file so that `read()` will stream from that location, whereas the `getPos()` method will return the current position on the `InputStream`. Below is the sample code that uses the `seek()`, `getPos()`, and `read()` methods.

```
FileSystem file = FileSystem.get (uri, conf);
FSDataInputStream in = file.open(new Path(uri));
byte[] btbuffer = new byte[5];
in.seek(5); // sent to 5th position
Assert.assertEquals(5, in.getPos());
in.read(btbuffer, 0, 5);//read 5byte in byte array from
offset 0
System.out.println(new String(btbuffer));//print 5
character from 5th position
in.read(10,btbuffer, 0, 5);// print 5 character staring
from 10th position
```

FSDataInputStream also implements PositionedReadable, which provides the read() and readFully() method to read part of the file's content from a position located by seek():

```
read(long position, byte[] buffer, int offset, int length)
```

FSDATAOUTPUTSTREAM

The FileSystem create() method returns FSDataOutputStream. It creates a new HDFS file or writes the content at the EOF. It does not provide a seek() method because of the HDFS limitation to write the EOF content only. It wraps Java IO's DataOutputStream and adds methods such as getPos() to get the file's position and write() to write the content at the last position.

```
public FSDataOutputStream create(Path f) //create empty
file.
public FSDataOutputStream append(Path f)//append existing
file
```

Note: Append might not support all HDFS implementations. The create() method also passes the Progressable interface to track the status during file creation.

```
public FSDataOutputStream create(Path f, Progressable
progress)
```

FILESTATUS

The following code shows how to use the getStatus() method of FileSystem, which provides the HDFS file's meta information:

```
URI uri=URI.create(strURI);
FileSystem fileSystem=FileSystem.get(uri,conf);
FileStatus fileStatus=fileSystem.getFileStatus(new
Path(uri));
System.out.println("AccessTime:"+fileStatus.
                                        getAccessTime());
System.out.println("AccessTime:"+fileStatus.getLen());
System.out.println("AccessTime:"+fileStatus.
                                    getModificationTime());
System.out.println("AccessTime:"+fileStatus.getPath());
```

If your URI is a directory and not a file, then `listStatus()` will give you array of `FileStatus[]` as below

```
public FileStatus[] listStatus(Path f)
```

DIRECTORIES

FileSystem provides the public boolean `mkdirs(Path f)` method to create directories and all the necessary child directories. It returns `true` if the directory is created successfully.

DELETE FILES

The `delete()` method on FileSystem removes a file or directory permanently.

```
public boolean delete (Path f, boolean recursive)
throws IOException
```

If `recursive` is `true`, then it will delete the non-empty directories.

C API LIBHDFS

`Libhdfs` is a C-based API used to access the HDFS file and the file system. `libhdfs` is precompiled and available with the Hadoop distribution. It is also compatible with Windows. The `libhdfs` APIs are a subset of the Hadoop FileSystem APIs and is thread-safe.

We can find details about the API in the following location:

```
$HADOOP_HDFS_HOME/include/hdfs.h.
```

CHAPTER 6

YET ANOTHER RESOURCE NEGOTIATOR

YARN is a resource-allocation-based framework that manages cluster resources and supports a wide variety of applications that can share the resources side by side and run parallel. The YARN is a core component of Hadoop. It interacts with computing resources on behalf of the application and allocates resources based on application needs.

YARN provides multi-purpose clusters to run different applications, such as data streaming, batch processing, and image processing on the same cluster at the same time.

YARN abstracts resources and shares them within a distributed application framework. It provides an ecosystem platform to support various application frameworks, such as Apache Tej, Apache Storm, Apache Hive, Apache Pig, and Apache Giraph.

Figure 6.1 depicts the high-level view of the YARN ecosystem.

Below are the key benefits of YARN:

Resource Utilization: YARN manages cluster resources and shares the resources across various applications, helping to better utilize resources across the cluster.

Scalability: YARN decouples the resource manager from the application manager, which gives it the capability to scale the number of clusters and scale many applications to run on the same clusters.

Multi-Tenancy: YARN uses a common standard platform to support multiple process engines for batches and streams, both interactive and real-time.

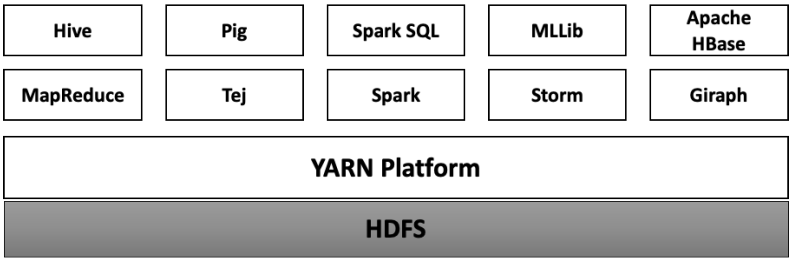

FIGURE 6.1 YARN ecosystems

YARN ARCHITECTURE

YARN decouples resource management from job execution management as a separate daemon. YARN uses the ResourceManager to manage resources and ApplicationManager to manage job execution. The loose coupling between the ResourceManager and ApplicationMaster enable Hadoop to scale to a much broader context. It also brings MapReduce and other programming models, such as graph processing, iterative processing, and machine learning, into the same platform.

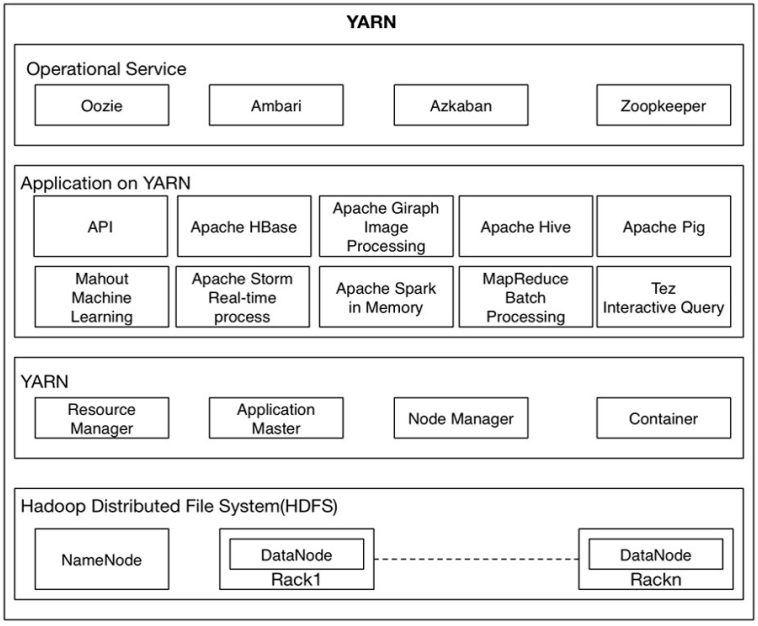

FIGURE 6.2 YARN's interaction with various components

Figure 6.2 shows how the YARN platform runs on top of the HDFS, interacting with various application frameworks, where each application is managed by a different ApplicationMaster that communicates with a ResourceManager to request the required resources.

YARN has four main components: ResourceManager, ApplicationMaster, NodeManager, and Container.

- **ResourceManager:** The ResourceManager manages the resource allocation and schedules the applications. It also tracks the status of individual applications. The ResourceManager is split into two main parts: the Scheduler and ApplicationManager. The Scheduler is responsible for allocating resources to the application. It has a pluggable policy (such as CapacityScheduler or FairScheduler) to define and apply the resource allocation. The ApplicationManager accepts the job, negotiating with the ApplicationMaster to execute the jobs and restart the job in case of a failure.
- **ApplicationMaster:** The ApplicationMaster executes and monitors the lifecycle of a task or application. It keeps sending the status through heartbeats to the ResourceManager at fixed intervals. It receives responses from the ResourceManager to receive a free container, notification on task failure, and slot/container termination.
- **NodeManager:** The NodeManager manages each node and tracks its health. The NodeManager manages containers representing per-node resources available for a particular application.
- **Container:** A container represents a unit of resources, including CPU, Memory, Heap, etc.

Figure 6.3 depicts how the YARN components (ResourceManager, ApplicationMaster, or NodeManager) communicate with each other.

The ResourceManager knows about all the resources (containers) on the cluster. The ApplicationMaster maintains the lifecycle of applications and communicates with the ResourceManager to obtain the resources. The ResourceManager allocates the resources based on a request raised by the ApplicationMaster. The ResourceManager takes the offer from the ApplicationMaster and allocates the resources once it finds free slots. The ResourceManager resource allocation happens asynchronously.

The NodeManager runs on each node and manages that node's slot. The ResourceManager communicates with the NodeManager to allocate slots for the ApplicationMaster to execute the task. The ApplicationMaster gets the slots from the NodeManager through the ResourceManager to execute its task and manage its lifecycle.

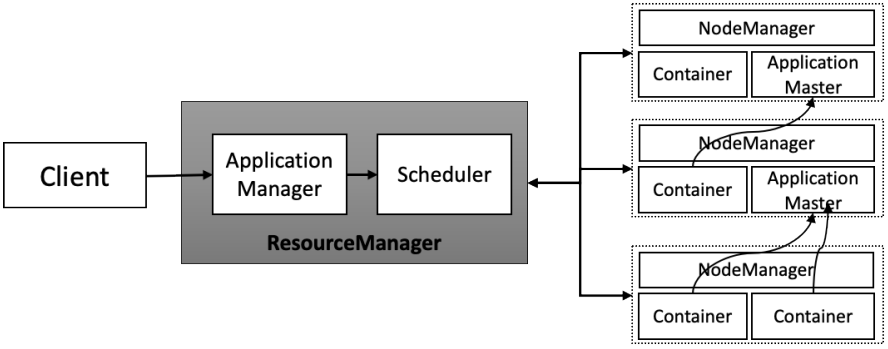

FIGURE 6.3 YARN ResourceManager

The ResourceManager manages the failure task and communicates with the ApplicationMaster using the next immediate heartbeat.

The ResourceManager notifies the ApplicationMaster before flushing the resource so that the ApplicationMaster verifies its running task. An ApplicationMaster manages task instances and resources from the ResourceManager. It monitors the execution and resource consumption of containers (such as the resource allocation of the CPU and memory).

YARN PROCESS FLOW

The YARN client connects to the ResourceManager and defines the local resources and environment configuration required for the ApplicationMaster.

The YARN client submits the request to start the ApplicationMaster. ApplicationMaster calculates the required slots and sends the request to the ResourceManager. The ResourceManager communicates with the NodeManager to receive the available nodes and assign the available slots back to the ApplicationMaster. The ApplicationMaster keeps sending heartbeats to ResourceManager, so ResourceManager is aware of its health.

Once the ApplicationMaster receives the available slots from the ResourceManager, it sends a request to the NodeManager to start the task. The NodeManager copies the application jar from the local filesystem to the HDSF (distributed file system). The NodeManager starts the task upon getting the request from the ApplicationMaster and keeps tracking the status of the query. The NodeManager informs the ResourceManager about the task's completion or failure. The ResourceManager responds to the ApplicationMaster on the next heartbeat response.

Figure 6.4 depicts how the application task executes in the YARN framework.

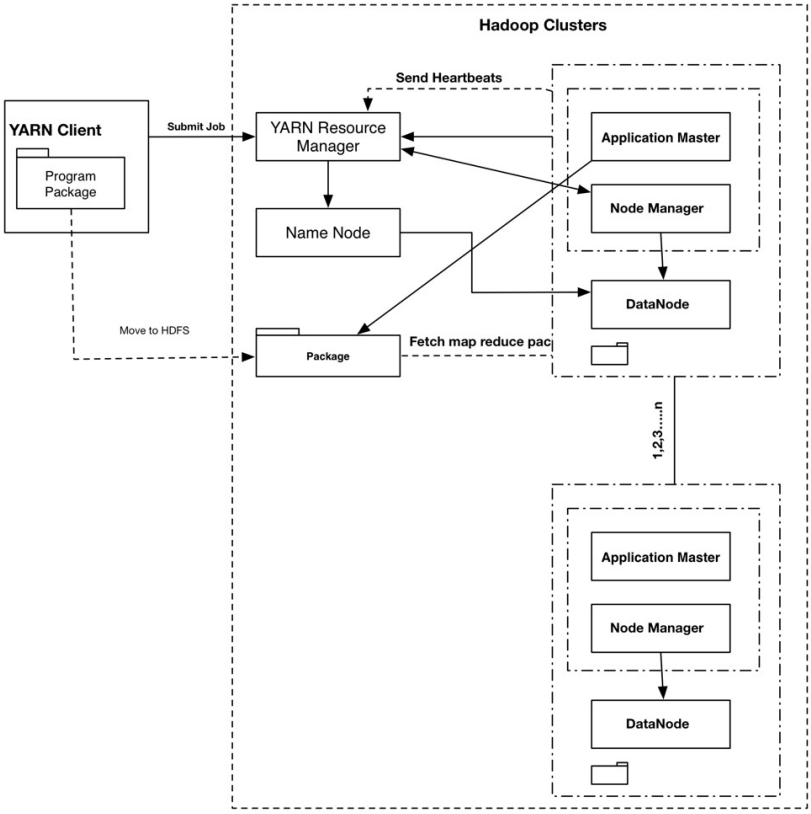

FIGURE 6.4 YARN process flow

The ApplicationMaster sends slot requests one time to ResourceManager. However, the ApplicationMaster can send the request again if more slots are required. The ResourceManager adds those slots and reassigns them to the ApplicationMaster. The ApplicationMaster releases the slots once a task gets completed or fails. The ResourceManager assigns the slot to ApplicationMaster asynchronously.

We can summarize the overall process as below:

- The ResourceManager receives a request from the client and allocates the required resources.
- The ResourceManager instantiates the ApplicationMaster.
- The ApplicationMaster invokes the resource containers for the application and executes the application.
- The ApplicationMaster releases its containers with the ResourceManager.

YARN FAILURES

Container Failure

The NodeManager handles the container failures. When a container fails, the NodeManager identifies the failure container and launches a new container. The NodeManager restarts the task execution in the new container.

Task Failure

The NodeManager handles container and task failures. When containers fail or die, or if a task fails, the NodeManager informs the ResourceManager, and the ResourceManager reports to the ApplicationMaster on the next heartbeat response. The ApplicationMaster validates its job and releases the slots. Since the failed task already has some data processed and may contain valuable information, the ResourceManager tells the ApplicationMaster to verify the task before releasing the slots. The ApplicationMaster re-initiates the failed tasks on the other slots.

ApplicationMaster Failure

In the case of an ApplicationMaster failure, the ResourceManager identifies the failed ApplicationMaster and starts a new instance of the ApplicationMaster with new container resources.

ResourceManager Failure

In earlier versions of Hadoop, YARN had a single point of failure, whereases after Hadoop 2.4, it introduced high availability using ZooKeeper to handle the ResourceManager at a single point of failure.

YARN HIGH AVAILABILITY

The ResourceManager is the single point of failure. That means in case of the ResourceManager failure, YARN won't respond, and the server needs to be restarted. This restart kills all running pipelines and loses all historical states. It is crucial to implement a high availability on the ResourceManager to prevent YARN failure.

The YARN high availability features are introduced in the latest Hadoop (Hadoop 2.4+) to solve a single point of failure. The YARN high availability

features use the Active/Standby redundant ResourceManager controlled by ZooKeeper.

Figure 6.5 depicts Active/Standby high availability architecture.

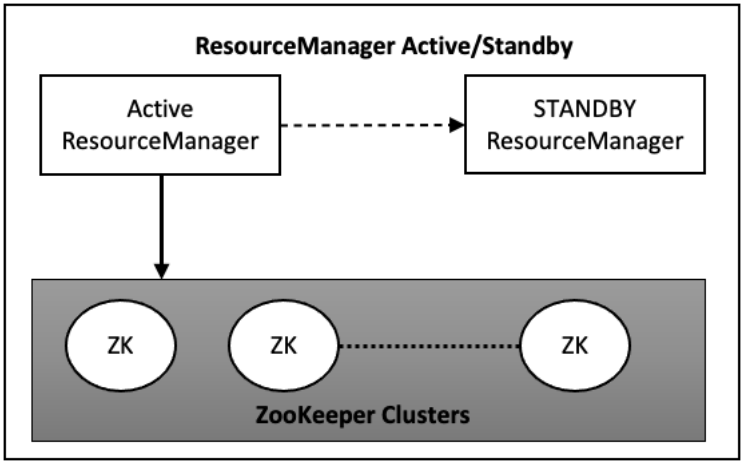

FIGURE 6.5 Zookeeper high availability (active/standby)

Figure 6.5 shows the YARN Active ResourceManager has one or more Standby ResourceManagers. These Standby ResourceManagers keep in sync with their states with the Active ResourceManager. The Standby ResourceManager takes the responsibility and switches to the Active ResourceManager in case the current/active ResourceManager fails.

There are two ways to switch from the Standby ResourceManager to the Active ResourceManager: the manual failover and automatic failover.

Manual Failover

In this case, the administrator can manually switch the ResourceManager from the Standby to Active state. The administrator uses the CLI command `yarn rmadmin` to transition to the Active or Standby ResourceManager.

Automatic Failover

The automatic failover uses the ZooKeeper to decide the Active/Standby ResourceManager selection. When the Active ResourceManager fails to respond or is unresponsive, the ZooKeeper automatically elects another standby ResourceManager to lead as the active resource manager.

A cluster uses multiple ResourceManagers to achieve high availability, where there is one the active ResourceManager and the rest are standby ResourceManagers. It is essential to keep the active ResourceManager to sync up its state with other standby ResourceManagers.

A cluster configured with a high availability has multiple ResourceManager services running; only one of them is active at a time, and the rest are in standby mode. The clients always connect to the active ResourceManager service. The ResourceManager provides a file-based and ZooKeeper-based state store implementation. A file-based state store mechanism stores the state into a shared file where the ZooKeeper uses the ZooKeeper quorum to store the ResourceManager state. The ZooKeeper allows only a single ResourceManager to remain in the persist state for the Zookeeper quorum. In the case when the active ResourceManager is unresponsive, the Zookeeper uses the ActiveStandbyElector to elect new a ResourceManager to become the active ResourceManager.

YARN SCHEDULERS

The YARN's ResourceManager consists of two main components: the ApplicationManager and Scheduler. The ApplicationManager accepts requested jobs from the client and validates the requested jobs, ensuring that no other application initiates the same job or that there are enough resources to execute the jobs. The ApplicationManager forwards the validated requested job to the Scheduler to proceed further. The ApplicationManager also keeps the summary status and logs of jobs even after job completion and provides a summary.

The Scheduler is responsible for allocating resources to the application-based pluggable scheduler policy (CapacityScheduler, FairScheduler). The ResourceManager's Scheduler is a pure scheduler, and its only duty is to apply the Scheduler policy and allocate the resource to the application. It doesn't perform monitoring or status tracking. The Scheduler receives application submission requests with resource requirement details and applies a pluggable scheduling policy to allocate resources to execute the application. YARN scheduling's primary goal is to provide a fair chance to complete the job without getting stuck with a single application.

THE FAIR SCHEDULER

The Fair Scheduler solves the problem of resource exploitation that occurs in the default Hadoop scheduler. The Fair Scheduler provides an equal chance to complete the entire job over time by providing similar resources.

The Fair Scheduler selects jobs and places them into a pool. Each pool has an equal number of resources, which provide the minimum required map slots, and the resource slots to complete the job. That means the entire job running in a cluster environment get some slots to complete the jobs.

A fair-sharing Scheduler ensures that each job gets the same number or a minimum of slots. A Fair Scheduler ensures that longer jobs do not consume all the resources and keep sharing the resources so that shorter jobs can be completed.

Note: Along with fair sharing, it also possible to define the app/pool priority to guarantee each application/pool gets some part of the total resources.

Each pool also uses a scheduling policy to share resources on the particular pool across the running applications. The default is the memory, however, we can also set FIFO or Dominant Resource Fairness (multi-resource).

Fair scheduling can also allow for assigning some fixed minimum resources to pool that guarantee a specific pool should get at least some resources. In that case, the Scheduler only allocates resources if the pool has some application to process; otherwise, it will split its resource part to another pool that has applications. This approach optimizes the overall resource utilization when the pool doesn't have any application to process. In general, a Fair Scheduler allows all applications to execute; however, we can limit the number of applications per user and per pool. If the limit exceeds the rest of the applications, it waits until applications in the queue get completed.

The following is an example that shows how the Fair Scheduler works.

Scenario

Let us assume we have one hundred total slots to share across the pool, and we have three pools: A, B, and C.

The Fair Scheduler allocates the resource based on demand. For example, if a pool does not ask for a slot, it would not distribute any slot.

If A asks for ten slots, B asks for fifty slots, and C asks for 60 slots, the Scheduler will calculate the slots and distribute equally or according to the minimum number of slots needed. In this case, each queue is supposed to get an equal number of slots, whereas A needs only twenty slots, so the Scheduler will allocate only twenty slots, and the rest of the slots get equally distributed to the B and C pools.

Table 6.1 Fair Scheduler

Pool	Resource Demanded	Resource Allocated
A	10	10
B	50	45
C	60	45

The Fair Scheduler tries to distribute the shared resources equally; however, it optimizes by not allocating any extra slots that are not required for a queue and get utilized by other pools.

If, at the same time, Queue C demands thirty slots, the Scheduler will reallocate the resources across the pools:

Table 6.2 Fair Scheduler

Pool	Resource Demanded	Resource Allocated
A	10	10
B	50	30
C	60	30
D	110	30

THE CAPACITY SCHEDULER

The Capacity Scheduler optimizes cluster resources by sharing resources across the organization. It advocates bringing all cluster resources together as a massive Hadoop Cluster and allocating specific resources to each organization or department. It minimizes low resource utilization and reduces the extra load needed to manage multiple Hadoop clusters.

The Capacity Scheduler gives a guarantee to each department to allocate some resources. The key benefit is to move unused resources from one department to others that need them, but at the same time, it restricts the limit so that a single application/queue doesn't consume a significant chunk of resources.

The Capacity Scheduler uses concepts of the queue similar to that of the Fair Scheduler's pools that have allocated capacity. These queues are hierarchical, where inside each queue application there is priority scheduling.

The Capacity Scheduler uses queues instead of the pools on the Fair Scheduler. Each queue has a configurable, limited number of slots. The jobs are assigned to a queue and then prioritized within a queue that is similar to FIFO.

Each queue gets some share of the resource slots. The Capacity Scheduler keeps monitoring each queue and readjusts the allocated shares, e.g., if any queue is not using its allocated slots, then the remaining slots are reallocated to another queue.

Users can submit jobs to multiple queues using CapacityScheduler. Initially, the slots provide the queues in such a manner that more running job queues will be allocated first. If a Scheduler finds that any queue does not have more tasks demanded during the heartbeat, it will release the extra unused slots to the other queue temporarily.

For example, if one user, A, executes the first job in the cluster, then all the resources are allocated to a particular queue. If another user, B, executes another job, both nodes have an equal number of shares to get allocated (meaning 50% each).

Similarly, if we add two more users to execute a job, say users C and D, each user gets a 25% share of the resources. Please note that there is a threshold not to include so many users, and the Scheduler would wait to complete an already executing job, which prevents the depletion of resources for all users.

Below are the key features for the Capacity Scheduler:

- **Multi-Tenancy:** The Capacity Scheduler shares and serves resources across the organization, where each organization must be guaranteed some capacity. This approach serves all organizations, groups, or applications to allocate the resource capacity and avoid monopolizing the cluster's resources.
- **Elasticity:** The Capacity Scheduler optimizes resources by reallocating unused resources from one queue to another queue. These enhance the elastic nature of the Capacity Scheduler.
- **Security:** The Capacity Scheduler allows strict ACSs to control applications to submit to a specific queue. It also protects applications that can't access(view/modify) cross-queue applications.
- **Capacity Guarantees:** The Capacity Scheduler allocates some capacity to each queue, which is configurable. Each application associated with the queue has full access to resources allocated to the queue.
- **Resource Scheduling:** Each organization has different resource requirements; hence, the Capacity Scheduler can configure different resource limits to each organization's specific queue.

- **Priority Scheduling:** The Capacity Scheduler also offers priority scheduling in the queue, allowing applications to schedule with different resource priorities.

THE YARN TIMELINE SERVER

Older versions of Hadoop provide a job history to track the finished MapReduce jobs. YARN supports many applications other than MapReduce, such as Tez and Spark. The job history stores all its tracking logs on the HDFS itself, but it allows for completed or historical MR jobs only. It does not capture YARN-level metrics and events. If the ApplicationMaster failed/crashed, it loses all current running applications' status. There is not much scope on the scalability and reliability in Hadoop's job history.

YARN comes with the Application History Server, which can run in parallel with the ResourceManager. The ResourceManager directly writes jobs' activity to the HDFS, and the Application History Server provides an interface to access the completed job's status. The Application History Server runs independently with YARN Resources and provides the RestUI, CLI, and separate UI to track completed jobs. It supports all applications that can run on YARN's ResourceManager.

The Application History Server is far better than the legacy JobHistory. However, it has limitations, such as it can only track completed jobs, and when it writes to the HDFS, a failure from the ApplicationMaster can lose all the statuses of the running applications. This is a challenge for its scalability and reliability.

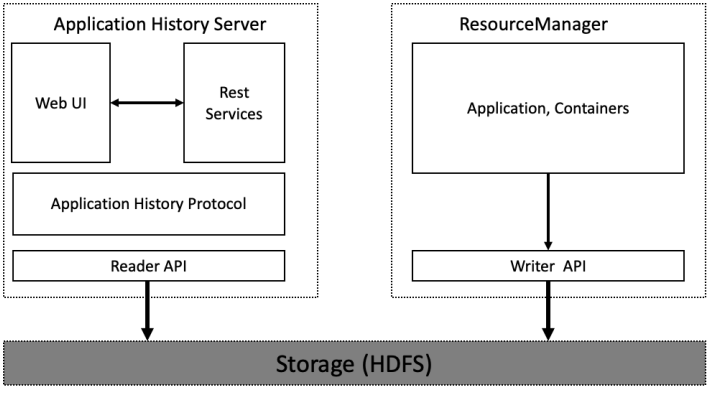

FIGURE 6.6 Application History Server

APPLICATION TIMELINE SERVER (ATS)

The new architecture Application Timeline Server (ATS) addresses these challenges. It introduces the flow concept, which is a series of jobs acting as one entity rather than a set of separate jobs (MapReduce).

The Timeline Server enables us to collect YARN applications' information via the TimelineClient and stores it into a time store. The Timeline Server maintains a historical state and provides metric visibility for the YARN applications. We can collect information about completed applications using the Web UI or via REST APIs.

The Timeline Server keeps the following information:

1. Information about completed jobs, such as queue name, user information, the application attempts information, container lists, executed application, and container information
2. Information about a running or completed application or framework

The Timeline server is an independent lightweight standalone server daemon deployed on the cluster. It may or may not co-locate with the ResourceManager.

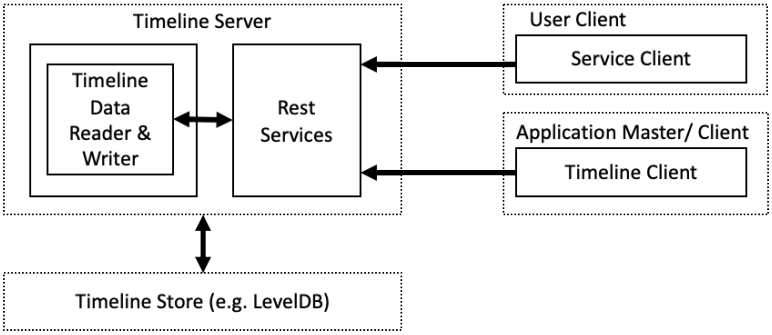

FIGURE 6.7 ATS V1 architecture

ATS DATA MODEL STRUCTURE

Timeline Domain

The Timeline Domain uses a namespace with a uniquely identified ID to host multiple entities. Each entity from the user's application is isolated from the other. The Timeline Domain provides security at this level. It includes the user information, created and modified times, and the ACL information.

Timeline Entity

A Timeline Entity contains the meta details for a conceptual entity such as an application, application attempts, a container, and a user-defined object. EntityId and EntityType uniquely identify an entity.

Timeline entities consist of a primary filter for indexing entities in the Timeline Store.

Timeline Events

A Timeline event is event information that relates to an entity of an application. We can define events, such as starting an application, operation failure, and getting allocated a container.

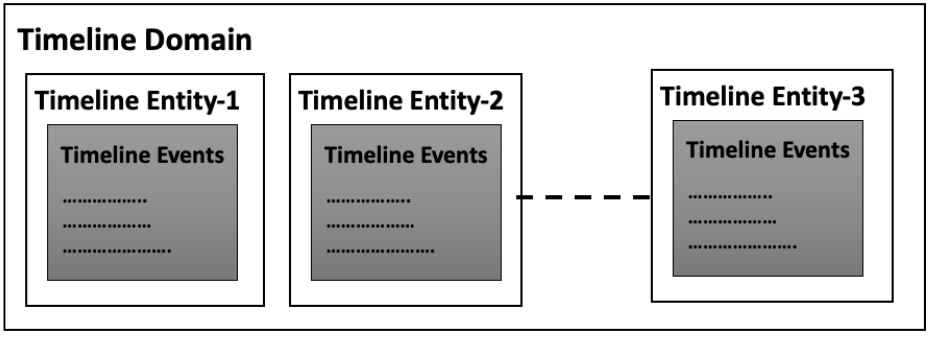

FIGURE 6.8 ATS data model

ATS V2

The Application Timeline Server (ATS V1) addresses many challenges, such as a separate, independent daemon for ATS, defined process flow, and tracking many more framework applications other than MapReduce. However, we still face challenges in regards to the scalability and user interface. ATS V1 uses a single global instance where the reader/writer could not adjust to the distributed environment. It also manages local disk-based levelDB storage, which is hard to scale. ATS V2 has solved many challenges faced by ATS V1, as discussed in the next section.

Scalability Improvements

The YARN Timeline Service v.2 keeps multiple, loosely coupled timeline collectors (writers) to write data into ATS storage. They run in parallel in a distributed environment (one collector per application) to store into the default Apache HBase. Apache HBase has a high scalability and can read/write event data. Timeline readers are a pool of reading events that occur using the REST API. The readers are separate instances that serve queries via the REST API.

Usability Improvements

YARN ATS V2 provides a logical group of yarn applications as a unit operation. It also supports configuration information and metrics.

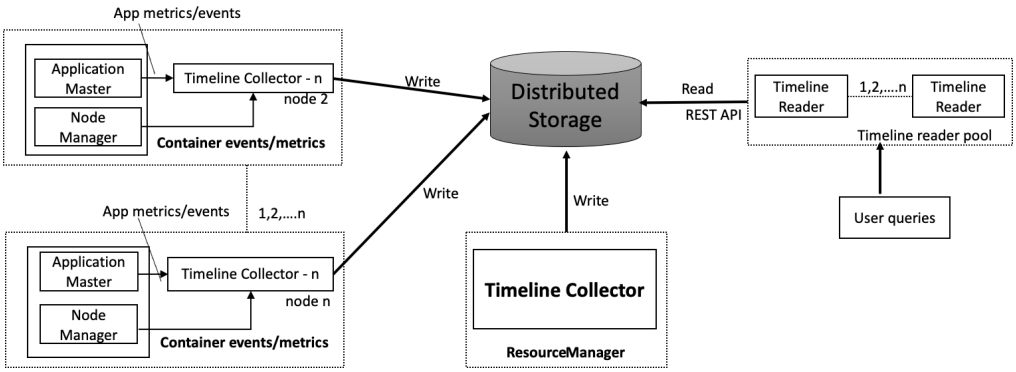

FIGURE 6.9 ATS V2 Architecture

Figure 6.9 shows that YARN ATS V2 uses a distributed Timeline collector to write data on the distributed backend storage. Each application is distributed on different worker nodes, where it collects events and metrics from the dedicated ApplicationMaster and Node manager, respectively, and sends them to the backend storage. The resource manager also keeps its own Timeline Collector and writes YARN-specific generic lifecycle events to backend distributed storage.

The Timeline reader is deployed on a separate daemon from the timeline collector and dedicated to serving queries via the REST API. ATS V2 supports a robust query interface for information related to the flow, flow runs, and apps. It also supports complex queries to filter out the results.

YARN FEDERATION

The YARN ResourceManager is a central point of connection to control and manage Hadoop clusters. A sizeable growing organization always has challenges getting scalability to work through a single YARN ResourceManager. A single YARN ResourceManager can easily manage many thousands of nodes; however, a large growing company where the number of nodes keeps growing may quickly hit the threshold limit. YARN Federation architecture solves growing YARN infrastructure by scaling YARN from a single node to many nodes. This architecture subdivides one big YARN cluster to many YARN sub-clusters, each with its YARN ResourceManager and NodeManager. The YARN Federation abstracts all YARN sub-clusters and provides a large cluster where each YARN sub-clusters manages their responsibilities. The YARN Federation system routes the submitted application to one of the federated clusters to schedule the application. Federated clusters coordinate with each other through the common shared state store.

The YARN Federation is an architecture where federated YARN sub-clusters work together through a Federation-provided single abstracted large YARN cluster. Figure 6.10 shows how different YARN Federation components work.

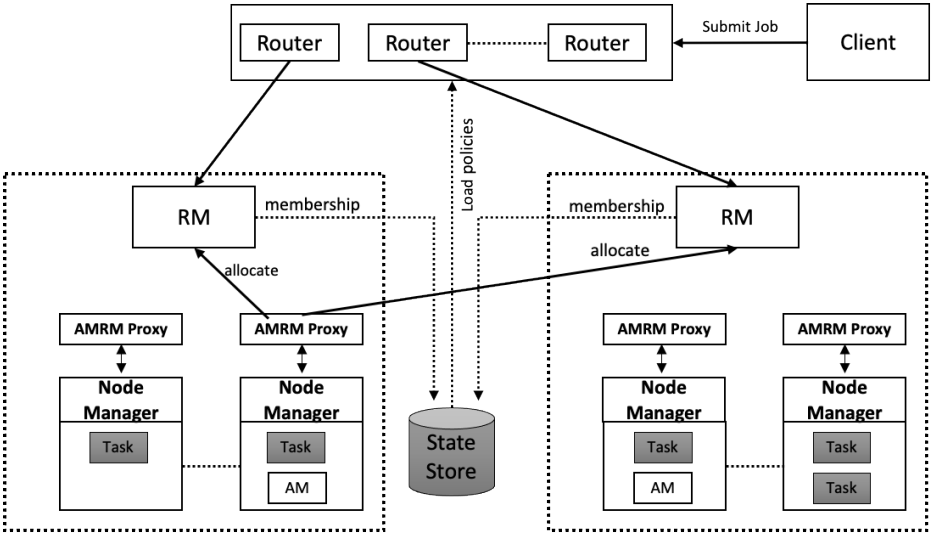

FIGURE 6.10 YARN Federation

YARN Sub-clusters

The federated YARN sub-clusters have fully functional ResourceManagers that handle sub-sets of requests. The federated YARN architecture is a highly available, scalable, and fault-tolerant system. If any sub-cluster fails, federated YARN would ensure that the jobs get processed by other federated sub-clusters. We can quickly scale up by adding more federated sub-clusters.

Router

A YARN cluster consists of a collection of routers, which abstract the federated YARN sub-cluster and its ResourceManager. The client submits its application to a router. The router applies the policy to determine the sub-cluster and submit it to the selected sub-cluster. The router uses the state store to get the sub-clusters' details and its ResourceManager and submit a job to the chosen sub-cluster's ResourceManager.

AMRM Proxy

The AMRM Proxy is a service that acts as a proxy for an ApplicationMaster between the ResourceManager and NodeManager. The ResourceManager cannot directly communicate with the NodeManager. The ResourceManager accesses the NodeManager locally through the AMRM proxy. The AMRM proxy provides dynamic routine access to multiple YARN federated ResourceManagers.

CHAPTER 7

MAPREDUCE

MapReduce is a framework on Hadoop for processing a massive amount of data in-parallel in a distributed environment. It's a reliable and fault-tolerant process running on common hardware, which makes it cost-effective.

The HDFS stores files in identical, fixed-sized blocks (such as 64 MB or 128 MB). The MapReduce framework accesses and processes these blocks in a distributed parallel environment. MapReduce spreads the jobs across these fixed-size blocks to execute in parallel and later aggregate the output to write into external storage, such as the HDFS or AWS S3. The MapReduce framework works on key-value pairs; it has two main parts, the Mapper and Reducer. Each mapper task uses a small data block as the key-value (<key, value>) pairs, and passes the input to the reducer, and the reducer processes a subset of the map task output, aggregates the data, and writes into the storage, e.g., the HDFS.

MAPREDUCE PROCESS

The MapReduce framework splits data into a key-value pair, which is passed as <key, value> in different phases and produces a different set (modified) of the key-value pair as the output. Since different stages of the key-value data are written into storage, it is serialized by implementing Writable.

```
(input) <k1, v1> -> map -> <k2, v2> -> combine ->
<k2, v2> -> reduce -> <k3, v3> (output)
```

MapReduce tasks go through different phases before writing the final output to the HDFS. The input file first passes to the Mapper task, converting

these input into key-value pairs using InputFormat. Sorting and shuffling allow the opportunity to decide which Reducer has to pass the map input collection, i.e., the key-value pairs.

In theory, the Reducer starts processing once the entire Mapper task is completed; however, sorting provides a way to start processing the Reducer while the Mapper is still processing it. The Reducer aggregates the input, processes it, and then writes into the HDFS file.

It's also possible to have zero Reducer tasks. A *zero reducer* means that the jobs run without shuffling and processing entirely in parallel.

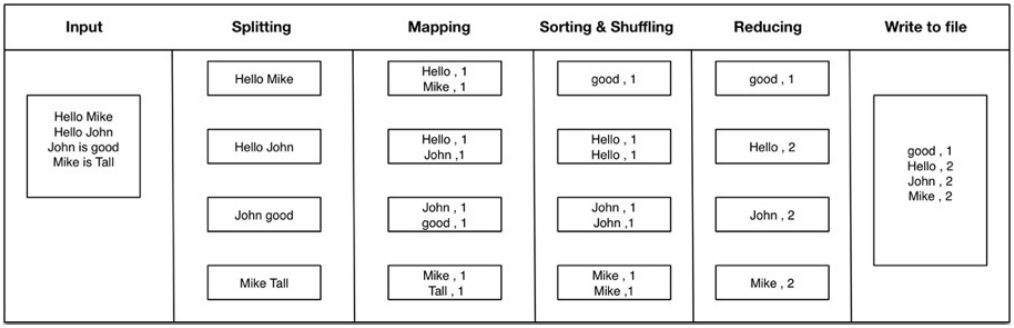

FIGURE 7.1 MapReduce process flow

Figure 7.1 shows the first phase input gets split into four Mappers that convert these input lines into key-value pairs. Then it gets sorted by key and grouped with the same key. Now we have four Reducers, which collected the same key's value and aggregated those values by counting them. Finally, this information gets written into the HDFS file.

The Reducers processes are completed in multiple phases. For example, if we have eight reducers, then in the first phase, we aggregate two sets of four reducers that will give two outputs, which in turn, aggregate to one Reducer to get written into one HDFS file.

There might be a possibility that the last Reducer is not the only Reducer, which means that in certain cases, the Reducer gets minimized and directly writes to the HDFS from multiple files.

MapReduce sorts and shuffles output data from the Mapper before passing it to the Reducers. The Mapper splits the data into key-value pairs and saves the information into a local intermediate disk. MapReduce sorts these values based on the keys and shuffles them to ensure all the same keys go to the same Reducer.

Sorting mechanisms minimize the Reducer's effort and enables the parallelism to start the Reducers before completing them. For instance, once

K1-V1 is completed and K2-V2's value is picked up, MapReduce is aware that no more K1-V1 exists in the input data and immediately instantiates the Reducer for K1-V1.

KEY FEATURES

The following are the key MapReduce features.

Data Locality

Map tasks process the input data and keep the output as an intermediate result, which will be input into the Reducer. After job completion, the Mapper output gets flushed. The Mapper task writes the intermediate result to the local disk, not in the HDFS, so the temporary data is not overhead on the HDFS. The other advantage is that if the Mapper's task fails before the Reducer's task begins, the Mapper task is re-run on the next DataNode.

Generally, Reducer tasks do not have data locality—the input to a single Reducer task is usually the output from all Mappers.

Parallel Processing

MapReduce enables parallel processing in a distributed environment. MapReduce spawns multiple Mapper tasks that access individual blocks to be processed independently. These Mapper tasks are executed in parallel and store the local disk's output that will be the input for the Reducer tasks.

Loose Coupling

MapReduce has Mappers and Reducers that run independently of each other. A Mapper task runs locally on the same data node; hence, it eliminates the network bandwidth and improves the stability and performance. Mapper and Reducer tasks run across the network on different nodes and are consolidated on the HDFS disk.

Scalability

MapReduce tasks (mapper and reducer tasks) don't share state and run parallel across the cluster's different nodes. Since there is no dependency between them, we can increase/decrease the hardware disk space with minimal changes.

Fault Tolerance

The ApplicationMaster keeps track of individual map tasks. If a map task is unresponsive or fails, the ApplicationMaster cleans the container for other tasks and reschedules the failed task to another NodeManager. Since there is no dependency between the tasks, it can be re-run without any impact.

Easy and Cost-Effective

MapReduce is an open-source framework and runs on common hardware. It doesn't support RAID, so it does not require expensive transaction-specific data storage.

DIFFERENT PHASES IN THE MAPREDUCE PROCESS

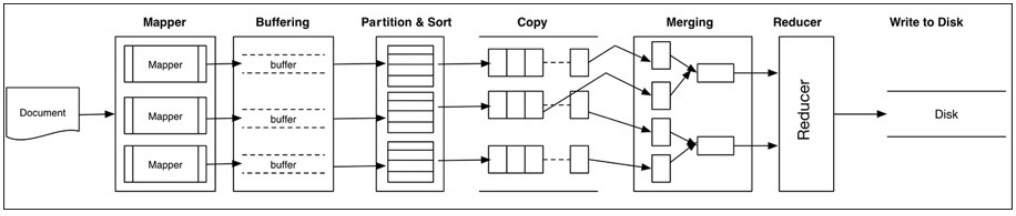

FIGURE 7.2 MapReduce data flow

Figure 7.2 shows how the data flows from Mapper to Reducer in different phases.

Mapper: Mapper Keeps input as key-value pairs and transforms these pairs into intermediate key-value output pairs. The output pair is optional, and it does not need to be the same type. That means any Mapper can generate zero or more key-value output pairs.

The Hadoop MapReduce framework generates map tasks for each input split for splittable InputFormat files. The map task applies a defined process to each key-value pair, and performs sorting and aggregation on the result. In the case when the Combiner is included, it is aggregated further using the defined Combiner.

A Combiner is applied to the map task, consolidates the data further, and transfers it to the Reducer. The Combiner consolidates and reduces the input data by using extra filters and aggregation.

Below is a simple Java program for the Mapper:

```
public class MessageCounterMapper
             extends Mapper<Object, Text, Text, IntWritable> {
   private final static IntWritable count = new
                                                IntWritable(1);
   private Text  message = new Text();
   public void map(Object key, Text value, Context
                                                context) throws
             IOException, InterruptedException {
      StringTokenizer it = new StringTokenizer(value.
                                                   toString());
      while (it.hasMoreTokens()) {
        word.set(it.nextToken());
        context.write(message, count);
     }
   }
}
```

The Mapper calls `map(WritableComparable, Writable, Context)` for each key-value pair. As shown above, the sample code map method accepts <key, value> as <Object, Text>, where Text implements WritableComparable to serialize, deserialize, and compare texts at the byte level. After processing out the key-value pair, the data is collected using `context.write (WritableComparable, Writable)`.

The intermediate output data from the Mapper is sorted, shuffled, aggregated, and then passed to the Reducer. The Reducer uses multiple Reducer tasks to accept the key-value pair from the Mapper. The number of Reducer tasks depends on the number of pieces of partitioned data output after the grouping is performed by the Mapper. We can apply various compression mechanisms to the intermediate data by configuring CompressionCodec.

Note: The total number of map tasks depends upon the number of a splits in the input file. Let's say an input file size is 1 GB and has blocks sized 128 MB that (around eight blocks); hence, it uses a maximum of eight map tasks to process it. In many scenarios, the input data is large, which will end up as a large number of map tasks and will degrade the performance. For example, if the data size is 1280 GB and the block size is the default of 128 MB, we will have 10,000 map tasks. We can control the maximum number of map tasks by using the configuration parameter to control the map task number `Configuration.set (MRJobConfig.NUM_MAPS, int)`.

Buffering Phase

The Mapper output goes to the buffer memory that uses the `mapreduce.task.io.sort.mb` parameter to estimate how much memory is allocated for the buffer memory. After a specific limit to fill on the buffer memory, it starts storing data into a local directory (defined by `mapreduce.cluster.local.dir`).

Sorting Phase

Before copying to a local directory, the data is grouped for a particular Reducer and sorted into individual groups. Sorting not only happens during partitioning and buffering, but also while merging to maintain the sorting order. Partitioned data are already separated; therefore, it is easy to keep the sorting order during the merging phase.

Copy Phase

The copy phase copies the Mapper output into the local disk and then moves these split data blocks to a Reducer. The reducer will receive the data files from various Mappers for further processing.

After the Mapper task completion, the Reducer starts copying the specific output from the local disk. The Reducer starts copying output data in parallel before completing all the Mapper tasks. Reducer tasks copyMapper output using multiple parallel threads to reduce the copy time. While copying the data first, it copies it to JVM in memory; once the JVM memory reaches the threshold, it starts copying the data to a disk.

Merging

First, the entire map output data is copied to the disk, and then the Reducer starts merging files into big files while maintaining their order. Reducer merges more than one file (configured by `mapreduce.task.io.sort.factor`) simultaneously and does so until it reaches a few big files.

For instance, if there are 40 files and `mapreduce.task.io.sort.factor` is 10 (the default), then in the first phase, it merges the 10 files and gives four intermediate joined sorted files, which directly move to the Reducer as input data.

Reducer

The Reducer takes input from the Mapper as key-value pairs, and transforms and aggregates the data into the final output. The Reducer spawns

parallel Reducer tasks, which are determined by the input partition split by Mapper. We can also configure the Reducer task number by setting `Job.setNumReduceTasks(int)`.

The code below includes some simple map tasks:

```
public class CountTotalReducer<Key> extends
Reducer<Key,IntWritable,
Key,IntWritable> {
   private IntWritable total = new IntWritable();
   public void reduce(Key key, Iterable<IntWritable> values,
                      Context context) throws IOException,
                                              InterruptedException {
     int intsum = 0;
     for (IntWritable val : values) {
       intsum += val.get();
     }
     result.set(intsum);
     context.write(key, total);
   }
 }
```

The Reducer call method `reduce(WritableComparable, Iterable<Writable>, Context)` for the collection of Writable values is passed to the same group partition. After processing, it calls Context with the final key-value pair to pass to the next phase. We can also have a scenario when no Reducer is required; in that case, the map will directly store the data into the HDFS FileSystem.

MAPREDUCE ARCHITECTURE

Hadoop uses YARN (Yet Another Resource Negotiator) as the resource management system to optimize and share the resources across distributed processing frameworks. MapReduce was the native batch processing distributed framework used on the legacy MR1 framework. The latest version of Hadoop uses YARN, and hence it replaced the MapReduce framework with the new YARN framework.

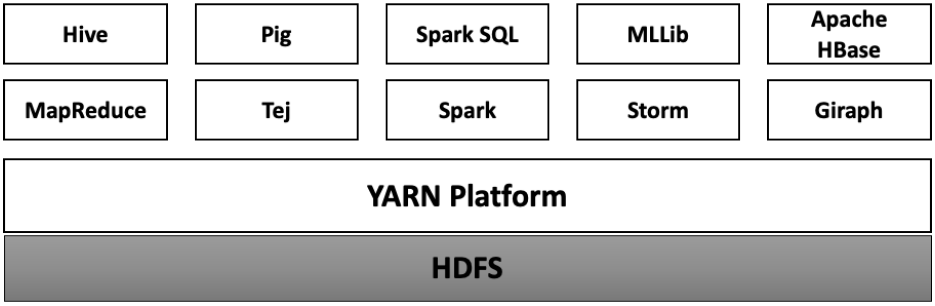

FIGURE 7.3 MapReduce framework

Figure 7.3 shows MapReduce is one application framework running on top of YARN with other distributed frameworks. YARN is abstracted on top of MapReduce to provide resource management and, at the same time, support other distributed processing such as Spark, Tej, and Giraph. Figure 7.4 depicts how various YARN components interact with each other.

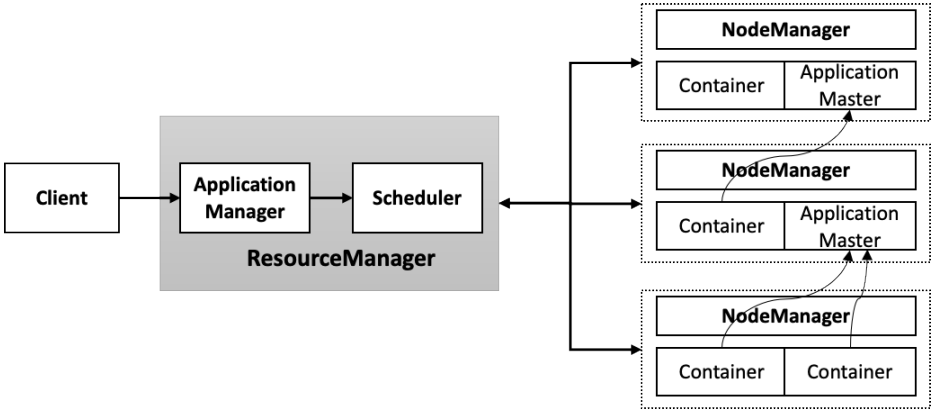

FIGURE 7.4 YARN - ResourceManager

The YARN consists of the ResourceManager and NodeManager, where the ResourceManager manages and allocates resources across the application and where the NodeManager manages resources on a single compute node. The ResourceManager consists of the ApplicationManager and Scheduler components. The Scheduler component receives the client requests, applies the Scheduler's plugin policy, and allocates resources.

YARN uses containers representing a unit of resources allocated by the ResourceManager consisting of a certain amount of CPU space and Memory

on a node. The ApplicationMaster launches processes on the Container node, coordinating the computation for a single process/task.

The client submits the application request to YARN, and the ResourceManager allocates the resources for a particular claim and launches the ApplicationMaster inside the container for that application. The ApplicationMaster takes all responsibility for the application's execution and its lifecycle. The ApplicationMaster coordinates with the NodeManager to manage the container and its task, which loosely couples the application execution to the ResourceManager. Figure 7.5 shows how YARN components interact with the MapReduce application.

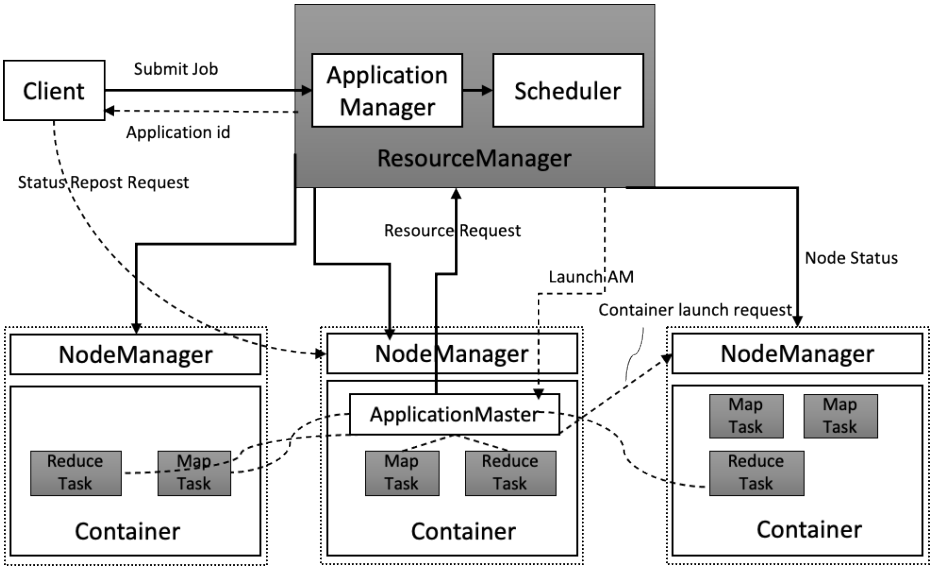

FIGURE 7.5 YARN in MapReduce process

The MapReduce application submits a job to the ResourceManager's ApplicationManager. The ResourceManager launches the ApplicationMaster on one of the nodes and sends the Application ID as an acknowledgment to the Client. The Scheduler of the ResourceManager allocates the resources and chooses the NodeManager for the ApplicationMaster. The Client can communicate directly with the ApplicationMaster to get the details about the ApplicationMaster and running tasks.

The ApplicationMaster coordinates with the ResourceManager and NodeManager for resource requirements and job execution. The ApplicationMaster computes the resource requirements (memory and CPU) for the

Mapper and Reducer to the Scheduler. The Scheduler uses the Scheduler plugin to calculate the required resources and passes it to the ApplicationMaster. The ApplicationMaster maintains and manages the launched Mapper and Reducer tasks and coordinates with the Scheduler for the resource requirements. The ApplicationMaster also handles task failure. If the task failed, the ApplicationMaster re-launches the failed task to the new NodeManager. Once the MapReduce process is complete, the ApplicationMaster sends a request to the ResourceManager to free up the occupied container resources.

MAPREDUCE SAMPLE PROGRAM

This section demonstrates how to write a basic program and how it works.

Word Count Program

In the following code, the Mapper reads the line and sets the word as a key with count 1 as the value.

```
public static class SampleMapper extends MapReduceBase
implements
        Mapper<LongWritable, Text, Text, IntWritable>
{
private final static IntWritable count = new
IntWritable(1);
private Text text = new Text();
public void map(LongWritable key, Text value,
            OutputCollector<Text, IntWritable> output,
                                        Reporter reporter)
            throws IOException {
        String line = value.toString();
        StringTokenizer tokenizer = new
                                    StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            text.set(tokenizer.nextToken());
            Output.collect(text, count);
        }
    }
}
```

The Mappers set the key-value to OutputCollector, which pushes the data to intermediate storage for sorting and shuffling. The key-value pair should be

serialized and a comparable interface should be implemented to write on the network and sort the output simultaneously.

Here, the `org.apache.hadoop.io.Text` class represents a key that provides methods to serialize and compares the texts at the byte level. The value `IntWritable` is again serialized as it implements a writable interface and supports sorting as it implements the WritableComparable interface.

Reducer

The output from sort/shuffle is the input to the Reducer. The Reducer processes the key-value data and writes the output to the disk. The Reducer uses `OutputCollector` for writing the output data into the disk.

```
public static class SampleReducer extends MapReduceBase
implements
        Reducer<Text, IntWritable, Text, IntWritable> {
public void reduce(Text key, Iterator<IntWritable> values,
            OutputCollector<Text, IntWritable> output,
                                        Reporter reporter)
            throws IOException {
        int sum = 0;
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}
```

JobConf

For running the MapReduce, we have to set the Mapper, Reducers, and other properties in `JobConf`. `JobConf` is the main configuration class, which configures the MapReduce parameters for things such as the Mapper, Reducer, Combiner, InputFormat, OutputFormat, and Comparator.

```
public static void main(String[] args) throws Exception {
        String inputPath=args[0];
        String outputPath=args[1];
        JobConf conf = new JobConf(SampleMapReduce.class);
        conf.setJobName("SampleMapReduce");
        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(IntWritable.class);
        conf.setMapperClass(SampleMapper.class);
```

```
        conf.setCombinerClass(SampleReducer.class);
        conf.setReducerClass(SampleReducer.class);
        conf.setInputFormat(TextInputFormat.class);
        conf.setOutputFormat(TextOutputFormat.class);
        FileInputFormat.setInputPaths(conf, new
Path(inputPath));
        FileOutputFormat.setOutputPath(conf, new
Path(outputPath));
        JobClient.runJob(conf);
    }
```

In the above code, the framework executes the map-reduce job, as described in `JobConf`.

Running

Running the MapReduce program is simple. We need to package JAR's Java application, say `samplMapReduce.jar`, and run it as shown below in the command line.

```
$hadoop jar samplMapReduce.jar.jar
org.hadoopetest.mapreducesample. SampleMapReduce /input
folder in HDFS /output folder in HDFS
```

MAPREDUCE COMPOSITE KEY OPERATION

In complex operations where we require a multi-column process, the primary key may not help. For example, if we need to calculate the total population group by country and state, then the selection of keys matters. If we choose the composite key wisely, it could solve the problem quickly. We can create a composite key with the fields "country" and "state." The composite group key implements the WritableComparable interface so that it can sort and write the data output.

Below is the composite key with two fields, country and state, which overwrites the `compare()` method to sort based on the country and state. It provides the `write()` and `readfields()` methods to serialize and de-serialize attributes.

```
private static class CompositeGroupKey implements
    WritableComparable<CompositeGroupKey> {
    String country;
```

```
    String state;
    public void write(DataOutput out) throws IOException
{
        WritableUtils.writeString(out, country);
        WritableUtils.writeString(out, state);
    }
    public void readFields(DataInput in) throws IOException
    {
        this.country = WritableUtils.readString(in);
        this.state = WritableUtils.readString(in);
    }
    public int compareTo(CompositeGroupKey pop)
    {
    if (pop == null)
        return 0;
    int intcnt = country.compareTo(pop.country);
    return intcnt == 0 ? state.compareTo(pop.state) : intcnt;
    }
    @Override
    public String toString() {
           return country.toString() + ":" + state.toString();
    }
}
```

We use the above composite key to create the MapReduce job to count the total population groups by country and state.

Table 7.1 Input

Country	State	City	Population (Mil)
USA	CA	Su	12
USA	CA	SA	42
USA	CA	Fr	23
USA	MO	XY	23
USA	MO	AB	19
USA	MO	XY	23
USA	MO	AB	19
IND	TN	AT	11
IND	TN	KL	10

Table 7.2 Output

Country	State	Total Population
IND	TN	21
USA	CA	77
USA	MO	84

MAPPER PROGRAM

Once we define the composite key, we create the Mapper class that uses input generated from InputFormat. InputFormat splits the file logically and passes it to the individual Mapper, which invokes multiple map tasks. The map task transforms the input split record into a key-value pair, where the key and value are implemented via the WritableComparable interface. The Writable interface provides the capabilities to sort and write the data into a disk. The number of map tasks is decided based on the InputSplits defined in InputFormat. The split is logical, not physical. MapReduce first invokes the `setup ()` method of context and then invokes map (Object, Object, Context) for each input split and, at last, invokes a clean-up (Context) method for clean-up activity. We extend the Mapper class to a basic generic Mapper<Key2, Value1, Key2, Value2> class, which indicates the input and output for the key and value(s). The Mapper class could overwrite the `map()` method to process the input data as shown in the following code.

```
public void map(LongWritable key, Text value, Context
                                                    context)
                throws IOException, InterruptedException
{
    String line = value.toString();
    String[] keyvalue = line.split(",");
    populat.set(Integer.parseInt(keyvalue[3]));
    CompositeGroupKey cntry = new
CompositeGroupKey(keyvalue[0], keyvalue[1]);
    context.write(cntry, populat);
}
```

The `map()` method in the code passes the key as the object and the value as the text. Value contains each line of the file. Inside the map method, we split the line and create the key-value and pass to an intermediate area

through context. The context fills in the I/O buffer with the key-value mapper and later to the local disk. The map task outputs groups by sorted key and writes the data to the local disk as intermediate data. MapReduce also provides a local Combiner that combines the intermediate map output and passes it to the Reducer to cut down the amount of data transferred from the Map to the Reducer.

Reducer

The Reducer copies intermediate map tasks' output to the local disk through HTTP and passes the data to individual Reducers based on each key. Before invoking the individual Reducer task, the Reducer shuffles, merges, and sorts the key-value pairs. The Reducer processes the collection of values for each key and writes it to the disk. The following code is the Reduce method, which is created by the Reducer for each key. Each Reducer task has a single key with a collection of values.

```
public void reduce(CompositeGroupKey key,
Iterator<IntWritable> values,Context context) throws
IOException, InterruptedException
{
    int cnt = 0;
    while (values.hasNext())
    {
        cnt = cnt + values.next().get();
    }
    context.write(key, new IntWritable(cnt));
}
```

Job

To run MapReduce, we have to set the Mapper, Reducers, and other properties in `JobConf`. `JobConf` is the main configuration class, and it configures the MapReduce parameters such as Mapper, Reducer, Combiner, InputFormat, OutputFormat, and Comparator. The code below shows how to create and run the job based on the above mapping and reducing code.

```
Configuration conf = new Configuration();
Job job = Job.getInstance(conf, "GroupMR");
job.setJarByClass(GroupMR.class);
job.setMapperClass(GroupMapper.class);
job.setReducerClass(GroupReducer.class);
job.setOutputKeyClass(CompositeGroupKey.class);
```

```
job.setOutputValueClass(IntWritable.class);
FileInputFormat.setMaxInputSplitSize(job, 10);
FileInputFormat.setMinInputSplitSize(job, 100);
FileInputFormat.addInputPath(job, new Path("/Local/data/
                                          Country.csv"));
FileOutputFormat.setOutputPath(job, new Path("/Local/data/
                                              output"));
System.exit(job.waitForCompletion(true) ? 0 : 1);
```

This main method calls the MapReduce job. Before calling the job, we need to set the MapperClass, ReducerClass, OutputKeyClass, and OutputValueClass. We can also set the FileInputFormat and FileOutputFormat.

Running

Running the MapReduce program is simple. We need to package the Java application in JAR, say `hadooptest.jar`, and run it as shown below in the command line:

```
hadoop jar hadooptest.jar.jar
org.hadooptest.compositesorting /input folder and
/output folder in HDFS
```

MAPREDUCE CONFIGURATION

Their many optional properties could be tuned in *mapred-site.xml* to improve the MapReduce performance. The following are key parameters we can configure:

mapreduce.cluster.local.dir

The *mapreduce.cluster.local.dir* defined the local disk location to store the intermediate output while processing a map reduces tasks. It supports a comma-separated list of local locations to the map task output and distributes it across multiple devices and locations (the default is `${hadoop.tmp.dir}/mapred/local`).

mapreduce.task.io.sort.mb

The total amount of buffer memory to use before sorting files, in megabytes, and, by default, is 100. The Mapper's task output is not stored directly into a local disk; the data is first stored in the buffer memory (configured

by *mapreduce.task.io.sort.mb*). After partially loading a specific limit into the buffer memory (which can be configured), it starts storing into local directory defined by *mapreduce.cluster.local.dir*.

The buffer memory is where output gets sorted before storing it into a local directory. Once the mapper task complete, the output data get shuffled or multiple files get merged into one big file and passed to the Reducer.

Suppose we increase the size of the *mapreduce.task.io.sort.mb* buffer, which decreases the number of files stored in the local disk, resulting in fewer files to shuffle or merge. Since *mapreduce.task.io.sort.mb* takes the memory from the child JVM configured by *mapred.child.java.opts*, increasing the buffer size might impact the memory allocated for the user's code execution. For example, one GB of *mapred.child.java.opts* with 200 MB would only provide 900 MB for the code of the MapReduce job execution.

If the number of files stored in the local directory from *mapreduce.task.io.sort.mb* is much larger, we can probably consider increasing the *mapreduce.task.io.sort.mb* size.

mapreduce.task.io.sort.factor

The *mapreduce.task.io.sort.factor* is defined by the number of files merging at once while sorting and merging. It is an iterative process and keeps sorting and merging files based on the defined *mapreduce.task.io.sort.factor*. Increasing *io.sort.factor* increases the number of files to sort and merge, but it also consumes more memory. In the case of vast numbers of local I/O operations during sorting and shuffling, that means *io.sort.factor* is relatively low compared to the total records.

mapreduce.job.reduces

mapreduce.job.reduces specifies the number of Reducer tasks to use to process a MapReduce job. The number of Reducer tasks can also be increased manually using JobConf's `conf.setNumReduceTasks(int num)`. The default number of MapReduce tasks (default is 1) is overkill for large data files because there would be no parallelism for the data processing. It would be impossible to manage the failover, as all the data would stick with only one reduce task. Keep in mind that many map tasks are just as harmful as a smaller number of tasks. Increasing the number of tasks may cause increased network transfer activity, impacting the overall shuffling. We might consider the criteria below for setting *mapreduce.job.reduces*.

```
Number of map reduce task> input data size/dfs.block.size.
```

The other rule of thumb is that total memory consumed by Mapper and Reducer should be less than the allocated memory

```
(Mapper +reducer) *block size<=allocated memory
```

mapreduce.job.maps

mapreduce.job.maps can be used to increase the number of map tasks, but it does not set the number that Hadoop determines via splitting the input data. The number of map tasks can also be increased manually using JobConf's `conf.setNumMapTasks(int num)`. It is just an indication of the MapReduce task, but the number of InputSplits drives the number of map tasks. This is ignored when the application ID is set to "local."

mapred.child.java.opts

The *mapred.child.java.opts* parameter allows the VM args, such as JVM heap size, and Java system properties, to serve as the child task.

Table 7.3 MapReduce Opts

Option	Description
`-XmxNu`	`Maximum JVM heap size e.g. -Xms2g for 2 GB`
`-XmsNu`	`Initial heap size e.g. -Xms200m for 200 MB`
`-Dproperty=value`	`Application system properties`

mapreduce.map.output.compress

By default, the output of the Map Tasks is uncompressed. However, we can configure it to compress map out, optimizing the I/O data transfer during the shuffling phase. The additional compression requires more CPU time; that is why there are tradeoffs between compression and CPU processing time.

mapreduce.map.output.compress.codec

The *mapreduce.map.output.compress.codec* parameter defines the compression of the codec, such as *Gzip* and *org.apache.io.compress.SnappyCodec*, required to compress the map task output in a MapReduce job. The default value of codec is *org.apache.hadoop.io.compress.DefaultCodec*.

mapreduce.output.fileoutputformat.compress.type

In the case of where the job output is compressed into a Sequence File, then *mapreduce.output.fileoutputformat.compress.type* tells how the compression codes are to be used in the map task.

Table 7.4 Compression Type

Type	Description
RECORDS	Each value individually compressed in sequence manner.
BLOCK	All key value records compressed in group of given size
NONE	No Compression

yarn.resourcemanager.resource-tracker.client.thread-count

This is the part of the YARN configuration setting that controls the number of threads to handle the resource tracker calls. By default, this thread pool is 50.

yarn.resourcemanager.scheduler.class

It is the part of the YARN configuration setting. The *yarn.resourcemanager.scheduler.class* sets the scheduler plugin required to schedule the tasks. By default, YARN uses the CapacityScheduler scheduler.

mapreduce.job.reduce.slowstart.completedmaps

The Reducer task allocates slots after the Mapper's task completion. However *mapreduce.job.reduce.slowstart.completedmaps* provides the option to allocate the Reducer's slots before completing all of the Mapper's tasks, e.g., "0.33" means start allocating Reducer's slots once 33% of the Mapper's task gets completed. The Reducer's tasks begin transferring the data over the network during the shuffle phase. This parameter increases the I/O bandwidths that reduce the CPU runtime of the job.

mapreduce.reduce.shuffle.parallelcopies

The Reducer's task uses parallel worker threads to copy the intermediate map output data from the local disk. The number of worker threads is configured through the *mapreduce.reduce.shuffle.parallelcopies* parameter. The default

value of *mapreduce.reduce.shuffle.parallelcopies* is five, which could be reconfigured. These parallel threads utilize the network bandwidth to optimize the copy during the shuffle process. However, note that an excessive use of *mapreduce.reduce.shuffle.parallelcopies* also impacts the copy data file because it overwhelms the network, and as a result, the job becomes slower.

mapreduce.shuffle.max.threads

mapreduce.reduce.shuffle.parallelcopies configures the number of parallel worker threads that copy data from the local disk to the Reducer node during the shuffle phase, whereas *mapreduce.shuffle.max.threads* controls the number of threads available to handle requests concurrently.

CHAPTER 8

HIVE

HDFS files can be processed using MapReduce in the Hadoop environment. MapReduce is an efficient approach to handling extensive data; however, it is tedious to write numerous MapReduce programs to get different data types. If you require a more complicated process that necessitates multiple Mappers and Reducers, it becomes monotonous to create MapReduce jobs. Developers have wasted many hours writing small MapReduce programs to query results from Hadoop HDFS files.

There was a need to improve Hadoop's processing capabilities to analyze the data more productively. Hive is the solution to these problems. Apache Hive is an open-source Big-Data-analyzing framework built on top of Hadoop, with a SQL-like language: HiveQL. Apache Hive permits the developer to write HiveQL statements that are similar to standard SQL statements.

Figure 8.1 depicts how different Hive various components interact with each other.

Hive's query language runs on different computing engines, such as MapReduce, Tez, and Spark. Hive can be accessed through the Command Line, Web Interface, and Thrift Server. HiveQL can be accessed in many ways, such as JDBC, ODBC, and CLI via the Hive Thrift Server. The Hive Thrift Server communicates with Hive's driver to compile, transform, and run.

The following are fundamental characteristics of Hive:

- The Apache Hive tool provides an abstraction of SQL to access Hadoop data by using a standard SQL-type language called Hive Query Language (HQL).
- The HiveQL language requires similar skills as those needed for SQL, so it's easy and faster to use compared to MapReduce.

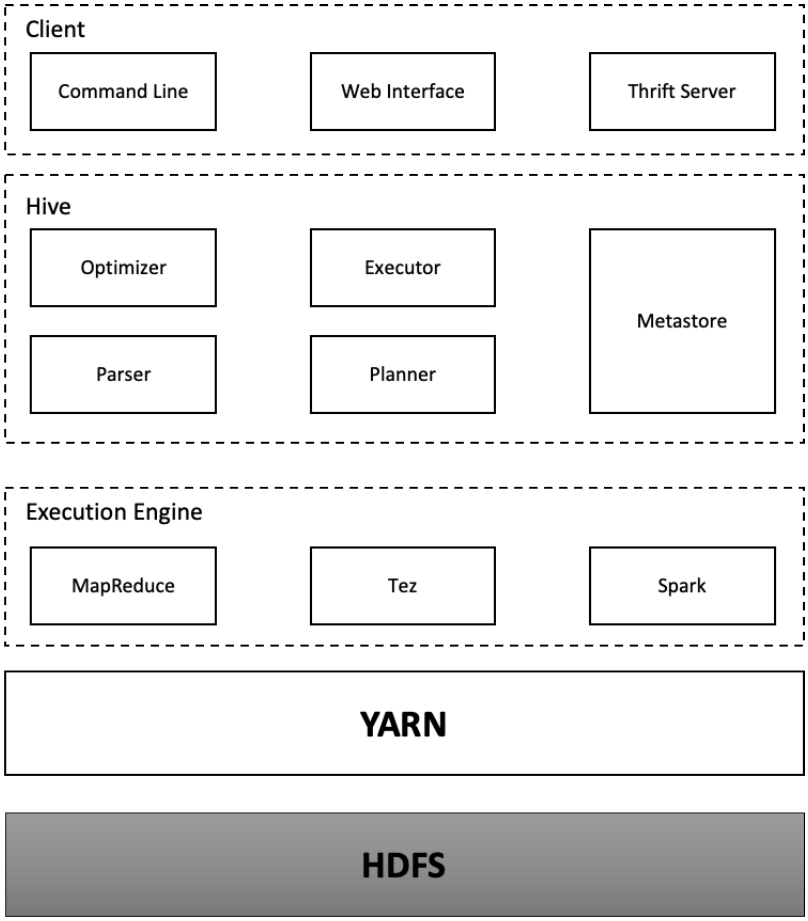

FIGURE 8.1 Hive Components

- HiveQL takes much less time to process and analyze the HDFS data.
- Hive translates the HiveQL language into the MapReduce code.
- Hive stores schema information, such as table names, column names, data types, and partition information.
- Hive can support many file formats and data formats.

Hive processes large sets of data. Therefore, it has a high latency (counted in minutes), so it is not appropriate where frequent writes are needed, and it is not suitable for applications that require speedy responses. HiveQL is similar to the RDBMS SQL language; however, there are several key differences.

Table 8.1 RDBMS vs. Hive

	RDBMS	Hive
Language	PL/SQL	Open and read a file
CRUD	Insert, update, and delete	Insert overwrite; no update or Delete(***available with Hive Transaction in Hadoop3***)
CRUD Transaction	Yes	Yes (optional configuration in Hadoop 3)
Latency	Seconds, milliseconds	Minute or more
Indexes	Any number of indexes, very important for performance	No indexes, data is always scanned (in parallel). However, defined partitioned can improve the performance.
Data Size	TBs	PBs

HIVE HISTORY

Prior to Hive, organizations used a commercial RDBMS-based data warehouse to analyze and process large datasets. But when the daily amount of data grew from gigabytes to terabytes and then to petabytes, and it was inadequate to handle such sizeable growing datasets. As a result, Facebook started using Hadoop, which improved data processing performance. It was not only less expensive but it also improved throughput.

Hadoop's MapReduce is not an easy to work with, and it doesn't advocate re-usability. A developer can spend hours building MapReduce programs to analyze straightforward datasets, which impacts productivity. Considering these challenges, Facebook created Hive to address MapReduce's complexity. Hive brings the familiar concepts of tables, columns, partitions, and a subset of SQL to the unstructured world of data while still maintaining Hadoop's extensibility and flexibility. Hive was open-source upon its first release in 2010. Many prominent organizations use it, such as Amazon, IBM, Yahoo, Netflix, and the Financial Industry Regulatory Authority (FINRA). Initially, it used MapReduce as an execution engine; later releases support the Tez and Spark engines.

HIVE QUERY

The Hive query (HiveQL) behaves like SQL. It supports SQL queries like select, joins (such as inner join, left outer join, and right outer join), Cartesian group by, and union all. It also supports various DBMS-like commands, such as show tables, create tables, and describe tables.

DATA STORAGE

Tables in the Hive are associated with the table data for the HDFS directories and map the HDFS directories' data.

- **Table:** A logical structure in Hive maps the data from HDFS directories.
- **Partition:** A table's partition is stored in sub-directories in HDFS directories.
- **Buckets:** A bucket is stored in a file within the partitioned table's directory.

File Formats

Hive doesn't restrict the input file format where the data needs to be stored. The file format can be defined while creating a table. Below are a few examples:

- `TextInputFormat`: For text files
- `SequenceFileInputFormat`: For binary files
- `RCFileInputFormat`: For column-oriented data
- `Avro format`: For schema evolution
- `RC File format`: For record columnar data
- `ORC`: For optimized row columnar format
- `Parquet`: For columnar data

The following code is for a table using the sequence file format.

```
CREATE TABLE EMPLOYEE (Firstname String, LastName
String, Address String) STORED AS
INPUTFORMAT
'org.apache.hadoop.mapred.SequenceFileInputFormat'
OUTPUTFORMAT
 'org.apache.hadoop.mapred.SequenceFileOutputFormat'
```

The `STORED AS` declared here is used to define the input and output formats.

DATA MODEL

Hive table's column can be either primitive types such as integers, doubles, floats, strings, bigint, or complex types, such as maps and structures.

Hive stores data in tables the same way as RDBMS does. Each column has an associated type, which is either a primitive or a complicated type.

Currently, the following primitive types are supported:

- **Integers** – bigint (8 bytes), int (4 bytes), smallint (2 bytes), and tinyint (1 byte). All integer types are signed.
- **Boolean** – Boolean-True/False
- **Floating point numbers** – float (single precision), double (double precision)
- **Fixed point numbers** – Decimal (a fixed point value of user-defined scale and precision)
- **String** – String (sequence of character), varchar (string with maximum length), char (sequence of character)
- **Date and time types** – Timestamp (a date and time with or without a time zone), Date
- **Binary types** – Binary (sequence of bytes)

Hive also supports the following complex types:

- **Arrays** – Arrays (indexable lists)
- **Maps** – map<key-type, value-type>
- **Structs** – struct<file-name: field-type, ... >
- **Union** – UNIONTYPE<data_type, data_type, ...>

These complex types are templates and can be composed to generate types of arbitrary complexity. For example, `list<map<string, struct<p1: int, p2: int>>` represents a list of associative arrays that map the `strings` to the `structs` type. This struct type contains two integer fields named `p1` and `p2`. These can all be put together in a create table statement to create tables with the desired schema. For example, the following statement creates a table, `t1`, with a complex schema.

```
CREATE TABLE EMPLOYEE(name string, address
list<map<string, struct<p1:int, p2:int>>);
```

Query expressions can access fields within the `structs` using a "." operator. Values in the associative arrays and lists can obtain using the "[]" operator. In the previous example, `Employee.address[0]` gives the first element of

the list, and `Employee.address[0]['key']` gives the struct associated with key in that associative array. Finally, the `p2` field of this struct can be accessed by `Employee.address[0]['key'].p2`. With these constructs, Hive can support structures of arbitrary complexity. Java data types inspired the Hive data types; therefore, it's easy to relate Hive data types with Java data types (except for the character array, because Hive doesn't support a character array).

Table 8.2 Primitive Data Types

Type	Size
`TINYINT`	1-byte signed (–128 to 127)
`SMALLINT`	2-byte signed
`INT`	4-byte signed integer
`BIGINT`	8-byte signed integer
`FLOAT`	4-byte single precision floating point
`DOUBLE`	8-byte double precision floating point
`STRING`	Sequence of characters defined by single or double quote
`Varchar`	String with length specified (between 1 and 65355) Available from Hive 0.12.0
`CHAR`	Fixed length up to 255 Available from Hive 0.13.0
`Timestamps`	Floating point numeric types: Interpreted as UNIX timestamp in seconds with decimal precision Strings: JDBC compliant java.sql. Timestamp format "YYYY-MM-DD HH:MM:SS.fffffffff" (9 decimal place precision)
`BOOLEAN`	True or False
`BINARY`	Arbitrary byte not to parse to data type Available from Hive 0.8.0

`TIMESTAMPS` uses `to_utc_timestamp, from_utc_timestamp` to convert to and from the timestamp to UTC.

Similar to Java typecasting, Hive can also implicitly cast any numeric type to the more significant data type as needed (for example, `FLOAT` to a `DOUBL SMALL` integer or `FLOAT`, `DOUBLE`). Hive can also explicitly cast from one data type to another data type by using CAST (abs as INT).

COMPLEX DATA TYPES

Hive uses complex data type structures such as ARRAY, MAP, and STRUCT, built on top of primitive data types. These types have the same behavior those types in Java and support nesting type structures.

Table 8.3 Complex Hive Data Types

Type	Size	Example
ARRAY	Ordered sequence of same type access by indexing ARRAY<data_type>	["red","yellow","green"]
MAP	A key-value collection. MAP<primitive_type, data_type>	{1:"red",2:"yellow"}
UNION	UNIONTYPE<data_type, data_type, ...> Available from Hive 0.870	{1:["red","orange"]}
STRUCT	Similar to JSON type object and field can be accessed by dot. STRUCT<col_name : data_type [COMMENT col_comment], ...> {1, "red"}	

HIVE DDL (DATA DEFINITION LANGUAGE)

Hive's database works as a catalog to keep the table namespace, and it helps to categorize a group of tables for a user and avoids name collision. Hive comes with the default database (default); however, we can also create a database. Hive creates folders/directories for each database, and sub-directories for tables consist of that specific database; however, we can explicitly define the other location while creating the table.

HiveQL is the Hive query language similar to RDBMS SQL language, but it is meant for Big Data processing. It has significant differences with the SQL query language:

- Hive doesn't support insert, update, or delete for records.
- It sacrifices transactions for performance during massive dataset processing

This section discusses various SQL operations, such as creating, inserting, viewing, functions, and tables. As explained below, we can manage tables and databases.

Table 8.4 Hive's Basic Queries

Operation	Query
Create database	`CREATE DATABASE IF NOT EXISTS ABC;`
Show list of existing databases	`show databases;`
List databases based on regular expression	`SHOW DATABASE like 'A*'`
Create dataset on explicit location	`Create database abc location 'usr/joe/abc'`
Describe database	`describe database abc;`
Drop Database	`Hive> drop database If exists abc;`

Create table	`Create table if not exists employee(`
	`Name STRING comment 'employee name',`
	`Salary float float,`
	`Address struct<street:STRING, city:STRING,zip:INT>`
	`PARTITIONED BY(year INT COMMENT 'Year', month INT COMMENT 'Month')`
	`ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'`
	`LOCATION '/usr/joe/employee'`

Use specific database	`use abc;`
Show tables in particular database	`show tables;`

Operation	Query
Show tables by using regular expression	`show tables like 'emp*';`
Detail about the table	`describe extended employee;`
Create external table	`create external table if not exists employee(name String, salary float) row format delimited fields terminated by ',' location '/usr/joe/employee'` `The EXTERNAL create table to external location`
Show partition for a table	`show partitions employee;`
Drop table	`DROP table if exists employee;`
Rename table	`alter table employee rename to employee_all;`
Truncate Table	`Removes all rows from a table or partition(s). The rows will be trashed if the filesystem Trash is enabled, otherwise they are deleted`
Alter partition	`alter table employee ADD if not exists partition (year ='2014',month='09') location '/user/joe/2014/09';`
Alter partition location	`alter table employee partition(year='2014', month='08') set location '/user/joe/employeenew/2014/08'`
Alter table columns	`alter table employee change column columnname newcolumnname type;`
Add column	`alter table employee add column(email String comment 'Employee email address')`
Replaces all the columns with new columns	`alter table employee replace columns(firstname String, lastname String, Address String)`

TABLES

Hive structured HDFS data files into tables, columns, rows, and partitions, and this metadata information is stored in a database called Metastore. It supports basic primitive types, such as integers, floats, doubles, and strings, and complex types, such as maps, lists, and structs. The Hive table's query language is similar to traditional RDBMS SQL. A table could be a partitioned table or non-partitioned table. Adding `PARTITIONED BY` in the `Create Table` statement as below could create a portioned table.

```
Create Table employee (firstname string, lastname string)
PARTITIONED BY (datetime String COMMENT datetime);
```

In the above example, the table partition is stored in */Hadoop_HDFS_PATH/TABLE_PART*. There is a distinct partition for each timestamp in the HDFS directory. A new partition is created by either using the `Alter` command or `Insert` command, shown in the following code:

```
Insert overwrite table employee_part PARTITION
(datetime ='2015-01-01') SELECT * FROM employee;
Alter table employee ADD PARTITION (datetime ='2015-01-02');
```

The above `Insert` command in this code creates a new table, partitioned with data from an employee, whereas the `Alter` command creates an empty partition. The above statement creates the following HDFS directories:

/Hadoop_HDFS_PATH/employee/2015-01-01 and
/Hadoop_HDFS_PATH/employee/2015-01-02

The hive scans directories to pull the data, as follows:

`Select * from TABLE_PART where datatime='2015-01-02'` scans data in all files within the directory `/Hadoop_HDFS_PATH/employee/2015-01-02`.

VIEW

The view is the logical construct on the Hive query that abstracts the large complex SQL query. It doesn't store any physical data; it is just a reference query that gets appended with the rest of the query.

The key benefit of Hive's `VIEW` is that it hides the query complexity, such as joining and subqueries. It provides a single abstraction, so developers can use `VIEW` without going into query complexity.

You can imagine a view as a sub-query that first gets executed and then applies it to the rest of the query. Create a `VIEW` script that can use any valid select query, as shown below:

```
CREATE VIEW VIEW_EMP_CA AS SELECT * FROM EMPLOYEE WHERE STATE
WHERE STATE='CA'
```

Now you can get the result from `VIEW_EMP_CA`, as shown below:

```
SELECT * FROM V_EMP_CA WHERE salary>2000;
```

PARTITION

Hive partitions the data in the table, which maps to a particular folder (for example, the `purchase_history` table partition with the year and month). Purchase history data for the 21st of June, 2014, goes to the folder location `purchase_history/2014/06` in the HDFS. The Hive query gets optimized to go to a specific year and month to fetch results, not to scan all the files located in the HDFS.

Partitioning Hive tables is a powerful tool to optimize the Hive query on large data sets. Hive can utilize this knowledge to restrict scanning to the folder before querying it. Partitioning reduces the data required to scan and filter initially.

The partition does the horizontal scaling of data; basically, it slices large datasets into small segments to avoid a full scan. By defining and partitioning, it creates sub-directories for each slice or partition column. If you have multiple partitions, the sub-directories are nested based on the columns' order in the definition statement. For example,

- `./state=CA/employer=ABC`
- `./state=CA/employer=XYZ`
- `./state=NA/employer=TYZ`

Now when a Hive query adds the criteria (the `where` clause), then the partition scans only specific directories, not all of them.

Therefore, when we select a list of employees where "state=CA," it limits the scan to only files/subdirectories under the "state=CA" folder and, as a result, improves performance dramatically. Not only does it improve performance, but it also logically organizes data. Without partitioning, Hive scans all the records from the directories and sub-directories, which is expensive and slows the process on large sets of data.

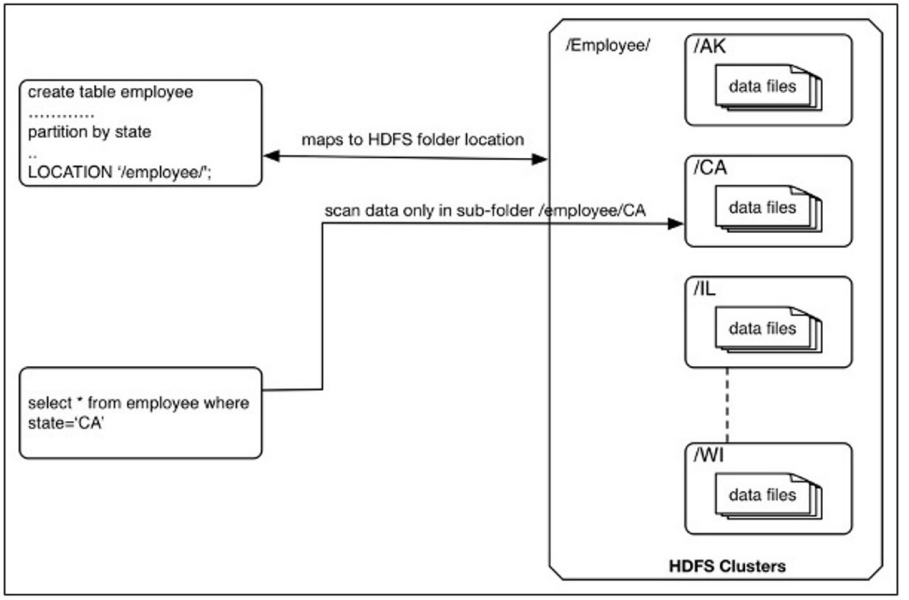

FIGURE 8.2 Hive partition strategy

Partitioning mechanisms are very effective only if we define the partitions wisely. They enable standard filtering in Hive queries; otherwise, there will be too many partitions, causing more overhead on HDFS's NameNode. If you have a large number of small partitions, scanning directories becomes more expensive than merely scanning data.

BUCKETING

Sometimes, partitions on small datasets result in many sub-folders/files that cause overhead on the NameNode. In turn, Hive has to do more work to scan all the small folders/files.

Data can be bucketed for each partition, which further improves the query performance. The right partition enhances query performance and manages the data, e.g., achieving the old data. We can do bucketing on a column, or we can do bucketing randomly. Setting up an excellent bucketing scheme helps in fast data sampling, rapid data verification, and quick data analysis.

Bucketing improves these scenarios by decomposing the table data into a fixed number of buckets. The statement below shows how the bucketing column gets created at the time of table creation:

```
CREATE TABLE EMPLOYEE (
name
employer
state
phone
) PARTITION BY state
CLUSTERED BY (name) INTO 10 BUCKETS;
```

Each slice goes to specific folders in partitions, whereas in bucketing, a hash value assigns them into individual buckets. In the above example, we created ten buckets and are grouping using one name. Each bucket contains multiple employees, while restricting similar employee names to a single bucket.

As shown in the example above, if we partition the table on the "name" column, it will result in many partitions that impact performance because each partition's meta information is collected into NameNode. But if we create buckets for the employee "name," then the same employee name will always go into the same buckets with those hash values. Since the number of the bucket is fixed, it doesn't fluctuate with the data. If two tables with similar bucketing join, the bucket's fields with the same value go to the same buckets, making it easy to perform Hive queries while sampling. Partition bucketing optimizes the table joining operation further, which reduces the data that needs to be parsed.

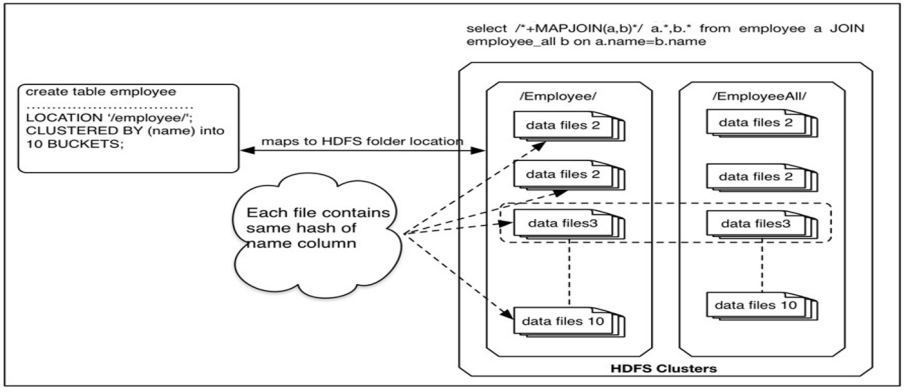

FIGURE 8.3 Hive MAPJOIN

Figure 8.3 shows that MAPJOIN selects query scans of a few relevant files in each Mapper task instead of performing a full table scan. It is good practice to optimize the bucket numbers; too many buckets or too few can lead to trouble with optimization. Since bucketing depends on the data load, the load will be balanced on each bucket. To optimize each bucket load, we can either set the Reducer number as the same as the bucket size or enable bucketing enforcement, as shown below:

```
set hive.enforce.bucketing = true;
set map.reduce.tasks = <bucket number>
```

HIVE ARCHITECTURE

Hive runs on top of Hadoop to process and analyze data using a SQL-like language called HiveQL. Hive translates the HiveQL query language into the execution plan and executes it. Figure 8.4 depicts how the main components of the Hive interact with each other.

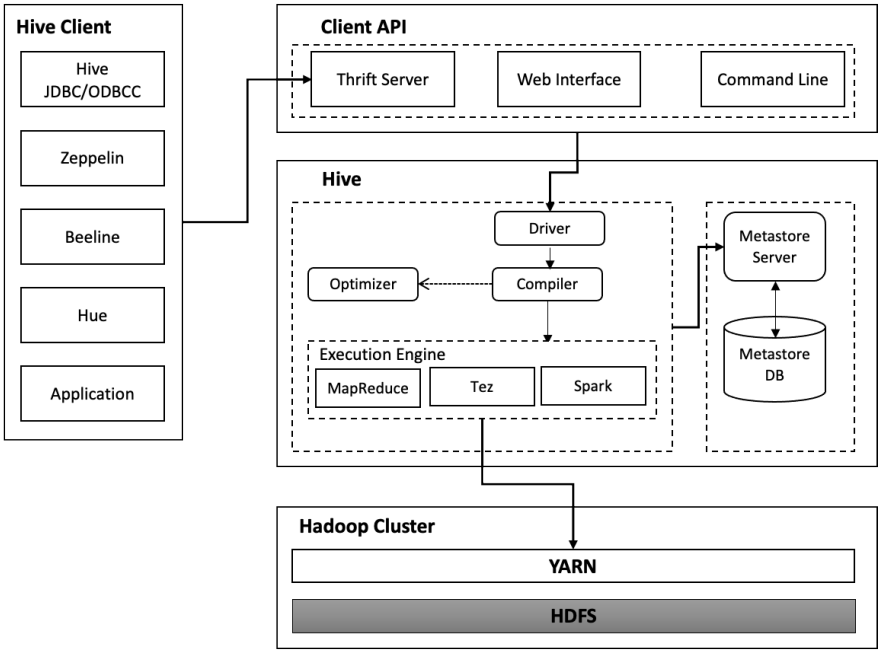

FIGURE 8.4 Hive architecture

The following components are the main building blocks in Hive:

- **Hive Client:** External tools and applications can communicate through Hive using the Hive Thrift Server. We can connect with any SQL tools using Hive JDBC/OBC. We can also communicate with Hive through the Thrift Server, Beeline, Hue, Zeppelin, or any custom application.
- **Client API:** Hive provides a client API to access Hive like the command line (CLI) and Web Interface and Thrift Server.
- **Metastore:** This component holds all the information about the tables, such as partitions, column, *SerDr* (serialize and de-serialize) information, and specific HDFS files. All the components of the Hive interact with Metastore to fetch the data.
- **Thrift Server:** The Thrift Server provides a platform to enable APIs for different languages, such as Java (JDBC), C++(ODBC), PHP, Perl, and Python, to execute HiveQL without the CLI command. It is an alternative of CLI that performs HiveQL in various languages.
- **Driver:** The driver receives the HiveQL request from the CLI or Thrift Server and processes those requests using compilation, optimization, and execution. It creates sessions for each application to maintain the state and flow of requests.
- **Compiler:** The compiler parses the HiveQL and interacts with Metastore to get the metadata information and generate the execution plan.
- **Execution Engine:** The Driver maintains the lifecycle of a HiveQL script and passes the execution plan to the Execution Engine. The Execution Engine gets the necessary metadata information from the Metastore and submits a job to Hadoop and sends the results back to Driver once it gets completed.
- **Serializers/Deserializers (SerDr):** This component has the framework libraries that allow users to develop serializers and deserializers for their data formats.
- **Command Line Interface:** This component has all the Java code used by the Hive command-line interface.

Process Flow

A HiveQL statement is submitted to the Hive Driver through the CLI, Web UI, or JDBC/ODBC. The Hive Driver is the standard interface, which parses, compiles, and executes the HiveQL statement. It is an interface consisting of multiple implementations for different integration points.

The Driver accesses the metadata information, such as cluster, partition, and file information from Metastore and parses and compiles the HiveQL and generates the Driver-optimized execution plan using the optimizer. Finally, it gets converted into an execution plan and submits it to the execution engine. The execution engine then executes these tasks on the Hadoop platform.

SERIALIZATION/DESERIALIZATION (SERDE)

In Hive, data uses serializers and deserializers to read data and write it back out to the HDFS in any format. Hive uses SerDe, i.e., serialize and deserialize, to read and write data from HDFS files. For instance, a Hive query internally uses SerDe to deserialize a record from a file returned to Hive, and the same thing happens when Hive loads the data and uses SerDe to serialize data into a file.

Hive has a default implementation of the SerDe Java interface called LazySerDe. The user implements the SerDe interface to build custom data formats and associate it with a table or partition. LazySerDe, the default implementation of the SerDe Interface, deserializes rows into objects slowly, so the data is retrieved, if needed.

METASTORE

Metastore is a repository containing information about Hive tables such as partitions, schemas, columns, and file locations. Hive's Driver accesses the metadata information while compiling it into MapReduce tasks.

Metastore stores the metadata information into its RDBMS repository using DataNucleus. DataNucleus is an open-source ORM layer used to convert an object into relations and vice versa. The RDBMS repository could be anything, such as MySQL.

Hive's Driver gets the metadata information required to process HiveQL, converts it into XML, and passes it to the MapReduce, created by the compiler. It clarifies the separation between Metastore and Hadoop's MapReduce and facilitates the scaling of the Metastore repository in the future.

QUERY COMPILER

- The Query Compiler parses the input HiveQL and generates the execution plan.
- Hive parses the query and generates an abstract syntax tree (AST).
- The Compiler fetches the metadata information from Metastore and creates a logical plan.
- The Compiler validates the AST and converts it into operator DAG.
- The Optimizer takes input as DAG and optimizes it by applying chains of transformations such as Node, GraphWalker, Dispatcher, Rule, and Processor.
- The Optimizer process uses DAG by identifying and walking each node.
- Each node gets visited and matches the rule to identify the matching rule.
- The Dispatcher, which maintains the mapping between the rule and processor, is used to get the identified rule's processor.
- The identified processor is passed to the GraphWalker.
- The Compiler uses different execution engines to generate an execution plan, e.g., MapReduce generates MapReduce plan, Tez generates Tez plan, and Spark generates spark plan.
- Finally, the Executor submits a job to Hadoop's YARN cluster.

HIVESERVER2

Like the database server, Hive uses the Hive Server to communicate with the client and execute queries. In the initial release, Hive used basic Hive Server1, which is non-scalable, to support a large number of concurrent requests, but it was not secure enough.

To address the above challenges, Hive introduced HiveServer2 (Hive 0.11) that supports sizeable concurrent requests and security. HiveServer2 provides an environment for Hive execution. It creates a new Hive execution context session to serve Hive queries.

HiveServer2 uses a Thrift-based RPC call to handle concurrent requests. It also supports pluggable authentications, such as Kerberos and LDAP. HiveServer2 includes the Hive JDBC Driver to support JDBC calls for Hive queries similar to other databases. HiveServer JDBC works with how we connect different database servers using the JDBC driver, as shown below:

```
Class.forName("org.apache.hive.jdbc.HiveDriver");
Connection connect = DriverManager.

getConnection("jdbc:hive2://localhost:10000/default",
    "user", "password");
```

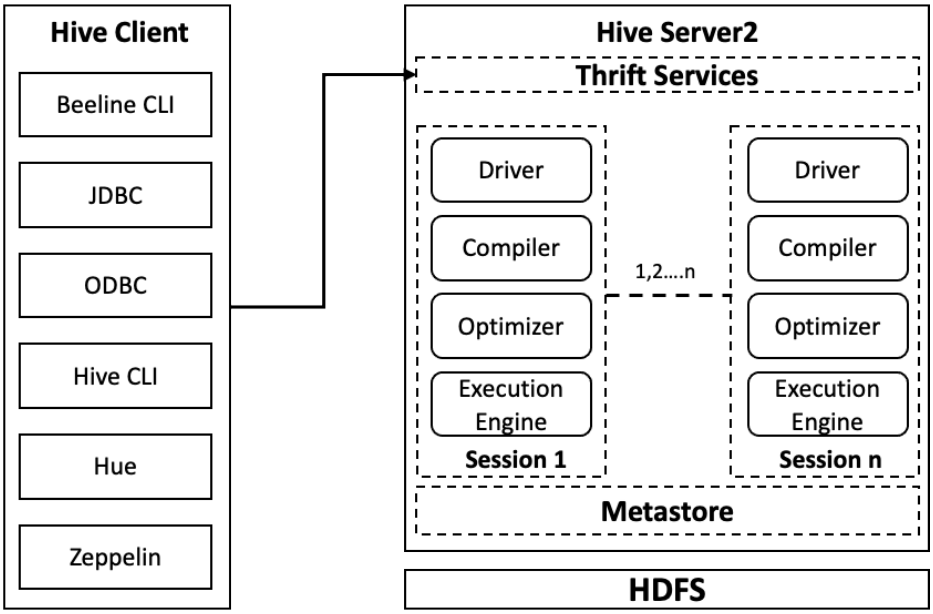

FIGURE 8.5 HiveServer2

Clients have to specify the JDBC driver and connection the URL along with the credentials to connect with the Hive server.

CHAPTER 9

GETTING STARTED WITH HIVE

HIVE SET-UP

A Hive set-up is can be done in three steps. Below is a brief description of how to set up Hive.

Prerequisites

- Java 8
- Installed Hadoop3.x (as explained in the Hadoop set-up)
- Cygwin, in case of a Windows OS

Download

- Download tar from *http://hive.apache.org/releases.html*
- `$ tar -xzvf hive-x.y.z.tar.gz`
- Extract into */Users/joe/software/hive-a.b.c*

Set-up Environment

```
$ export HIVE_HOME=/Users/application/apache-hive-x.y.z
$ export PATH=$HIVE_HOME/bin:$PATH
```

Add it into your bash profile so you can avoid setting it again and again.

Running Hive

Please make sure you should have Hadoop in your class path, as given below:

```
export HADOOP_HOME=<hadoop-install-dir>
```

Create the HDFS folders */tmp* and */user/hive/warehouse* to configure *hive.metastore.warehouse.dir* with permission `chmod g+w`.

- `$HADOOP_HOME/bin/hadoop fs -mkdir /tmp`
- `$HADOOP_HOME/bin/hadoop fs -mkdir /user/hive/warehouse`
- `$HADOOP_HOME/bin/hadoop fs -chmod g+w /tmp`
- `HADOOP_HOME/bin/hadoop fs -chmod g+w /user/hive/warehouse`

Hive CLI

Launch the Hive shell as below

```
% hive
hive>
```

Hive could be launched using the Hive command or we could also use the `-e` option run from command inline. By default, Hive keeps the default database:

```
Hive> show databases;
Or $hive -e 'show databases'
OK
Default
```

HiveQL is not case sensitive. Below is the command use to display the list of tables in current the database, which is "default."

```
Hive> show tables;
OK
```

Tables

A Hive table's data is stored in the HDFS filesystem and associated with the schema stored as metadata in the Metastore.

The table could be one of two types:

- **Managed table:** Hive manages the data stored in the HDFS, and if the table gets dropped, the data is also removed from the HDFS.

- **External table:** The external table maps already existing data in the HDFS filesystem, and if the table gets dropped, the data is still available in the HDFS filesystem.

Create Table

```
Create file with value
Employee.txt
================
name1,address1
name2,address2
```

Header

We can see the column header at the top by using the following command:

```
hive> set hive.cli.print.header=true;
```

We can put the above line on *$HOME/.hiverc* to see the header by default.

Hadoop Command from Hive

We can run the Hadoop DFS command from the Hive prompt:

```
Hive> dfs –ls ;
```

We use DFS's `–help` to see all the commands:

```
hive> dfs -help;
```

Copy Local File to the HDFS

```
Hadoop fs –copyFromLocal employee.txt /tmp/employee/
```

```
CREATE EXTERNAL TABLE IF NOT EXISTS employee (EMPID
STRING, EMPNAME STRING) ROW FORMAT DELIMITED FIELDS
TERMINATED BY ','
LINES TERMINATED BY '\n'
LOCATION '/user/joe/employee';
```

```
Hive> select * from employee;
OK
name1 address1
name2 address1
```

HIVE CONFIGURATION SETTINGS

Hive uses the *hive-site.xml* configuration file to set up the Hive properties. We may also set these properties to be on-demand for a particular Hive query by directly using Hive CLI. For example,

```
Hive>set hive.exec.parallel=true;
```

Below are some key properties to set up.

hive.execution.engine

This identifies the Execution Engine to use while creating the execution plan. The *hive.execution.engine* is used to specify the Execution Engine while processing the Hive query. Its default is `mr` (MapReduce), but it can use `tez` for (Tez execution engine) and `spark` (Spark execution engine).

```
hive.execution.engine=mr(default could be tez or spark)
```

Note: `mr` has been decommissioned and could be removed in a future release.

Tez

The Tez framework executes on YARN with complex directed acyclic graphs (DAC). It's one of the alternatives to MapReduce and Spark.

mapred.reduce.tasks

The *mapred.reduce.tasks* is used to set a reduced task number. Hive sets this value to –1 by default, meaning that Hive will automatically identify Reducer numbers. This value gets ignored when *mapred.job.tracker* is set to "local."

hive.exec.reducers.bytes.per.reducer

The *hive.exec.reducers.bytes.per.reducer* specifies the size of a Reducer in bytes. The default value is 256 MB i.e., 256,000,000 bytes. Hive identifies the number of Reducers based on this parameter. For instance, if the total file size is 20 GB and *hive.exec.reducers.bytes.per.reducer* is 1 GB, then the number of Reducers is 10.

hive.limit.optimize.enable

The limit in a Hive query obtains a few records from a result set; however, it still reads all the query results that impact the memory's performance. We can

restrict the limit by enabling *hive.limit.optimize.enable* as "true," as shown below:

```
hive.limit.optimize.enable=true
```

After enabling the limit, we can set a maximum limit and maximum file to sample, as shown below:

```
hive.limit.row.max.size=200000
hive.limit.optimize.limit.file=10
```

The only drawback is that this does not provide actual results, especially on an aggregated function, because it works on a few sample files.

hive.exec.parallel

hive.exec.parallel allows us to process tasks for mapping, reducing, merging, and sampling, in parallel, and the default is false. It improves the overall execution time but increases the cluster utilization.

```
hive> set hive.exec.parallel=true;
```

We can also set the "how many" thread by using *hive.exec.parallel.thread.number* to set the number of threads to execute a map job in parallel; by default, its value 8.

```
Hive>hive.exec.parallel.thread.number=10;
```

Table 9.1 Hive Configuration Parameters

Property	Default	Description
hive.execution.engine	mr	To select the Execution Engine: mr for MapReduce tej for Tej, spark for Spark
hive.execution.mode	container	Chooses whether query fragments will run in container or in llap
mapred.reduce.tasks	-1	To set the number of Reducers
hive.exec.reducers.bytes.per.reducer	256	Size per Reducer
hive.exec.reducers.max	1009	To set max number of Reducers

(continued)

Property	Default	Description
hive.hadoop.supports.splittable.combineinputformat	False	To combine small input files to bug file
hive.map.aggr	True	To set map-side aggregation in Hive group by queries
hive.mapred.local.mem	0	To set the local memory of MapReduce job in local mode
hive.map.aggr.hash.percentmemory	0.5	To set ratio of total memory to be used by map-side group aggregation hash table
hive.optimize.groupby	True	To enable the bucketed group by from bucketed partitions/tables.
hive.multigroupby.singlereducer	True	To optimize multi group by query to generate a single M/R job plan
hive.optimize.index.filter	False	To enable automatic use of indexes
hive.join.cache.size	25000	To set the number of rows to be cached in in the joining
hive.mapjoin.bucket.cache.size	100	To set the keys value to cache in map-joined table
hive.mapred.mode	strict	To run Hive query in strict or nonstrict mode
hive.exec.compress.output	False	To enable final output will be compressed.
hive.exec.parallel	False	To execute jobs in parallel.
hive.exec.parallel.thread.number	8	To set number of job to execute in parallel
hive.limit.optimize.enable	False	To enable to fetch a smaller subset of data for simple LIMIT first.
hive.limit.optimize.fetch.max	50000	To set maximum number of rows allowed for a smaller subset of data for simple LIMIT

LOADING AND INSERTING DATA INTO TABLES

Load files into tables

This section covers loading data from files copied or moved from a file to the corresponding Hive table's directory. For a partitioned table, the data is loaded into a specific partition. Below is the syntax for loading data from files:

```
Load data [LOCAL] inpath 'filepath' [OVERWRITE] INTO
TABLE tablename [PARTITION (partcol1=val1,
partcol2=val2 ...)]
```

- `[LOCAL]`: Local is optional, and in case of the present, it will locate the file from the local filesystem.
- `FILEPATH`: filepath specified. Filepath can be relative path, absolute path, or full URI.
- `[OVERWRITE]`: Overwrite is an optional scenario. In case of an overwrite, the table data will get overwritten with new data.
- `tablename`: Hive table
- `[PARTITION (partcol1=val1, partcol2=val2 ...)]`: Partition is an optional scenario. If table is partitioned, then we define the partition in load data command.

Example

```
CREATE TABLE employee (name int, salary int)
PARTITIONED BY (dept int) STORED AS ORC;
LOAD DATA LOCAL INPATH 'filepath' INTO TABLE employee;
```

INSERT FROM A SELECT QUERY

We can insert into the Hive table using the select query, as shown below:

Insert from Select

As shown below, the `insert` statement allows for new data to be appended into a table and partitions, keeping the existing data intact.

```
INSERT INTO TABLE employeeCA partition (country= 'US',
state ='CA') select * from employee where state='CA'
TBLPROPERTIES ("immutable"="true")
```

The default is "immutable"="false". The Hive table immutable feature does not allow for us to insert data into a Hive table if the Hive table already exists. In case of the empty immutable table, insert will work. The immutable property protects the Hive table from accidentally inserting data multiple times.

Insert Overwrite

Insert overwrite replaced existing records and partitions from a Hive table.

```
Insert overwrite table employeeCA partition
(country='US', state='CA') select * from employee where
state='CA'
TBLPROPERTIES("auto.purge"="true"):
```

If auto.purge=true is included with TBLPROPERTIES for the Hive table, then the overwritten data of the table is not moved to the trash. Please note that the auto.purge property is only applicable to managed tables.

Dynamic Partition Insert

The above insert may cause a problem with many partitions, so we can use dynamic partition insert described below. First, we enable the dynamic partition:

- hive.exec.dynamic.partition(default true): true will enable dynamic partition inserts
- hive.exec.dynamic.partition.mode (default is strict)
  - strict: must specify at least one static partition
  - nonstrict: all partitions are allowed to be dynamic
- hive.exec.max.dynamic.partitions.pernode (default 100): maximum allowed dynamic partitions for each mapper/reducer node
- hive.exec.max.dynamic.partitions (default 1000): maximum allowed dynamic partitions
- hive.exec.max.created.files (default 100000): maximum number of HDFS files created by all Mappers/Reducers in a MapReduce job
- hive.error.on.empty.partition(default false): Throws an exception in the case when results are generated

The following shows an example of a dynamic query:

```
INSERT OVERWRITE TABLE employee Partition
(country,state) Select em.*,em.country,em.state from employee
```

Hive identifies the partition field's values from last column of the above selected query.

LOAD TABLE DATA INTO FILE

Use `INSERT OVERWRITE` as defined below to export data from the table into the local directory.

```
INSERT OVERWRITE LOCAL DIRECTORY
'/usr/joe/tmp/employee'
SELECT firstname, lastname, address from employee;
```

CREATE AND LOAD DATA INTO A TABLE

```
Create table employeeCA AS Select * from employee where
state-'CA'
```

In the above query, the schema is gathered from the select query. This is useful when creating sub-tables with a limited number of columns and data. Please note it doesn't support external tables.

HIVE TRANSACTIONS

Hive supports transactions with inserts, updates, and deletes in Hive 1.0.0 and above. ACID stands for Atomicity, Consistency, Isolation, and Durability as explained below:

- **Atomicity:** It guarantees that each transaction is represented as a unit operation. The whole operation is treated as a unit operation, meaning an operation can either completely succeed or completely fail; it does not leave partial data behind.
- **Consistency:** Data written to the database is consistent and visible to other operations.
- **Isolation:** Database transactions can happen concurrently. Isolation is a critical characteristic to make sure independent operation will not impact other operations.

- **Durability:** It guarantees that once a transaction has been completed (committed), it is preserved in the database in the case of a system failure (e.g., power outage or crash).

Hive supports ACID transactions, meaning a different application can read and write on the same table and same partition without interfering with each other. Update, delete, and merge operations on Hive tables are supported via ACID transactions. But there is a limitation for using a table with the ACID transactions. Currently, only the ORC format supports transaction operations in Hive. The table must be bucketed to use transactional ACID and transactions are allowed on the managed table only. We need to set transaction the manager *org.apache.hadoop.hive.ql.lockmgr.DbTxnManager* to work with the ACID tables. There is a possibility that the above limitations will be omitted in a future release.

ENABLE TRANSACTIONS

Insert, update, and delete are not possible without enabling ACID transactions in Hive. We need to configure the primary client and server configuration to allow ACID transactions in Hive, as explained below.

Client Side

- `hive.support.concurrency(default false)`: Sets concurrency as `true` to enable the concurrence operation in Hive
- `hive.exec.dynamic.partition.mode(default strict)`: sets the dynamic partition mode to nonstrict.
- `strict`: must specify at least one static partition
- `nonstrict`: all partitions are allowed to be dynamic
- `hive.txn.manager`: Set to *org.apache.hadoop.hive.ql.lockmgr.DbTxnManager* as part of turning on Hive transactions.

Server Side

- `hive.compactor.initiator.on(default false)`: Set value to true to turn on Hive transactions.
- `hive.compactor.worker.threads`: a positive number on at least one instance of the Thrift Metastore services.

Note: To support ACID writes, i.e., insert, update, and delete, the table has to enable transactions by setting the property "`transactional=true`." There is no option to convert the transaction table to a non-transactional table and vice versa.

INSERT VALUES

Same as RDBMS, SQL Hive also allows you to insert records using SQL insert values, as shown below:

```
CREATE TABLE employee (name VARCHAR(164), age INT,
salary DECIMAL(3, 2))
  CLUSTERED BY (age) INTO 3 BUCKETS STORED AS ORC;

INSERT INTO TABLE employee VALUES
('emp1','23','123.32'), ('emp2','24','128.42');
```

As shown in the above example, each value set in the value expression is inserted into the Hive table. In the current release, inserting into only a few columns is not supported yet. Therefore, while inserting records, it is mandatory to provide all table columns in the insert query. Below are some more insert values queries:

```
CREATE TABLE employee (name VARCHAR(164), age INT,
salary DECIMAL(3, 2))

PARTITIONED BY (state STRING) CLUSTERED BY (age) INTO 3
BUCKETS STORED AS ORC

INSERT INTO TABLE employee PARTITION (state = CA')
VALUES ('abc', '25', '123.54');

INSERT INTO TABLE employee PARTITION (state)
VALUES ('abc',  25', '123.54','CA');

INSERT INTO TABLE employee
VALUES ('abc', '25', '123.54','CA');
```

UPDATE

Hive query using the `UPDATE` statement to update one or more columns of a table when an ACID transaction is enabled.

```
UPDATE tablename SET column = value [, column = value ...]
[WHERE expression]
```

Below are some constraints on the UPDATE statement:

- The UPDATE statement doesn't update partition and bucket columns.
- UPDATE statements are auto-commit.
- The UPDATE statement supports arithmetic operators, UDFs, casts, and literals.
- Subqueries are not supported in the UPDATE statement.
- Available on + Hive 0.14

Note: You should disable `hive.optimize.sort.dynamic.partition` while making an update statement. It will provide more efficient execution plans.

```
Set hive.optimize.sort.dynamic.partition=false
```

DELETE

Hive queries use the DELETE statement to delete one or more rows of a table when an ACID transaction is enabled. Below are some constraints on the DELETE statement:

- Delete statements are auto-commit.
- Available on + Hive 0.14

```
DELETE FROM tablename [WHERE expression]
```

MERGE

Hive uses the MERGE statement to insert, update, or delete based on the JOIN condition when the transaction is enabled. Below is the standard syntax:

```
MERGE INTO <target table> AS T USING <source
expression/table> AS S
ON <boolean expression1>
WHEN MATCHED [AND <boolean expression2>] THEN UPDATE
SET <set clause list>
WHEN MATCHED [AND <boolean expression3>] THEN DELETE
WHEN NOT MATCHED [AND <boolean expression4>] THEN
INSERT VALUES<value list>
```

A MERGE example is as follows:

```
MERGE INTO employee1 as emp1 USING employee2 as emp2
ON emp1.id = emp2. id
WHEN MATCHED and emp1.status<> 'Y' THEN UPDATE SET date
emp2.date>
WHEN MATCHED and emp1.status = 'Y' THEN DELETE
WHEN NOT MATCHED THEN INSERT VALUES (emp2.id,
emp2.name, emp2.date, emp2.status);
```

Below are some constraints on the `UPDATE` statement:

- Merge statements are auto-commit.
- Available on Hive 2.2+
- Merge statement allow actions on target tables based on JOIN result outcomes.

LOCKS

Locks were introduced into Hive to support the ACID property in concurrency scenarios. The lock was introduced in Hive 0.7.0 and upgraded to the new lock manager in version 0.13.0. Lock helps to support concurrent read and write. In the current scenario, only implicit locks acquire support. There are two types of locks:

1. Shared lock(S): This is a shared lock to allow sharing concurrently. It is acquired while reading a table or partition.
2. Exclusive lock(X): This is acquired while doing an update/modify operation.

Lock Behavior on Tables

1. Lock behavior on non-partitioned tables: For the non-partitioned table, the S lock is acquired for a reading operation, where the X lock is acquired for all other operations, such as insert and modify.
2. Lock behavior of partitioned tables: For partitioned tables, the S lock is acquired for reading operations, where the X lock is acquired for all other operations. In case of a modification on the newer partitions, only the S lock is acquired, whereas if the change applies to all partitions, the X lock is acquired on the table.

Table 9.2 Hive locks acquired

Hive Command	Locks Acquired
`select .. T1 partition P1`	S on T1, T1.P1
`insert into T2(partition P2) select .. T1 partition P1`	S on T2, T1, T1.P1 and X on T2.P2
`insert into T2(partition P.Q) select .. T1 partition P1`	S on T2, T2.P, T1, T1.P1 and X on T2.P.Q
`alter table T1 [rename| add cols | replace cols| change cols | concatenate]`	X on T1
`alter table T1 touch partition P1`	S on T1, X on T1.P1
`alter table T1 set [serdeproperties| serializer | file format | tblproperties]`	S on T1
`alter table T1 partition P1 concatenate`	X on T1.P1
`drop table T1`	X on T1

Lock Manager

Hive added a new Lock Manager in v 0.13.0 to manage transaction locks on row-level ACID semantics. Lock Manager persists all transactional and locks information in the Hive Metastore. The lock manager uses a regular interval heartbeat to identify any stale lock and transaction. If the Metastore doesn't receive any heartbeat within a specified time (configured in *hive.txn.timeout*), then the lock/transaction gets aborted. Below is the command to show a list of locks on the table:

```
SHOW LOCKS <table_name>;
```

HIVE SELECT QUERY

The following is the generic `SELECT` syntax:

```
SELECT [ALL | DISTINCT] select_expr, select_expr, ...
  FROM table_reference
```

```
  [WHERE where_condition]
  [GROUP BY col_list]
  [ORDER BY col_list]
  [CLUSTER BY col_list
    | [DISTRIBUTE BY col_list] [SORT BY col_list]
  ]
 [LIMIT [offset,] rows]
```

The Hive select queries are similar to RDBMS. `table_reference` can be a regular table, a view, a subquery, or a join construct. The select query in Hive is similar to the RDBMS SQL query, which uses "`SELECT` what fields `FROM` which table `WHERE` condition."

```
Hive>SELECT firstname, lastname from employee where
state='CA'
Joe Hans
Sam Rough
```

SELECT BASIC QUERY

The following shows the database list of all databases available in Hive:

```
Hive> SHOW databases;
```

We can choose a database to execute a query either by adding the database name as a prefix or using the `USE database` command

```
Hive> USE abc; where database name abc.
Hive> SELECT query_specifications;
Or
Hive> SELECT abc.query_specifications;
```

The `USE` command sets the database for all subsequent HiveQL statements.

WHERE Condition

The `where` condition restricts the query to return a result on the `where` condition. For example, the following query provides a list of California employees.

```
Hive>SELECT firstname, lastname, state from employee
where state='CA'
Joe Hans CA
Sam Rough CA
```

Distinct Expression

The `distinct` column removes duplicate records and displays unique rows:

```
Hive>SELECT distinct firstname, lastname, state from
employee;
Hive>SELECT distinct firstname state from employee;
Hive>SELECT distinct * from employee;
```

Limit

The limit expression for selecting a query restricts the number of records:

```
Hive>SELECT firstname, lastname, state from employee
limit 1;
Joe Hans CA
```

HIVE QL FILE

Hive can execute queries from files by using the option `-f`, as shown below from the console panel:

```
$ hive -f /usr/joe/employee.hql or
>hive source /usr/joe/employee.hql
```

Hive uses JSON (JavaScript Object Notation) in complex type columns, such as collection, array, and structs.

```
Create table if not exists employee(
Name STRING,
Emails ARRAY<String>,
Previous_company<STRING,STRING>,
Address struct<street:STRING, city:STRING,zip:INT>
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
LOCATION '/usr/joe/employee'
```

HIVE SELECT ON COMPLEX DATATYPES

Array

An array is an ordered collection of the same type elements. The Hive select query can be used to retrieve array column values:

```
Hive>SELECT name, emails from employee;
    Joy smith   [abc@x.com,cde@y.com]
Hive>select name,emails[0] from employee;
    Joy smith   abc@x.com
```

Map

The Hive select query can be used on the map data type. Map is an unordered key-value pair collection. The below query shows the select query on the map data type.

```
Hive> Select name, Previous_company from employee;
      Joe smith, {"comp1":"abc","comp2":"tbe"}
Hive> select name,previous_company["comp1"] from employee;
      Joe smith   abc
```

Structs

In contrast to an array, a struct is a collection of elements of different types.

```
Hive> select name,address from employee;
    Joy smith {"street":"123
    Michagham","city":"SF","zip:789123}
Hive> select name,address.city from employee;
    Joe smith SF
```

ORDER BY AND SORT BY

The `order by` query results in either a descending or ascending order; hence, it uses only one Reducer to sort all the records, which might take a long time for an extensive database.

```
Hive> Select * from employee order by empid;
```

`SORT BY` only orders the data in each Reducer locally, which significantly improves the performance. `SORT BY` could ascending or descending (the default sorting order is ascending (ASC)).

```
SORT BY COLUMN_EXPR ( ASC | DESC )
Hive>Select * from employee sort by empid DESC;
```

`SORT BY` sorts the data for each Reducer, whereas `order by` orders data across all Reducers, hence, all the data is in order, but `SORT BY` only guarantees a partial order (per Reducer). In the case of more Reducers, `order`

by shuffles the data across the Reducers to bring all data into one Reducer. Thus, it impacts the performance for large datasets. However, SORT BY sorts data per Reducer locally, which improves the performance significantly. Let's assume there are N Reducers, then each Reducer sorts locally, which can have overlapping ranges of data. That means we will have N or more sorted files with overlapping ranges.

DISTRIBUTE BY AND CLUSTER BY

Using DISTRIBUTE BY <ColumnName> causes rows of data to get distributed to multiple Reducers based on columns. That means the same column value always goes to the same Reducer. It improves performance and optimizes the output data in case of partitioned tables.

DISTRIBUTE BY ensures each Reducer gets a non-overlapping range data, but it doesn't guarantee sorted or ordered data in each Reducer. In the case of N Reducers, DISTRIBUTE BY creates N or more non-overlapping ranges files. The following example shows the distribution with the included SORT BY orders the records in each Reducer without a non-overlapping data range:

```
select name,address from employee DISTRIBUTE BY name SORT
BY name ASC
```

CLUSTER BY <ColumnName> combines DISTRIBUTE BY and SORT BY. That means each of N Reducers gets non-overlapping ranges, and then each Reducer sorts locally. CLUSTER BY provides global ordering on the table for columns without compromising the performance. CLUSTER BY is the optimized, scalable version of ORDER BY.

```
select name,address from employee CLUSTER BY name
```

GROUP BY AND HAVING

The group by clause groups the data from records based on one or more columns. It is used in conjunction with the aggregate functions (like SUM, COUNT, MIN, MAX, and AVG). For example, we can get an average salary of employees by state.

```
Hive>Select state, avg (salary) from employee group by
state;
```

`Having` is always used in combination with `group by` and places more restrictions on `group by` (for example, a list of states whose average salary is higher than two thousand).

```
Hive> Select state, avg (salary) from employee group by
state having avg(salary)>2000
```

BUILT-IN AGGREGATE FUNCTIONS

Hive supports the aggregate function, which aggregates a single value from multiple rows, e.g., `count (*)` from employee tables.

```
Hive> select count (*) from employee;
```

This code returns the total employee count.

The aggregation function collects data to get more specific details on a group of data based on a given condition. Hive provides various in-built aggregate functions, such as `max`, `count`, `min`, and `avg`. The latest Hive release does support advanced aggregate functions such as `GROUPING`, `SETS`, `ROLLUP`, and `CUBE`. The essential aggregate functions work with the `GROUP BY` condition.

Table 9.3 Built-in Aggregate Functions

Function: return type	Description
Count (*): BIGINT	Get number of rows
Sum (col): Double **Sum (distinct col): double**	Get the sum of column value Get the sum of distinct column value
Avg (col): double: double **Avg (distinct col): double**	Get average of column value Get average of distinct column value
Min (col): double **Max (col): double**	Get minimum value of column value Get maximum value of column value
var_pop(col): double **var_samp(col): double**	Get the variance of set of numbers of column Get the sample variance of set of numbers of column

(continued)

Function: return type	Description
stddev_pop(col): double **stddev_samp(col): double**	Get the standard deviation of set of number of column Get the sample standard deviation of set of number of column
covar_pop(col1, col2): double **covar_samp(col1, col2)**	Get the covariance of a set of numbers of columns `col1` and `col2` Get the sample covariance of a set of numbers of columns `col1` and `col2`
corr(col1, col2): Double	Get the correlation of two sets numbers
percentile(value_int, p): Double	Get percent of `value_int` at `p`
collect_set(col): ARRAY	Get unique element from a column

ENHANCED AGGREGATION

Hive enhances aggregator functions, which are introduced in Hive version 0.10.0.

Grouping Sets

`GROUPING SETS` is a superset of the traditional group that allows more than one `GROUP BY` option against the same sets of table data. `GROUPING SETS` simplifies multiple groups when combined with the `UNION` operation in a simple one-line statement. We can specify `GROUPING SETS` as `UNION ALL` of many groups. Let's say we have the `UNION` of two groups by a query:

```
SELECT A, B, SUM( C ) FROM tableA GROUP BY A, B
UNION
SELECT A, null, SUM( C ) FROM tableA GROUP BY A
Which can be easily represent using GROUPING SETS on below
query
SELECT A, B, SUM( C ) FROM tableA GROUP BY A, B GROUPING
SETS ( (A,B), A)
```

Cube and Rollup

CUBE and ROLLUP are always used with GROUP BY. CUBE generates all possible combinations of the column sets in the GROUP BY condition. For N aggregation columns, CUBE creates a 2N aggregation combination.

```
GROUP BY a,b,c WITH CUBE is equivalent to
GROUP BY a,b,c GROUPING SETS ( (a, b, c), (a, b), (b, c),
(a, c), (a), (b), (c), ( )).
```

ROLLUP used with GROUP BY aggregates at the hierarchy level of the dimension.

For example:

```
GROUP BY a, b, c, WITH ROLLUP is equivalent to
GROUP BY a, b, c GROUPING SETS ( (a, b, c), (a, b), (a), ( )).
```

TABLE-GENERATING FUNCTIONS

The aggregate function summarizes multiple records into a few summary records, whereas the table-generating function expands single-row records into numerous records. For example, in the "employee has email" column, an array of emails, we can expand "email" using the explode function:

```
Hive> select explode (emails) AS email from employee
    abc@x.com
    cde@y.com
```

Table 9.4 Hive Table Functions

Function: return type	Description
Explode (array column): N rows	Explodes number of elements in arrays into rows
Explode (map column): N rows	Explodes map <key-value> pair to row.
json_tuple(jsonValue, p1, p2, ..., pn): tuple	Converts JSON to N tuple
parse_url_tuple(url, partname1, partname2, ..., partna meN): tuple	Converts URL to N part tuple

(continued)

Function: return type	Description
stack(n, col1, ..., colM): N row	Converts M columns into N rows of size M/N each
posexplode (ARRAY<T> a): int,T	Explodes an array to multiple rows with additional positional columns of `int` type
inline(ARRAY<STRUCT<f1:T1,...,fn:Tn>> a): T1,...,Tn	Explodes an array of structs into multiple rows.

Examples

```
Explode(array)
Below example explode array of values into rows
Select explode (array('john','sam','messy')) as empName;
```

empName
john
sam
messy

```
Explode(map)
select explode(map('john',23,'sam',24,'messy',25)) as
(empName, age);
```

empName	age
john	23
sam	24
messy	25

```
posexplode (array)
select posexplode(array('john','sam','messy')) as (pos, name);
```

pos	name
0	john
1	sam
2	messy

```
inline (array of structs)
select inline(array(struct('john',23,date '2015-01-01'),
struct('sam',24,date '2016-02-02'))) as (name,age,date);
```

name	age	date
john	23	2015-01-01
sam	24	2016-02-02

```
stack (values)
select  stack(2,'john',23,date  '2015-01-01','sam',24,date
'2016-01-01') as (name, age, date);
```

name	age	date
john	23	2015-01-01
sam	24	2016-01-01

BUILT-IN UTILITY FUNCTIONS

Mathematical Functions

Hive provides various mathematical built-in functions. Table 9.5 shows some key mathematical built-in functions.

Table 9.5 Built-in Utility Functions

Function: return type	Description
round(DOUBLE a): double **floor(DOUBLE a): double**	Return the round of double
ceil(DOUBLE a):bigint **ceiling(DOUBLE a):bigint**	Return maximum bigint less than or equal to a
rand(), rand(INT seed): double	Return random value
pow(DOUBLE a, DOUBLE p):double power(DOUBLE a, DOUBLE p): double	Return a^p
sqrt(DOUBLE a):double **sqrt(DECIMAL a):double**	Return the square root of a

(continued)

Function: return type	Description
`abs(DOUBLE a): double`	Return absolute value of a
`bin(BIGINT a): String`	Return number in binary format
`pmod(INT a, INT b): Int` `pmod(DOUBLE a, DOUBLE b):double`	Return the positive value of `a mod b`
`factorial(INT a): bigint`	Return factorial of a

COLLECTION FUNCTIONS

Hive provides various collections of built-in functions.

Table 9.6 Hive Function Collections

Function: return type	Description
`size(Map<K.V>): int` `size(Array<T>): int`	Returns a number of records in Map and Array
`map_keys(Map<K.V>: array<K>`	Returns an array
`map_values(Map<K.V>): array<K>`	Returns a random array of map keys
`rray_contains(Array<T>, value): boolean`	Returns `true` if the value is in Array
`sort_array(Array<T>): array<t>`	Returns a sorted array in ascending order

DATE FUNCTIONS

Hive provides various built-in date functions.

Table 9.7 Date Functions

Function: return type	Description
`from_unixtime(bigint unixtime[, string format]): string`	Returns Converted date format string from epoch time
`unix_timestamp(): bigint`	Gets current UNIX timestamp in seconds

Function: return type	Description
`unix_timestamp(string date): bigint` `unix_timestamp(string date, string pattern): bigint`	Returns epoch timestamp for given date It provides the date string pattern as second arguments
`to_date(string timestamp): date`	Converts string to date
`year(string date): int`	Returns year part from string date
`month(string date): int`	Returns month part of date string
`hour(string date): int`	Returns hour part of date string
`minute(string date): int`	Returns minute part of date string
`second(string date): int`	Returns second part of date string
`weekofyear(string date): int`	Returns the week number
`extract(field FROM source): int`	Returns specific field of the extracted date (the month from "2020-11-30" yields "11")
`from_utc_timestamp({any primitive type} ts, string timezone): timestamp`	Converts time to specific zone
`to_utc_timestamp({any primitive type} ts, string timezone): timestamp`	Converts to UTC time
`current_date: date`	Returns current date
`current_timestamp: timestamp`	Returns current timestamp
`add_months(string start_date, int num_months, output_date_format)`	Adds month `add_months('2020-01-01', 1)` returns '2020-02-01'.
`last_day(string date): String`	Returns last day of the month
`next_day(string start_date, string day_of_week): string`	Returns next day from start date

(continued)

Function: return type	Description
trunc(string date, string format): string	Returns date truncated to the unit specified by the format trunc('2020-09-18', 'MM') = 2020-09-01
months_between(date1, date2): double	Returns number of months between dates date1 and date2
date_format(date/timestamp/ string ts, string fmt):string	Converts a date/timestamp/string to a value of string in the format specified by the date format fmt

CONDITIONAL FUNCTIONS

Hive provides various date built-in conditional functions.

Table 9.8 Hive Conditional Functions

Function: return type	Description
if(boolean testCondition, T valueTrue, T valueFalseOrNull):T	Returns `valueTrue` when condition is `true` else returns `valueFalseOrNull`
isnull(a): boolean	Returns `true` if `a` is `NULL` and `false` otherwise
isnotnull (a): boolean	Returns `true` if `a` is not `NULL` and `false` otherwise
nvl(T value, T default_value):T	Returns default value if value is `null` else returns value
COALESCE(T v1, T v2, ...):T	Returns the first v that is not `NULL`, or `NULL` if all v's are `NULL`
CASE a WHEN b THEN c [WHEN d THEN e]* [ELSE f] END : T	When `a = b`, returns `c`; when `a = d`, returns `e`; else returns `f`.
CASE WHEN a THEN b [WHEN c THEN d]* [ELSE e] END:T	When a = true, returns b; when c = true, returns d; else returns e.
nullif(a, b):T	Returns `NULL` if `a=b`; otherwise returns `a`
assert_true(boolean condition):void	Throw an exception if 'condition' is not true, otherwise returns `null`

STRING FUNCTIONS

There are many built-in string utility functions.

Table 9.9 Hive String Functions

Function: return type	Description
`ascii(string):string`	Returns an ASCII character
`base64(binary bin): string`	Converts the argument from binary to a base 64 string
`character_length(string str): int`	Returns the number of UTF-8 characters contained in str
`chr(bigint|double A): string`	Returns the ASCII character having the binary equivalent to A
`concat(string|binary A, string|binary B...): string`	Returns the string or bytes resulting from concatenating the strings or bytes passed in as parameters in order
`context_ngrams(array<array<string>>, array<string>, int K, int pf): array<struct<string,double>`	Returns the top-k contextual N-grams from a set of tokenized sentences
`concat_ws(string SEP, string A, string B...):string`	Like `concat()` above, but with custom separator SEP
`decode(binary bin, string charset): string`	Decodes the first argument into a string
`elt(N int,str1 string,str2 string,str3 string,...): String`	Return string at index number
`encode(string src, string charset): binary`	Encodes the first argument into binary
`length(string):INT`	Gets the length of string
`reverse(string):STRING`	Returns the reverse of string
`substr(string,index):String`	Returns substring of string starting from index

(continued)

Function: return type	Description
substr(string,start,length) :String	Substring starting from `start_index` to the length
upper(string):STRINg	Returns uppercase characters of the string
lower(string):String lcase(string):String	Return lowercase characters of the string
trim(string):STRING	Removes extra whitespace from both sides
ltrim(string):String	Removes extra whitespace from left side
rtrim(string):String	Removes extra whitespace from right side
regexp_replace(string, regexp, replacementString): String	Replaces the string based on regular expression match
regexp_extract(subject, regex, index): String	Fetches substring for the index match using the `regex_pattern`
parse_url(url, partname, key): String	Fetch the specified part from a URL
space(n):String	Returns n spaces
repeat(string, n): String	Returns string in n time
lpad(string, len, pad): String	Adds `len`, `pad` on left of string
rpad(string, len, pad): String	Adds `len`, `pad` on right of string
split(string, pattern):ARRAY<String>	Splits string on pattern
find_in_set(string, commaSeparated String):INT	Returns the index of the comma-separated string where string is found, or `NULL` if it is not found.

Function: return type	Description
`locate(substr, str, pos]):Int`	Returns the index of `str` after `pos` where `substr` is found
`instr(string, substr):Int`	Returns the index of `string` where `substr` found
`In_file(string, filename):boolean`	Returns `true` if `s` appears in the file named `filename`.

HIVE QUERY LANGUAGE-JOIN

One of Hive's significant advantages is its use of a simple `JOIN` query to build a complex Map Side join or Reduce Side join. The developer does not have to worry about making a complex MapReduce join. He has to join queries and let Hive build the MapReduce join automatically. The following are the generic `JOIN` select query expressions.

Inner Join

This type of join returns the typical key-value as described.

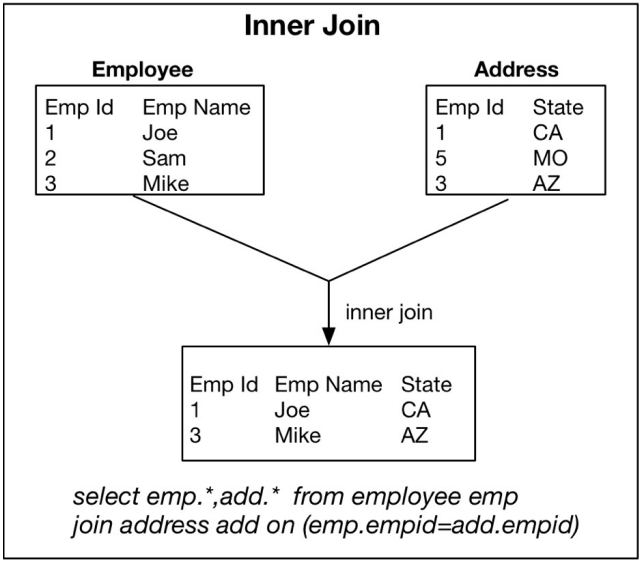

FIGURE 9.1 Hive inner join

```
SELECT * FROM tableA at JOIN tableB bt ON at.id=bt.id JOIN
tableC ct ON (bt.id=ct.id)
```

Note: The Hive version 0.13.0+ onwards `JOIN` also supports the implicit `JOIN`:

```
SELECT * FROM tableA at, tableB bt, tableC ct WHERE at.id
= bt.id AND bt.id = ct.id
```

Left Outer Join

The left join returns all the data from the left table regardless of whether the key data is available in the right table. It will display the NULL value for the non-available value in the right table.

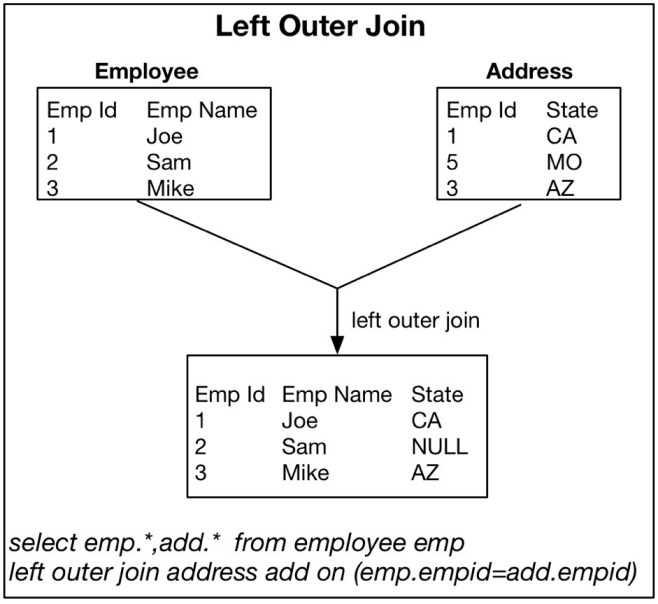

FIGURE 9.2 Hive left outer join

Right Outer Join

The right outer join is the reverse of the left outer join, meaning it keeps all the left table records and NULL for fields missing in the table on the left.

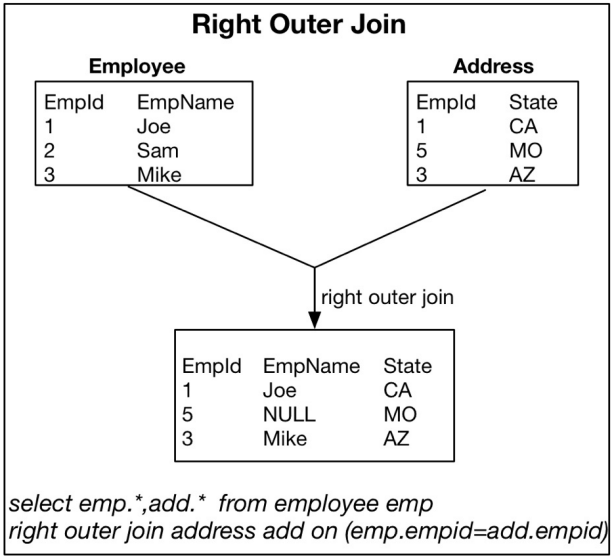

FIGURE 9.3 Hive right outer join

Full Outer Join

A full outer join returns all the records from both tables regardless of a key match. It uses NULL if the value is not available in any of the fields.

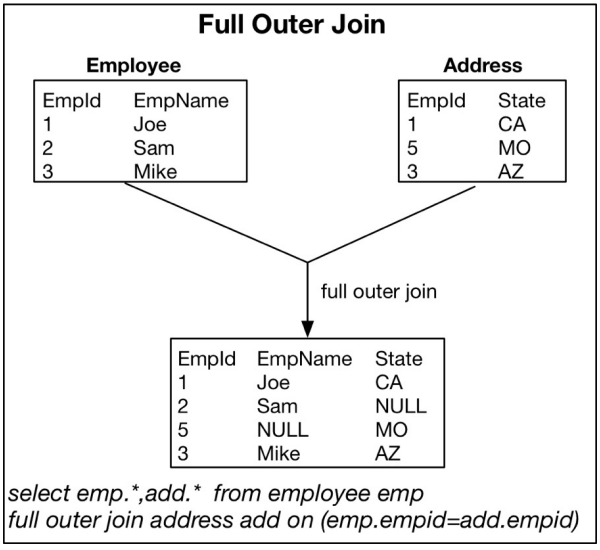

FIGURE 9.4 Hive full outer join

Left Semi Join

Hive doesn't support `IN EXISTS` between two tables, so you cannot use `IN` to get a list of employees:

```
Hive> SELECT emp.* from employee emp where emp.empid IN
(select empid from address)
```

To achieve the `IN` scenario, we have to use the `left semi join`:

```
Hive> Select emp.* from employee emp left semi join
address add on emp.empid=add.empid;
```

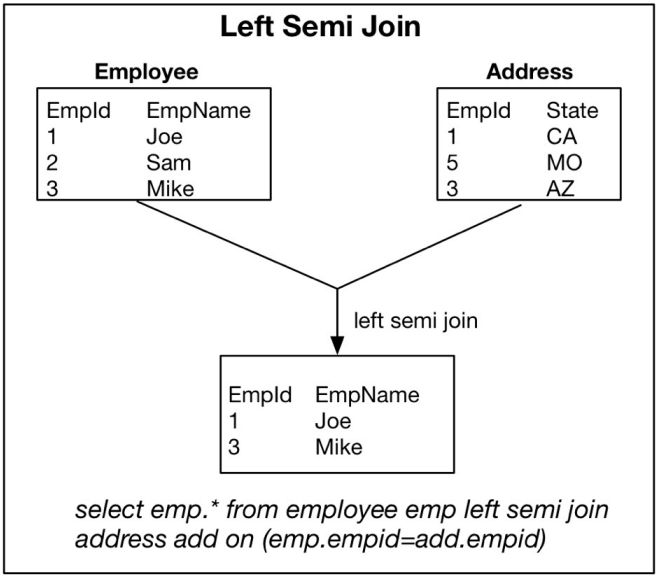

FIGURE 9.5 Hive left semi join

Map Side Join

Hive can optimize the `JOIN` query to identify and load the smaller table into memory first to compare it with each of the Mapper's large data tables. It improves memory allocation and reduces the steps involved in joining. In an earlier version before 0.7, we defined `mapjoin` by embedding it in the table:

```
Hive>SELCT /*+MAPJOIN(add) */ emp.*,add.* from employee
emp join address add on (emp.empid=add.empid);
```

Embedded hint still works, but not it is deprecated on Hive v0.7:

```
hive> set hive.auto.convert.join=true;
```

You can also set the file size property to optimize `mapjoin`:

```
hive.mapjoin.smalltable.filesize=25000000(default value in
bytes)
```

`Mapjoin` also optimizes the bucketed table if the data bucketed for every table on the keys used in the `JOIN` condition says `empid`. Hive can join buckets by avoiding individual records comparison in each bucket, which further improves the performance. We can use this feature by setting the following property.

```
set hive.optimize.bucketmapjoin=true;
```

Note: Before release 0.11, to execute `MAPJOIN`, we needed to provide a `MAPJOIN` optimizer hint:

```
Hive>SELCT /*+MAPJOIN(add) */ emp.*,add.* from employee
emp join address add on (emp.empid=add.empid);
```

Or we can set it as shown below:

```
set hive.auto.convert.join=true; (default false)
```

In release 0.11, `hive.auto.convert.join` changed the default to `true`, which avoids using `MAPJOIN` to optimize the `JOIN` query. If the auto-join condition is enabled, the `MAPJOIN` hint is not required in the `MAPJOIN` query.

Union All

`UNION ALL` is used to combine two or more sub-queries of the same number of columns with the same type and position. It may be used in the same source tables to remove the complex `where` clauses.

```
Select * from (select a.empid, a.empname from employee a
UNION ALL
Select a.empid, a.empname rom employee b) emp;
```

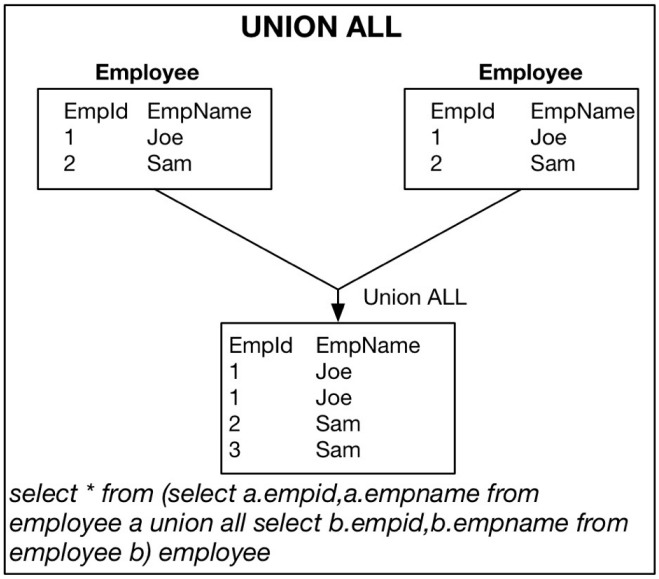

FIGURE 9.6 Hive union all

CHAPTER 10

FILE FORMAT

FILE FORMAT CHARACTERISTICS

Hadoop uses various file formats to persist data in the Hadoop file system (HDFS). It supports multiple Hadoop specific file formats such as Parquet, Seq, and ORC. Each file format has a particular way of storing data and characteristics that can help you decide if it is a suitable file format for your Big Data application. Before discussing file formats, let's first consider their characteristics.

COLUMNAR FORMAT

Columnar Format is one of the key characteristics of many file formats. Columnar format keeps column details and records so you can select individual column values without scanning all of the data horizontally. Columnar format allows for the retrieval of the necessary column only, which optimizes I/O calls, making the query fast.

Let's say you have a large table with numerous columns (< 100 K), and you are trying to aggregate on that query or selecting a few columns. If this table is in row format, it has to scan and retrieve all columns when you are only looking for a few columns. In columnar format, you would only fetch the selected columns.

Figure 10.1 shows how columnar format only chooses selected columns to retrieve results.

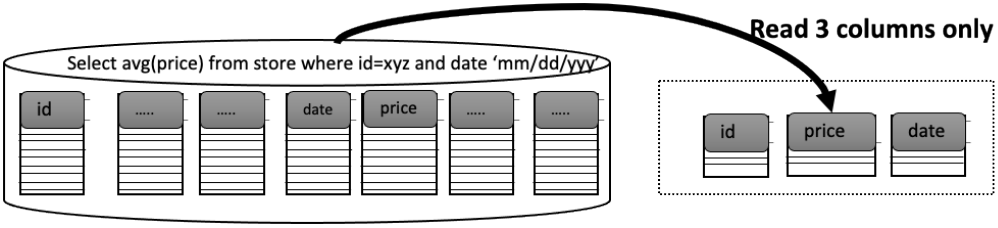

FIGURE 10.1 Column-oriented data processing

Figure 10.2 shows row format reads all columns, which in turn impacts the I/O calls and performance.

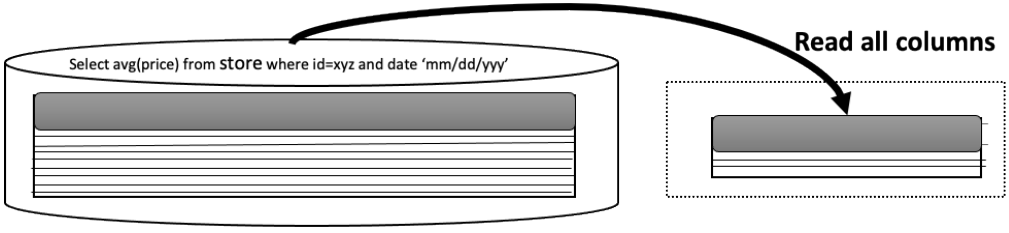

FIGURE 10.2 Row-oriented format data processing

In columnar format, records get partitioned horizontally and vertically. Slicing the records improves performance on wide columnar tables if you look for a few columns in the records.

Note: In columnar format, you should select only the required columns, not all the columns (e.g., `select col1,col2,col3,...,colN from table where column like query` is highly recommended. Try to avoid `select * from table where column like query`).

Columnar-oriented file formats compress the same column data together, which improves the storage optimization as the corresponding column data is of a similar data type. In some scenarios, if the requirement is to retrieve all columns, columnar format becomes an expensive option compared to row format. In contrast, columnar file format gives better performance when we perform analytic queries that require only a subset of columns on large datasets. Table 10.1 shows how columnar format slices each record in a columnar manner.

Table 10.1 Row-Oriented vs. Columnar-Oriented

Row-oriented : Rows stored sequentially

Id	FirstName	LastName	State	Zip	Dept
1	John	Jack	CA	11234	Hist
2	Benny	Tier	CA	12345	Eco
3	Ruby	Kaiser	MN	34567	Sci

Column-oriented : Column stored separately

Id	FirstName	LastName	State	Zip	Dept
1	John	Jack	CA	11234	Hist
2	Benny	Tier	CA	12345	Eco
3	Ruby	Kaiser	MN	34567	Sci

Row-Oriented vs Columnar-Oriented

Benefits of Columnar Formats

Optimized Read

Column values are grouped in a columnar format, which can be distributed across different nodes. Let's say we have a table with three columns that have values, as shown in Table 10.2.

Table 10.2 Example Data

Column Name	Col1	Col2	Col3
Value	Col1val1	Col2val1	Col3val1
Value	Col1val2	Col2val2	Col3val2

In this case, each column value is grouped and stored into various distributed data nodes.

```
Col1,Col1Val1,Col1Val2 stored into disk 1
Col1,Col1Val1,Col1Val2 stored into disk 2
```

If you would like to retrieve column one, i.e., a `col1` value, you have just to read the disk stored column `col1` to retrieve the result.

Let's discuss another query: `SELECT COUNT(1) from employee where name = "abc"`.

In the standard non-columnar format, we have to scan each row, parse each column, and then extract the name-value equal to "abc" count for all values. In columnar format, you need to examine the "name" column value and get the count with the name equal to "abc."

Optimized Compression

The same type of column data is stored together with the same column type. Because the data with the same type has similar sizes, this tends to result in further storage optimization. It could support both file-level compression and block-level compression.

Note: File-level compression compresses the entire file the same way as files are compressed in Linux. Block-level compression depends on the Hadoop file format, and it compresses individual pieces of the HDFS block data within the file. It gives more flexibility on a splittable compressed block, such as Snappy.

Conclusion

In columnar format, since the stored data is sequentially grouped by a column, computation is more efficient when selecting a few columns in a query. Columnar format optimizes high computing costs because unwanted columns filter out. In the case of analyzing a few selected columns, the columnar file format is a wise choice.

SCHEMA EVOLUTION

Schema evolution allows us to write new data with a changed schema and supports backward compatibility with the old schema(s). There are many use cases where the schema of the underlying data keeps changing over time. In this scenario, we need to specify a file format that handles frequent schema changes such as adding, deleting, and modifying the schema's fields.

If you have few data with a specific schema and would like to change the schema, you can efficiently re-write the aggregated data with the new schema. However, what happens if you have terabytes of data and want use a new schema? Schema evolution is a challenging issue because re-writing massive amounts of data with a new schema is an expensive job.

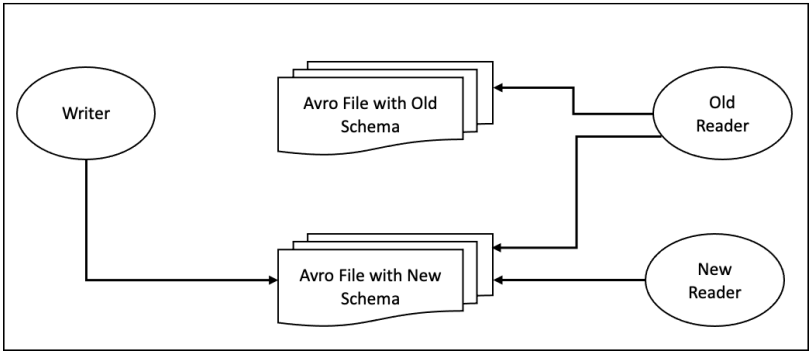

FIGURE 10.3 Schema evolution

SPLITTABLE

Hadoop stores data in blocks that are distributed across the clusters to advocate parallel distributed processing. In the Hadoop environment, data can grow exponentially, and this large amount can impact performance if a file's data is not spread across the cluster. It is crucial that the file format be splittable into multiple chunks and spread across the cluster data node.

In general, not all file formats are splittable; nonsplittable file formats significantly impact performance in the Hadoop environment. For example, JSON and XML are not splittable, which means they cannot be easily divided into small chunks that can be handled independently. The splittable file format is a critical characteristic to consider before designing a data storage pipeline in the Hadoop environment.

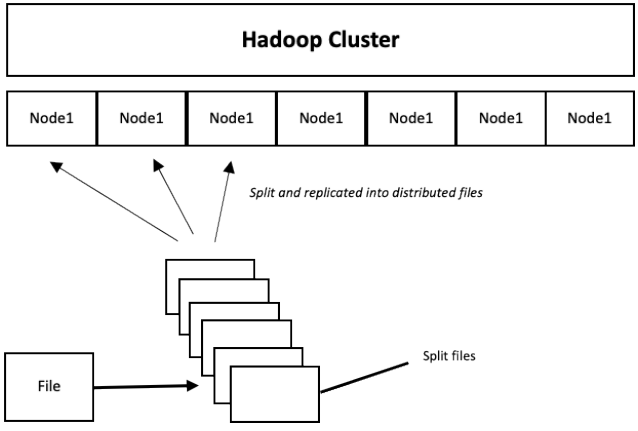

FIGURE 10.4 Hadoop splittable example

COMPRESSION

Hadoop stores large datasets into blocks, and those blocks are stored on different data nodes across the cluster. Hadoop supports large datasets that require ample storage and require a higher I/O bandwidth. HDFS easily supports growing large datasets by scaled storage horizontally; however, it's the best to optimize data storage before moving to scaled storage. Hadoop provides various compression mechanisms to compress data in blocks, which not only optimize data storage, but also improve the I/O transfer bandwidth. Compression can also remove irrelevant or redundant data, making analysis and processing faster.

If you compress data, then it has to be decompressed, as well, while compute processing. Data decompression adds extra processing effort to computing processing; hence, it's still advisable to choose better trade-offs between storage and compute processing and network bandwidth.

File compression provides two key benefits to reduce storage costs and improve transfer rate across networks. When we talk about file format selection in the Hadoop environment, it's always advisable to consider the best compression format suitable for your application carefully.

Table 10.3 Compression Formats

Properties	Compression Format			
	.gzip	bzip2	lzo	snappy
File extension	.gzip	.bz2	.lzo	.snappy
Compression level	Medium	High	Average	Average
Speed	Medium	Slow	Fast	Fast
CPU usage	Medium	High	Low	Low
Is Splittable	No	No	Yes, if indexed	No

FILE FORMATS

Text

Text is a simple file format to store data; it writes into lines terminated with the newline character `\n`, as in UNIX. You need to understand the layout of the file to read it. Text supports file-level compression, such as a Bzip2 (a splittable file-level compression). Text is a very simple, structured, and lightweight format; however, it's relatively slow to read and write.

Sequence

Hadoop stores data in a fixed-sized block. If the files are large, the data will be split and stored into a small chunk of a block. If files are under the block size, they get directly stored into a block with unused space. Small files cause a significant performance problem and are costly.

Sequence files combine small files into fixed-size sequence files that can fit the HDFS block and optimized HDFS storage. The sequence file format helps to solve the small file problem in Hadoop.

A sequence file is a container format that consists of binary key-value pairs with attached metadata, where each value includes file content in binary. The sequence file format supports various compression codes, such as Snappy and LZO. The value byte of a sequence file is directly compressed by configured code without changing its structure. The sequence file can compress through the block-level.

Note: Record level compression will not compress the key, whereas block-level compression compresses key and value.

The sequence format is dependent on Hadoop MapReduce to solve the Hadoop MapReduce small file problem. The fundamental concept of the sequence file format to combine small files to a larger single file. Let's say we have more than 1000 small files in KB. If these are stored into the HDFS, NameNode keeps the meta details of each file. Each file occupies more blocks with unused space, creating pressure on the storage and network. Besides storing all the data separately, the sequence files can be combined to create a bigger file. This is equal to the block size for a fully utilized block capacity, as shown Figure 10.5.

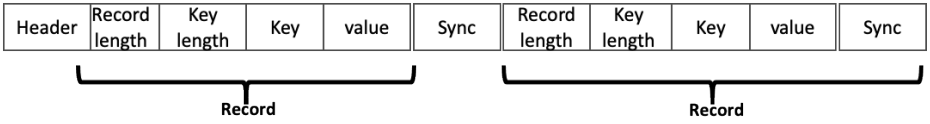

FIGURE 10.5 Sequence file format

Benefits

- Reduced overhead on NameNode by reducing the number of small files to manage
- It is splittable, hence, it find the best fits for the MapReduce program.
- It supports both block and record compression.

Record Compression

Record compression compresses each record's value and compressed codec store in the header. Figure 10.6 shows the values compressed in the sequence file meta detail, and the key doesn't get compressed.

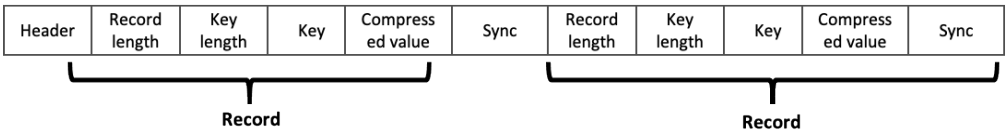

FIGURE 10.6 Record compression

Records combine with the configured parameters (`io.seqfile.compress.blocksize`). The block size is configured and combined with the records with blocks. Please note blocks in sequence don't have a direct relationship with the HDFS block. Once the block size reaches the configured size limit, it starts the next block, and the synchronization (Sync) flag is added between two blocks. Block compression combines multiple files; hence, it provides a better compression ratio than record compression, and it's the preferred option in the Hadoop environment.

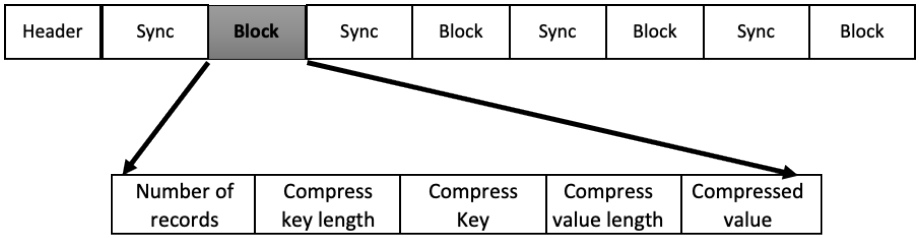

FIGURE 10.7 Block compression

Even record compression provides excellent compression ratio options in Hadoop MapReduce; however, it's highly dependent on the Hadoop environment and its API doesn't have language portability. It also doesn't support schema evolution.

Avro

The sequence file format is less popular because of its language dependency with Hadoop API and doesn't support schema reconciliation for frequently-changed schema. These challenges are solved by Avro data serialization, with its unique language-neutral and schema reconciliation characteristics. It provides cross-language portability, which means data written in Avro can be easily read and written using different languages without any specific dependency.

It works with the Hadoop environment and supports Plain Java, Pig, and Python without any specific dependency. The Avro file format keeps the metadata within the Avro data file and provides and independent schema called *avsc* for Avro.

Avro supports dynamic schema reconciliation, meaning we can start writing data with new schema while also supporting backward schema for old Avro data. You can rename, add, and delete the column field by defining a new schema without any impact on Avro data.

Avro supports rich data transfer.

Avro is not just string/int key-value data, but also supports various primitive data types such as int, long, Boolean, float, double, bytes, and string. It also supports various complex data types such as record, enum, arrays, map, union, and fixed.

Avro is an adaptable serialized file format data in Hadoop with a serialization and de-serialization framework. A schema is directly embedded inside Avro, which results in faster processing without any external mapping dependency. Its schema is defined on top of JSON, which supports interoperability and supports schema evaluation without much difficulty.

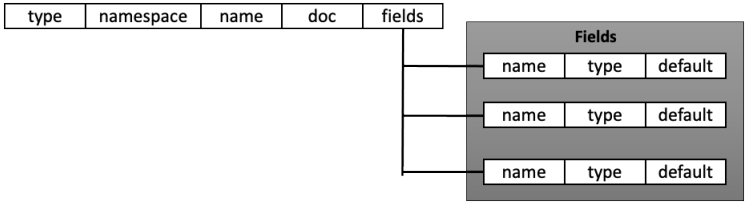

FIGURE 10.8 Avro file format

Avro data is stored with schema details and field mapping with each schema in the JSON structure. It is kept with the compression codec; hence, the files can be easily encoded and parsed by any program. If the schema was changed, the next latest Avro would generate with new Avro schema. These frequent schema changes were resolved efficiently, since both old and new Avro consist of their embedded schema. Avro uses a predefined schema to write data into a file for further processing.

Avro Schema

Avro schema uses JSON (JavaScript Object Notation), i.e., a lightweight text and data interchange format used to create Avro schemas. Below is a simple example of an Avro schema.

```
{
      "type": "record",
      "namespace": "com.myschema ",
      "name": "employee",
      "doc":"Employee detail",
```

```
     "fields": [
       { "name": "empname", "type": "string", "default"
: null },
       { "name": "dept", "type": "string", "default" :
null }

     ]
 }
```

The above Avro schema has the following key fields:

- **Type:** This field type is always be recorded for Avro. Type `record` has multiple fields defined.
- **Namespace:** This distinguishes one schema from other Avro schemas.
- **Name:** This is the schema name when combined with the namespace, and it uniquely identifies the schema. The schema field names must begin with [A-Za-z_], and subsequently contain only [A-Za-z0-9_]
- **Doc:** a description of the schema
- **Fields:** These define what field values and data types are contained in the fields.
- **Default:** The default value is filled in the case when there is no value for a particular field. It is a good practice to add a default value for the schema evolution. For example, if you add new fields old Avro doesn't have, this field will prefill with a default value.

Below are key features on Avro format:

Language portability

The Avro data is embedded into the schema in itself, which makes it language-independent.

Cross-Language Interoperability

Avro uses the JSON format, which is independent of language; hence, different languages (such as Java or Python) can communicate with Avro without any extra dependency.

Schema Evolution

There is a possibility that a schema gets changed during ingestion time. The schema change could be a column removed or a new column added on data records that can significantly impact the ingestion pipeline or retrieve queries.

Avro provides schema evolution to handle frequent changes in the schema's structure. Schema evolution allows for controlling different schema versions while reading or writing data with backward compatibility with the old schema.

Splittable and Compression

The splittable and compression are essential features on Hadoop, a distributed processing system. Avro is splittable and allows compression. By using the right compression codes in Avro, we can achieve efficient performance and storage capacity. The following code shows an example of writing an Avro file.

```
public class WriteToAvro {

    public static void main(String[] args) throws
    IOException {
    Schema schema = createSchema();
    DataFileWriter<GenericRecord> writer = new
    DataFileWriter<>(new GenericDatumWriter<>());
    writer.create(schema, new File("/home/avro/emp.
    avro"));

    // Create generic records and append to writer
    GenericRecord record1 = new GenericData.
    Record(schema);
    record1.put(0, "EmpName1");
    record1.put(1, "Dept1");
    writer.append(record1);

    GenericRecord record2 = new GenericData.
    Record(createSchema());
    record2.put(0, "EmpName2");
    record2.put(1, "Dept2");
    writer.append(record2);
    // Closed the existing writer
    writer.close();
    System.out.println("Successfully Created Avro file");
}

// Create Schema
    public static Schema createSchema() {
    List<Field> fields = new ArrayList<Field>();
    List<Schema> childSchemas = new ArrayList<Schema>();
    childSchemas.add(Schema.create(Schema.Type.NULL));
```

```
    childSchemas.add(Schema.create(Type.STRING));
    Schema avroSchema = Schema.createUnion(childSchemas);
    fields.add(new Field("empname", avroSchema, null,
    ""));
    fields.add(new Field("dept", avroSchema, null, ""));

    Schema schema = Schema.createRecord("Employee",
    "Generated Schema", null, false);
    schema.setFields(fields);
    return schema;
  }
}
```

The following sample code reads an Avro file:

```
public class ReadAvro {
    public static void main(String[] args) throws
    IOException {
        InputStream input = new FileInputStream(new
        File("/home/avro/emp.avro"));
        DataFileStream<GenericRecord> readerStream
        = new DataFileStream<>(input, new
        GenericDatumReader<>());
        while (readerStream.hasNext()) {
            GenericRecord record = readerStream.next();
System.out.println("Record:"+record.toString());
        }
    }
 }
```

Avro can be easily extrapolated in a Hive table environment as an external table. First, we create a schema (say, employee, as in the following example).

```
{
   "type":"record",
   "name":"Employee",
   "doc":"Generated Schema",
   "fields":[
      {
         "name":"empname",
         "type":[
            "null",
            "string"
         ],
```

```
            "default":""
        },
        {
            "name":"dept",
            "type":[
                "null",
                "string"
            ],
            "default":""
        }
    ]
}
```

Place them in the HDFS location:

```
hadoop fs -put employee.avsc /user/temp/employee/schema/
```

Let's assume the dataset is in the following HDFS location:

```
/user/temp/employee/data/
```

In that case, we can create a Hive table pointing to the Avro data location with the Avsc schema as shown below

```
hive> create external table employee stored as avro
location '/user/temp/employee/data/' TBLPROPERTIES ('avro.
schema.url' = 'hdfs:///user/temp/employee/schema/employee.
avsc');
OK
 Time taken: 0.119 seconds
```

We can describe table as follows:

```
hive> show create table  employee;
OK
CREATE EXTERNAL TABLE 'employee'(
  'empname' string COMMENT'',
  'dept' string COMMENT '')
ROW FORMAT SERDE
  'org.apache.hadoop.hive.serde2.avro.AvroSerDe'
STORED AS INPUTFORMAT
  'org.apache.hadoop.hive.ql.io.avro.
AvroContainerInputFormat'
OUTPUTFORMAT
  'org.apache.hadoop.hive.ql.io.avro.
```

```
AvroContainerOutputFormat'
LOCATION
  'hdfs:///user/temp/employee/data'
TBLPROPERTIES (
  'avro.schema.url'='hdfs:///user/temp/employee/schema/
employee.avsc',
  'transient_lastDdlTime'='1593929941')
 Time taken: 0.145 seconds, Fetched: 14 row(s)
```

Let's insert some records into hive table as below

```
INSERT INTO TABLE employee
  VALUES ('abc', "xyz");

  INSERT INTO TABLE employee
  VALUES ('def', "pqn");
```

```
hive> select * from employee;
OK
abc   Xyz
def   pqn
Time taken: 1.647 seconds
```

RC (ROW-COLUMNAR) FILE INPUT FORMAT

RCFile (Record Columnar File) partitions/groups records in two phases; first, it groups into a row, and then each row-group gets sliced into a columnar group.

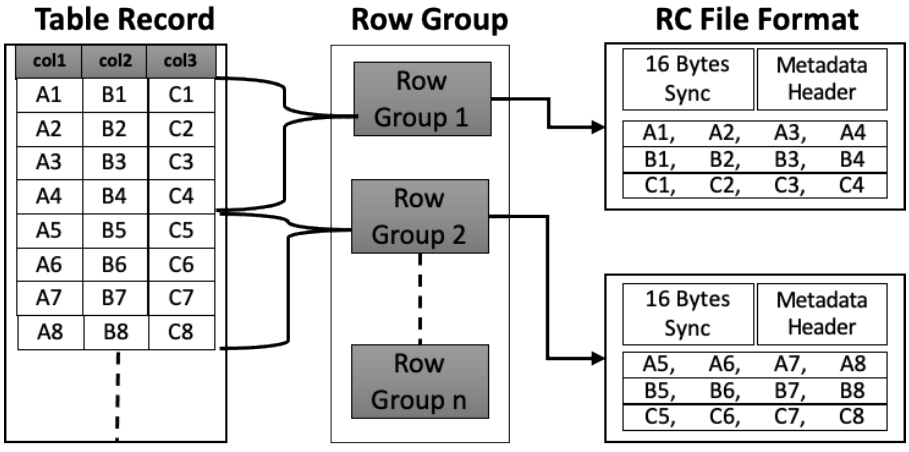

FIGURE 10.9 RCFile input format

RCFile forms key-value pairs where the metadata of a row splits as the key part of the record and all the row data splits as the value part. This is a better approach than using the sequence file format as it allows for fast data loading and query processing.

RCFile ensures all row data is stored into the same node and is an efficient choice for a query group of records with a selected column.

RCFile format provides high compression on rows and encourages column-oriented storage with indexing on the row group. RCFile does row-level grouping very efficiently when we query on a select group of a row, and columnar format is very useful when we perform analytics.

Note: Data cannot be loaded directly into RCFile; first, it needs to be loaded into a temp table and then overwrites into the RCFile table.

It stores columns separately, which can be read and decompressed on-demand or when needed. As it eventually becomes a columnar format, it provides better compression.

OPTIMIZED ROW COLUMNAR (ORC) FILE FORMAT

ORC (Optimized Record Columnar) provides for the efficient reading, writing, and processing of data. ORC writes row data in columnar format, which is highly efficient for compression and optimizes storage. The columnar format allows parallel processing and only reads the selected column. ORC supports all the benefits that RC provides with more enhancements (its compression is better than that of RCFile and it has faster queries). It stores data compactly and skips the irrelevant parts without large, complex, or manually maintained indices.

The ORC file format has many advantages:

1. Stores compressed columnar format, which leads to optimized and efficient disk storage. ORC is also an idea for vectorization optimizations in Tez.
2. ORC has a built-in index, min/max boundary on a strip, which can be selective on the strip to read and skip an unwanted strip. It uses a predicate pushdown and Boolean filter, which further optimize read performance.
3. It generates single ORC files as the output of each task, optimizing storage with not much load on the NameNode.

4. It supports concurrent reads of the same file.
5. It supports Hive's rich and complex data types such as struct, list, map, and union.
6. It can split files without scanning for markers.
7. It estimates the heap memory's upper bound using file footer information.
8. It supports schema evolution with the support of field addition and deletion.

The ORC file format contains rows of data in a group called *stripes*. The ORC file breaks up into fixed-size stripes, including an index, data, and footer, where the index keeps the range value of the columns stored and the light-weight row index position. These strips are independent of each other and are skipped if not needed for data analytics. Within each stripe, we can fetch the chosen columns. The stripe's footer includes analytical details of a strip-like count, sum, min, and max. Its advanced level of compression (approximately 75% of the original data) can process heavy loads very efficiently. It provides better performance than a sequence or RCFile format. It stores columns separately.

Note: Each stripe block's size default is 256 MB and has an index for block binderies.

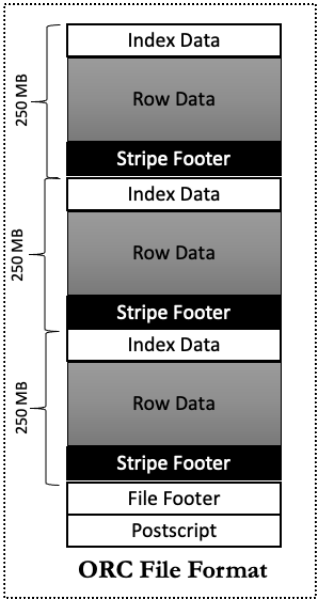

FIGURE 10.10 ORC file format

An ORC file consists of a collection of row data called stripes, including the supplementary information in a file footer. Postscripts keep compression parameters and the size of the compressed footer. The default stripe size is 250 MB, and the large files optimize HDFS storage and improve reads.

- **File Footer:** The stripe footer keeps a directory of stream locations.
- **Row Data:** The row of data used when the table is scanned.
- **Stripe Footer:** The stripe footer keeps a directory of the stream locations.
- **Row Data:** The rows of data used when the table is scanned.
- **Index Data:** The index data contains the min and max range values for each column and rows' position within each column. The ORC indexes are used for filtering and selecting stripes and row groups.

Configuration

In general, the ORC file format works well with the default configuration parameter; however, we can optimize it by setting an advanced configuration setting for exceptional cases.

Table 10.4 ORC Configuration

Key	Default Setting	Notes
orc.compress	ZLIB	Compression type (NONE, ZLIB, SNAPPY).
orc.compress.size	262,144	Number of bytes in each compression block.
orc.stripe.size	268,435,456	Number of bytes in each stripe.
orc.row.index.stride	10,000	Number of rows between index entries (>= 1,000).
orc.create.index	true	Sets whether to create row indexes.
orc.bloom.filter.columns	--	Comma-separated list of column names for which a Bloom filter must be created.
orc.bloom.filter.fpp	0.05	False positive probability for a Bloom filter. Must be greater than 0.0 and less than 1.0.

Conclusion

The ORC file format improves performance in all areas (reading, writing, and processing) compared to text, sequence, and RCFile. Compared to RCFile, ORC takes less time to access the data and takes less space to store it. However, the ORC file format uses more CPU overhead because it takes more time to decompress and perform decompression activities.

PARQUET

Parquet is an optimized columnar file format for Hadoop. It is a nested data structure in columnar format, making it a highly compressed and splittable file format. The Parquet file format partitions data horizontally and vertically, which gives a performance improvement if you need a subset of the data column. It can access single column values without accessing whole records.

Compared to row-level records (e.g., Avro), Parquet format is more efficient terms of storage and performance. It's the best for queries and it reads specific columns from a wide-column (with many columns) table since it only reads the required columns with a minimized I/O.

As we discussed earlier, columnar format groups the same columns of data together. Let's say we have the data as shown in Table 10.5.

Table 10.5 Example Data

id	Name	address
1	First1	Add1
2	First2	Add2
3	First3	Add3

The data in Table 10.5 can be represented in row-level format:

1	Fir st1	Ad d1	2	Fir st2	Ad d2	3	Fir st3	Ad d3

However, in the columnar format, it will appear differently:

1	2	3	First1	First2	First3	Add1	Add2	Add3

The Parquet file format is more efficient when you query a few columns because it will read only the required column, which minimizes the unwanted I/O call.

Let's say you are looking for only a few columns; in that case, its row-level format has to load all the row records and parse each record to extract data for the required column. However, it can directly go to the name column for the columnar format in Parquet format, as all the same columns are grouped.

The columnar file format increases the query performance as fewer I/O classes are needed. Parquet stores all data with a nested structure, meaning the data with a nested structure is stored together in a columnar group that can read individually. Parquet format uses the record shredding and assembly algorithm for storing nested structures in a columnar fashion.

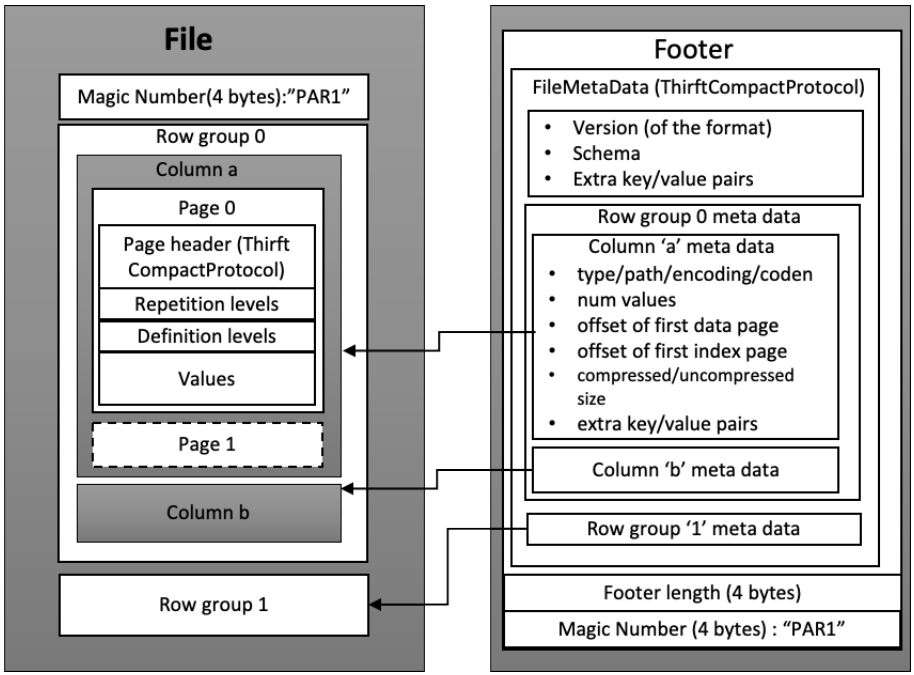

FIGURE 10.11 Parquet file format

Figure 10.11 shows the Parquet file format with a row, column chunk, page, and metadata.

- **Row group:** A horizontal slicing of data into a collection of rows
- **Column chunk:** A chunk of column data for a particular column

- **Page:** Column chunks that break into pages and are written back to back. A page shares a standard header, and the reader can skip an uninteresting page.
- **Metadata:** There are three types of metadata: file metadata, column (chunk) metadata, and page header metadata. All thrift structures are serialized using the TCompactProtocol.

Metadata includes data types, compression, schemas, statistics, and field names. Parquet data is organized by row group for each row group data held by a column, enabling column-level optimized compression. Parquet is a good option for a heavy data workload.

Parquet has a significant computation resource cost while writing data. It reduces a lot of the I/O cost and storage capacity on the read side. In the Hadoop, we remember to "write once and ready many;" hence, it makes sense to improve the read side's performance with some compromise on the write side. If you are interested in a column subset on a wide (a broad set of columns) table, then the Parquet file format can be beneficial for its query performance. It supports both file-level compression and block-level compression. In file-level compression, the entire file gets compressed, whereas the block level compression allows for compressing individual blocks of data within the file.

FILE FORMAT COMPARISONS

Avro vs. Parquet

Avro is a row-oriented serialized data file format that is useful for retrieving a dataset as a whole. It is fast reading and fast writing. The Avro file format also has excellent schema evolution, which supports frequent schema changes on the record table. It uses the JSON format, which promotes language-neutral processing.

Parquet is the columnar format, which is useful if you have a wide range table, and you are only looking at a subset of the columns in your data read. Parquet gives better read performance, as well as highly optimized storage compared to Avro. However, it requires more computation resources while writing data. If you compromise data and write performance and look only for a subset of data to read every time, it's better to use Parquet.

The following are some key differences between Parquet and Avro:

- Avro is a row-oriented file, whereas Parquet is a columnar- oriented storage format.
- Parquet is better for the analytics and aggregation of data from a few columns.
- Parquet reads and querying are much more efficient than its writing.
- Write operations in Avro are better than in the Parquet file format.
- Avro is quite impressive on schema evolution, such as adding, deleting, and updating column fields, whereas Parquet only supports schema append.

Parquet is ideal for querying a subset of columns in a multi-column table. Avro is perfect in the case of ETL operations, where we need to query all the columns.

ORC VS. PARQUET

Parquet and ORC both are columnar file formats with a few differences.

The Parquet file format supports a nested-like structure; hence, it's good to have nested columnar data; in contrast, ORC file format does better with flattened data. ORC consists of block-level indexes on a columnar structure, making them have a better read performance compared to the Parquet file format. It makes ORC a more optimized I/O call, allowing the search to skip an entire unwanted block.

ORC also supports the ACID operation, so if you are interested in an ACID transaction, choose the ORC file format. As explained earlier, Parquet is more for storing nested data, whereas ORC is more towards a Predicate Pushdown.

CHAPTER 11

DATA COMPRESSION

Many of us have worked on Big Data projects and have used a wide range of frameworks and tools to solve customer problems. Bringing the data to distributed storage is the first step in data processing. For Extract, Transform, Load (ETL) or Extract, Load, Transform (ELT), the first step is to extract the data and bring it in for processing. A storage system has a cost associated with it, and we always want to store more data in less storage space. Big Data processing happens over massive amounts of data, which may cause I/O and network bottlenecks. The shuffling of data across the network is always a painful process that consumes significant processing time. Hadoop uses data compression to address these challenges.

Data compression encodes the original data and persists with fewer bits. It helps to reduce the size of data files on the disk. Data compression in Hadoop reduced the storage capacity on a drive while decreasing the data transferred over the network.

DATA COMPRESSION BENEFITS

Less storage

A storage system comes with a significant amount of cost associated with it. Companies are moving toward the cloud, and even if we have to pay less for storage in the cloud, it is good to compress the data so that you pay less for storage. In some cases, compression helped gigabytes of data become megabytes of data.

Reduce Processing Time

You must think of how compression can reduce the processing time because there will be time to decompress the data before processing. It must be valid when the data size is small, and there is not much shuffling required across a distributed cluster. Think about a scenario where gigabytes or terabytes of data need to be shuffled across a network. The significant I/O and network bandwidth involved may cause performance problems, and thus compression gives performance a substantial boost in such cases.

CPU and I/O Trade-off

Compression reduces a significant amount of time consumed in I/O, but more data must be processed, which is the result of the decompression operation. When the amount of data is significant, the overall job processing time increases accordingly.

Block Compression in Hadoop

Hadoop uses splittable and non-splittable file formats. In most cases, splittable file formats are a better choice as they allow each block to be processed in parallel, which improves processing time. It is not advisable to compress one big file in a splittable file format and then use it for processing; in such cases, block compression can be a useful option. In block compression, data stored on HDFS blocks is compressed and used by MapReduce (or another processing tool) at a later stage.

DATA COMPRESSION IN HADOOP

Hadoop splits large datasets into fixed-size blocks and stores the data in distributed DataNodes across the cluster. Hadoop uses an extensive network I/O transfer to process data across nodes; hence, compressed data means less data travel across the system, which eventually improves performance. Hadoop supports various stages of data compression.

Input Compression

Input compression is the best approach to use for already compressed input files in the Hadoop environment. The files occupy less space, and when the dataset is processed through MapReduce, the data get decompressed automatically. If compression is splittable, it can be easily distributed across the block and improve overall Hadoop performance.

Intermediate Compression

Hadoop MapReduce stores the intermediate output in the HDFS, which can access next stage of the process. In the case of intermediate level compression, the temporary storage occupies less storage and improves the I/O transform rate.

```
conf.set("mapreduce.map.output.compress.codec",
"org.apache.hadoop.io.compress.SnappyCodec");
```

Output Compression

We can configure the Hadoop MapReduce job to compress the final output files before storing them to disk. It optimizes the HDFS storage and helps Hadoop maximize access by other applications for further processing.

```
FileOutputFormat.setOutputCompressorClass(job,
GzipCodec.class);
```

SPLITTING

When compressing files in Hadoop, we have to consider the compression codec support split. The split allows for splitting files into small chunks stored into the HDFS block; it can execute them by running independent map tasks. MapReduce can easily create input split tasks, reading each block separately and running tasks in parallel. This improves performance and optimizes storage.

In contrast, if input files are not splittable, map tasks must read and execute whole files by accessing all the blocks. Only one task has to obtain all the block data, which involves all blocks transferring data to the node where the map task is running. It degrades performance, and it requires a lot of memory to complete the whole process using one task and overhead data. It is the key reason why we should always consider splittable format when compressing the file in the Hadoop environment.

COMPRESSION CODEC

The compression codec is defined as the compression and decompression algorithm in the Hadoop environment. Hadoop provides various implementations of CompressionCodec interfaces, which are incapsulated as a streaming compression and decompression pair. The following are a few

codec implementations of the CompressionCodec interface: DeflateCodec, GzipCodec(for Gzip compression), HadoopSnappyCodec, Lz4Codec, SnappyCodec, and SplittableCompressionCodec.

```
package org.apache.hadoop.io.compress;

public interface CompressionCodec {

  /**
   * Create a CompressionOutputStream that will write to
the given
   * OutputStream}.
   */
  CompressionOutputStream
createOutputStream(OutputStream out)
  throws IOException;

  /**
   * Create a CompressionOutputStream that will write to
the given
   * OutputStream with the given Compressor.
   */
  CompressionOutputStream
createOutputStream(OutputStream out,
  Compressor compressor)
  throws IOException;

  /**
   * Create a  CompressionInputStream that will read from
the given
   * input stream.
   */
```

As shown in the code above, CompressionCodec provides a basic interface to create a Compression output stream and create an input stream which gets implemented by various compression codecs. `createOutputStream` creates `CompressionOutputStream` to write compressed data and `createInputStream` creates `CompressionInputStream` to read data from compressed files.

DATA COMPRESSIONS

CompressionCodec supports many compression formats in the Hadoop environment, as listed below.

Gzip Compression

Gzip uses GzipCodec (org.apache.hadoop.io.compress.GzipCodec) to compress and decompress the Gzip file. Gzip compression is a GNU zip compression used with the basic Deflate compression format to compress data. Gzip compression generates a file that has a .gz file extension. Gzip provides a high compression ratio; hence it's perfect for archival or data not accessed frequently. Due to the high compression ratio, it uses high CPU utilization on compression and decompression. Gzip compression is non-splittable; hence it's not the right choice for data used in the MapReduce job. The following is code for how we can set GzipCodec in the MapReduce program.

```
hadoop jar mapreduce-example.jar sort /
"-Dmapreduce.compress.map.output=true"      /
"-Dmapreduce.map.output.compression.codec= /
org.apache.hadoop.io.compress.GzipCodec"     /
"-Dmapreduce.output.compress=true"     /
"-Dmapreduce.output.compression.codec=/
org.apache.hadoop.io.compress.GzipCodec" -outKey  /
org.apache.hadoop.io.Text -outValue /
org.apache.hadoop.io.Text input output
```

We can use the following Hadoop configuration to use Gzip compression.

mapred-site.xml:

```
<!-- Add LZO Codecs details -->
<property>
<name>mapreduce.map.output.compress</name>
<value>true</value>
</property>
<property>
<name>mapreduce.map.output.compress.codec</name>
<value>org.apache.hadoop.io.compress.GzipCodec</value>
</property>
```

BZip2 Compression

The Bzip2 gives higher compression than Gzip; however, it is a splittable compression. Bzip2's higher compression takes more CPU resources for compression and decompression. The Bzip2 compression is a slow operation due to its high compression feature. The basic rule is more compression takes

more time for processing, which degrades the performance but improves the storage capacity.

BZip2 uses `BZip2Coded (org.apache.hadoop.io.compress.BZip-2Codec)` in Hadoop for compression and decompression. Gzip is not suitable for frequently accessed data; however, if you are storing data as archival that you will rarely need to query and are looking for high space optimization, Bzip2 is worth considering. It uses the .bzip2 file extension to write compressed files.

Lempel-Ziv-Oberhumer(LZO) Compression

LZO compression provides a tradeoff between the compression ratio and speed. LZO compression provides a lower compression ratio than Bzip2 and Zip2, and hence, it's fast to compress and decompress. It uses the .lzo file extension to write a compressed file. It uses an index to split files that inform MapReduce from where the files got split, which helps distribute files on the block and process them in parallel to improve the performance. Its moderate compression and decompression improve CPU utilization during compression and decompression. It allows small blocks to compress and decompress as an independent task. It creates a splittable compressed file in Hadoop that helps divide jobs and run them in parallel in distributed environments. It improves compression and decompression speed. It doesn't require any external indexing.

We can use the following Hadoop configuration for the LZO compression.

core-site.xml:

```
<property>
<name>io.compression.codecs</name>
<value>com.hadoop.compression.lzo.LzoCodec</value>
</property>
<property>
<name>io.compression.codec.lzo.class</name>
<value>com.hadoop.compression.lzo.LzoCodec</value>
</property>
```

mapred-site.xml:

```
<!-- Add LZO Codecs details -->
<property>
<name>mapreduce.map.output.compress</name>
<value>true</value>
```

```
</property>
<property>
<name>mapreduce.map.output.compress.codec</name>
<value>com.hadoop.compression.lzo.LzoCodec</value>
</property>
```

Snappy Compression

Google created Snappy compression, previously known as Zippy, for fast data compression and decompression. It is written in C++ with LZ77 algorithms and moved to open source in 2011. Snappy uses `SnappyCodec` (`org.apache.hadoop.io.compress.SnappyCodec`) in Hadoop for compression and decompression. The Snappy compression is used with various splittable file formats, such as Sequence Files, Avro, and Paraquat. It is inherently splittable.

Snappy has a modest compression and decompression ratio, but high-speed compression and decompression speed. Snappy is very fast and utilizes few CPU resources, and it is splittable to support parallel processing. If you are looking for compression for widespread data access, Snappy is the right choice. It doesn't give maximum compression like Gzip or Bzip2, but provides speedy data processing.

We can use the following code for Snappy:

mapred-site.xml:

```
<!-- Add LZO Codecs details -->
<property>
<name>mapreduce.map.output.compress</name>
<value>true</value>
</property>
<property>
<name>mapreduce.map.output.compress.codec</name>
<value> org.apache.hadoop.io.compress.SnappyCodec </value>
</property>
```

Choosing the Right Compression Approach

Selecting the right compression depends on your application use case. Do you want to maximize your application speed, or are you more concerned about keeping storage costs down? In general, you should try different strategies for your application, and benchmark them with representative datasets to find the best approach. Table 11.1 shows some basic characteristics of each compression technique.

Table 11.1 Compression Codec

Codec	Splittable	Degree of compression	Compression speed
Gzip	No	Medium	Medium
BZip2	Yes	High	Slow
Snappy	No	Medium	Fast
LZO	No, unless indexed	Medium	Fast

Many characteristics factors, such as the CPU capacity, I/O, and network bandwidth characteristics, can help you decide on the right compression selection.

- Bzip2 has a good compression ratio, but it is very slow; Snappy is very fast, but offers a low compression ratio.
- Bzip2's compression is very high and CPU intensive, but its decompression speed is better than Gzip. Since we write once, read many times in Hadoop, Bzip2 is more effective than traditional Gzip.
- LZO, on the other hand, performs better in processing speed. It is better than Gzip, but slower than Snappy in regards to compression. LZO compression is moderate, and lies between Bzip2 and Snappy.

Before selecting a compression approach, we should think about if the compression is splittable, which can further enhance the performance. As discussed previously, when large files are stored in the HDFS, they are split and stored in fixed blocks in the HDFS. If a file format is splittable and supports splittable compression, then each of the split block processes is run separately in parallel.

If compression doesn't support the splittable option, like Gzip, the large files are stored into multiple blocks, but processing a single MapReduce task pulls all blocks together to decompress and process.

Suppose you have non-splittable files with the limit under the block size. In that case, each file gets stored into a single block. However, a very large file requires splittable compression. Basically, for large files with frequent access patterns, you should not use non-splittable compression. If your data is already compressed, such as a binary file, then compression won't help, and will result in extra overhead compression.

Note: Zip is an archive format; it contains multiple files into a single zip. Each file is compressed and stored separately inside Zip; Zip can process separate files independently. This property means Zip file supports splittable at the file boundaries.

REFERENCES

[HDPDEFG03] *Hadoop: The Definitive Guide Third Edition by Tom White.*

[APAHDP3] *https://hadoop.apache.org/docs/r3.0.0/*

[APAHIV3] *https://hive.apache.org/*

INDEX

I

J