MODULE-2 HDFS (HADOOP DISTRIBUTED FILE SYSTEM)

Introduction to Distributed Filesystems

When a dataset exceeds the storage capacity of a single physical machine, it becomes necessary to partition and distribute it across multiple machines. Distributed filesystems manage this distributed storage across a network of machines, allowing for scalability and fault tolerance.

Key Points:

- **Definition**: Distributed filesystems are filesystems that operate over a network, managing storage across multiple machines.
- **Complexity**: They are more complex than traditional disk filesystems due to the challenges of network programming and ensuring data integrity across nodes.
- **Challenges**:
  - **Node Failure**: Ensuring the filesystem can tolerate the failure of nodes without data loss.
  - **Network Issues**: Handling latency, bandwidth limitations, and reliability of the network.

2. Hadoop Distributed Filesystem (HDFS)

HDFS is a distributed filesystem designed to store and manage large datasets across a cluster of machines. It is a core component of the Hadoop ecosystem.

Key Features:

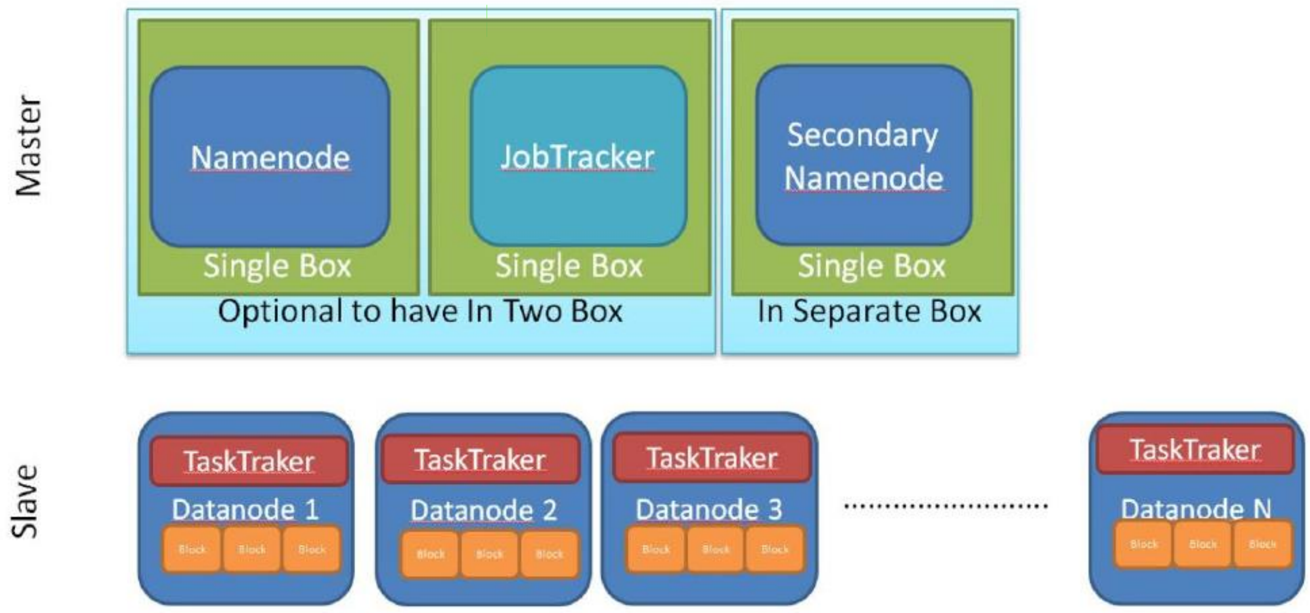
- **Scalability**: HDFS is designed to scale out by adding more nodes to the cluster, which increases **storage capacity and computational power.**
- **Fault Tolerance**: It **replicates data** across multiple nodes to ensure data is not lost in case of hardware failures.
- **High Throughput**: Optimized for **high throughput rather than low latency**, making it suitable for processing large files.

Key Components:

- **NameNode**: Manages the metadata of the filesystem, such as file names, directory structure, and file-to-block mapping. It is a single point of failure and crucial for the filesystem's operation.
- **DataNode**: Stores the actual data blocks. Data is replicated across multiple DataNodes for fault tolerance.
- **Secondary NameNode**: Performs periodic checkpoints of the NameNode's metadata to provide a backup in case of failures.

HDFS Operation:

- **Data Storage**: Files are split into blocks (default size is 128 MB or 256 MB) and distributed across the cluster. Each block is replicated multiple times (default replication factor is 3) to ensure data reliability.
- **Access**: HDFS is designed for large-scale data processing tasks and provides write-once, read-many access patterns. It is not optimized for small file storage or frequent updates.



Integration with Other Storage Systems

Hadoop's filesystem abstraction allows it to integrate with various storage systems beyond HDFS:

- **Local Filesystem**: Hadoop can use the local filesystem for development or smaller datasets.
- **Amazon S3**: Hadoop can interact with Amazon S3 as a storage backend, allowing for scalable storage in the cloud. This integration uses the S3A file system client to read and write data.
- **RDBMS, DateLake, DataWarehouses, streaming systems, cloud systems and so on**

Key Considerations:

- **Data Consistency**: When integrating with other storage systems, ensure that the data consistency models are compatible with your use case.
- **Performance**: Consider the performance implications of different storage backends, particularly in terms of latency and throughput.

Design and Use Cases of HDFS

1. Overview of HDFS Design

HDFS (Hadoop Distributed File System) is specifically designed to address the needs of storing and processing very large files in a distributed computing environment. Here’s a breakdown of its key design principles:

Key Features:

- **Very Large Files**:
  - **Definition**: In HDFS, "very large" refers to files that range from hundreds of megabytes to terabytes, and even petabytes in some cases.
  - **Purpose**: It is optimized for storing such large datasets efficiently, which is essential for big data applications and analytics.
- **Streaming Data Access**:
  - **Access Pattern**: HDFS is built around a write-once, read-many-times model. This model suits scenarios where data is initially ingested and then analyzed multiple times.
  - **Performance**: The design prioritizes high throughput for reading large datasets over low-latency access. The time to read the entire dataset is more critical than the time to read the first record.
- **Commodity Hardware**:
  - **Cost Efficiency**: HDFS is designed to run on clusters of inexpensive, commonly available hardware. This design choice makes it cost-effective and scalable.
  - **Fault Tolerance**: It anticipates hardware failures and is designed to continue operating with minimal user impact, leveraging the redundancy built into the system.

2. Challenges and Limitations

While HDFS is powerful for many use cases, there are specific scenarios where it may not be the best fit:

Low-Latency Data Access:

- **Latency Constraints**: Applications requiring quick, sub-second data access (in the tens of milliseconds range) may not perform well with HDFS. Its optimization for high throughput rather than low latency means it may not meet the performance needs of such applications.
- **Alternative Solutions**: HBase, which is built on top of HDFS, is often used for scenarios requiring low-latency data access. HBase provides capabilities for fast random access to data.

Handling Lots of Small Files:

- **Metadata Management**: HDFS stores metadata (such as file and directory information) in memory on the NameNode. This means the scalability of HDFS in terms of the number of files is limited by the memory capacity of the NameNode.
- **Scalability Constraints**: Each file, directory, and block take up about 150 bytes of memory. For example, handling one million files requires at least 300 MB of memory. While managing millions of files is feasible, managing billions of files can exceed current hardware capabilities.

Multiple Writers and Arbitrary Modifications:

- **Write Restrictions**: HDFS supports a single writer per file, with data being appended to the end of the file. It does not support multiple writers or modifications at arbitrary file offsets.
- **Future Considerations**: Although support for multiple writers and arbitrary modifications might be introduced in the future, such features are likely to be less efficient compared to the current append-only model.

HDFS Concepts

Blocks

1. Understanding Blocks in Filesystems

In both traditional filesystems and HDFS, the concept of "blocks" is fundamental. Here's a detailed look at what blocks are and their significance:

Filesystem Blocks vs. HDFS Blocks:

- **Traditional Filesystems**:
  - **Disk Blocks**: The smallest unit of storage on a disk, usually 512 bytes.
  - **Filesystem Blocks**: Typically larger, a few kilobytes in size, used by the filesystem to manage data. The filesystem block size is an integral multiple of the disk block size.
  - **Tools**: Commands like df and fsck operate at the filesystem block level for maintenance and checking.
- **HDFS Blocks**:
  - **Size**: Much larger than traditional filesystem blocks, with a default size of 128 MB.
  - **Function**: Files in HDFS are divided into blocks of this size, which are stored independently across the cluster.
  - **Storage Efficiency**: If a file is smaller than a block, only the space needed for the file is used (e.g., a 1 MB file on a 128 MB block uses only 1 MB of disk space).

2. Why Are HDFS Blocks So Large?

The choice of large block sizes in HDFS is driven by several practical considerations:

- **Minimizing Seek Time**:
    - **Seek Time vs. Transfer Rate**: The time to seek to the start of a block can be minimized relative to the time taken to transfer the data. Large blocks ensure that the transfer time dominates over the seek time.
- **MapReduce Considerations**:
    - **Map Tasks**: In MapReduce, tasks typically process one block at a time. Having too few blocks compared to the number of nodes can lead to slower job execution due to insufficient parallelism.

3. Benefits of Block Abstraction

The block abstraction in HDFS provides several advantages:

- **Handling Large Files**:
    - **Scalability**: HDFS allows files to exceed the size of any single disk in the cluster. Blocks can be distributed across all available disks, enabling the storage of very large files.
- **Simplified Storage Management**:
    - **Fixed Size**: Blocks are of a fixed size, making it straightforward to calculate storage requirements and manage space on disks.
    - **Metadata Management**: Since blocks are only chunks of data, file metadata (like permissions) is managed separately from the blocks, simplifying the storage subsystem.
- **Replication and Fault Tolerance**:
    - **Replication**: To ensure fault tolerance, each block is replicated across multiple machines (typically three). This redundancy allows for recovery from disk or machine failures.
    - **Automatic Recovery**: If a block becomes unavailable due to corruption or a failure, it can be replicated from other available copies to restore the required replication factor.
- **Load Distribution**:
    - **Read Load**: Applications may set a higher replication factor for blocks in frequently accessed files to distribute the read load across the cluster.

4. Filesystem Check (fsck) with HDFS

HDFS provides a command to understand and manage blocks:

- **Command**: hdfs fsck / -files -blocks
    - **Function**: Lists the blocks that make up each file in the HDFS filesystem.

    - **Purpose**: Useful for checking the health and integrity of the filesystem and its blocks.

Namenodes and Datanodes in HDFS

HDFS (Hadoop Distributed File System) operates with a master-worker architecture involving two main types of nodes: Namenodes and Datanodes. Understanding their roles and mechanisms is essential for maintaining the health and performance of an HDFS cluster.

1. Namenode

The Namenode is the master node in the HDFS architecture, responsible for managing the filesystem namespace and metadata.

Responsibilities:

- **Filesystem Namespace Management**:
    - Maintains the directory tree of the filesystem and metadata for all files and directories.
    - Stores metadata persistently on local disks in two files: the namespace image and the edit log.
- **Block Management**:
    - Keeps track of which Datanodes store the blocks for each file.
    - Does not persistently store block locations; instead, this information is reconstructed from Datanodes during system startup.
- **Client Interaction**:
    - Clients interact with the Namenode to perform filesystem operations.
    - Provides a filesystem interface, abstracting the complexities of the Namenode and Datanodes from the user.

Failure and Recovery:

- **Critical Role**:
    - If the Namenode fails, the filesystem cannot be used, and data loss can occur because the system would not know how to reconstruct files from the blocks on Datanodes.
- **Resilience Mechanisms**:
    - **Backup**:
        - Namenode's persistent state is backed up to multiple filesystems. This includes synchronous and atomic writes to local disks and remote NFS mounts.
    - **Secondary Namenode**:
        - **Role**: Periodically merges the namespace image with the edit log to prevent the edit log from becoming too large.

    - **Operation**: Runs on a separate physical machine with sufficient CPU and memory. It keeps a copy of the merged namespace image.
    - **Limitations**: The state of the Secondary Namenode lags behind the primary, so there is a risk of data loss if the primary fails. In such cases, the primary's metadata files are copied to the Secondary Namenode, which then acts as the new primary.
    - **Alternative**: A hot standby Namenode can be used for high availability, which provides a more robust failover solution.

2. Datanodes

Datanodes are the worker nodes in the HDFS architecture, responsible for storing and managing the data blocks.

Responsibilities:

- **Block Storage and Retrieval**:
  - Store and retrieve blocks as instructed by clients or the Namenode.
  - Report periodically to the Namenode with lists of blocks they are storing.

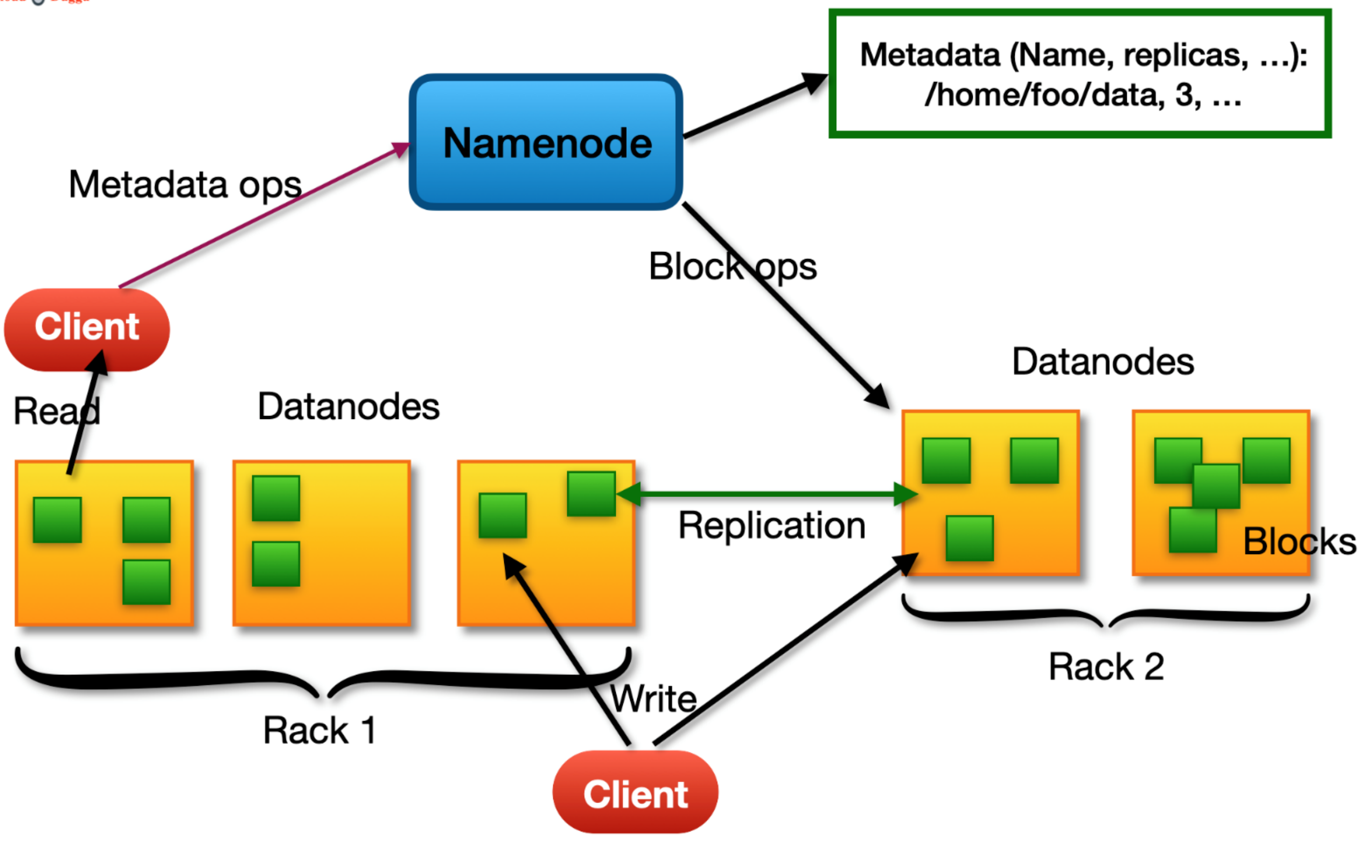
Operation:

- **Data Management**:
  - Datanodes handle the actual data blocks, and their efficiency directly impacts the performance of HDFS.
  - They ensure data redundancy and availability by storing multiple replicas of each block.

3. Interaction Between Namenode and Datanodes

- **Client Operations**:
  - Clients interact with the Namenode to get information about where to find the blocks of a file.
  - The Namenode directs clients to the appropriate Datanodes for block retrieval or storage.
- **Datanode Reporting**:
  - Datanodes regularly send heartbeat signals and block reports to the Namenode.
  - This reporting helps the Namenode monitor the health of the Datanodes and manage data replication and block recovery.

HDFS Architecture



Block Caching

Purpose:

- Block caching improves read performance by storing frequently accessed blocks in the memory of Datanodes.
- It allows quick access to data without having to read from disk repeatedly.

Mechanics:

- **Caching Location:** Blocks are cached in an off-heap memory area on the Datanodes. Off-heap memory is used to avoid garbage collection overhead.
- **Cache Scope:** By default, each block is cached in one Datanode's memory. This is configurable on a per-file basis, allowing multiple Datanodes to cache the same block if needed.
- **Cache Management:** Administrators can specify which files should be cached and the duration of caching using cache directives added to a cache pool.
- **Cache Pools:** Cache pools are used to manage cache permissions and resource usage. They help in organizing and controlling access to cached data.

Benefits:

- **Performance Improvement:** Increases read performance by reducing disk I/O. For instance, job schedulers in frameworks like MapReduce or Spark can schedule tasks on Datanodes that have relevant blocks cached, leading to faster data access.
- **Use Cases:** Ideal for use cases such as small lookup tables used in joins, where data is frequently accessed.

HDFS Federation

Purpose:

- HDFS Federation enhances the scalability of HDFS by allowing the use of multiple Namenodes.

Mechanics:

- **Namespace Management:** In HDFS Federation, the filesystem namespace is split across multiple Namenodes. Each Namenode manages a portion of the namespace and its associated block pool.
- **Namespace Volumes:** Each Namenode is responsible for a namespace volume, which includes metadata for its portion of the namespace. These volumes are independent, meaning that the failure of one Namenode does not affect others.
- **Block Pool Storage:** Datanodes register with all Namenodes in the cluster and store blocks from multiple block pools. This ensures that blocks from different namespaces can be stored and managed efficiently.

Access:

- **Client Interaction:** Clients use client-side mount tables to map file paths to the appropriate Namenodes. Configuration is managed using ViewFileSystem and the viewfs:// URIs.

Benefits:

- **Scalability:** Allows the HDFS cluster to scale beyond the limitations of a single Namenode's memory by distributing the namespace management.
- **Fault Tolerance:** Improves fault tolerance by isolating namespace management across multiple Namenodes.

HDFS High Availability (HA)

Purpose:

- HDFS HA addresses the single point of failure issue associated with the Namenode by providing a pair of Namenodes in an active-standby configuration.

Mechanics:

- **Active-Standby Configuration:** One Namenode acts as the active Namenode, handling client requests, while the other acts as the standby, ready to take over if the active Namenode fails.
- **Shared Storage:** Namenodes use highly available shared storage to keep the edit log. Two main choices for this storage are:
  - **NFS Filer:** Traditional Network File System for shared storage.
  - **Quorum Journal Manager (QJM):** A specialized HDFS component designed to provide highly available edit logs. It uses a group of journal nodes where each edit must be written to a majority of nodes (e.g., three nodes, allowing for one node failure).
- **Datanode Reporting:** Datanodes must report block information to both the active and standby Namenodes.
- **Client Configuration:** Clients must be configured to handle Namenode failover transparently.

Failover Process:

- **Quick Failover:** The standby Namenode can take over quickly (within seconds) as it maintains up-to-date state in memory, including the latest edit log and block mappings.
- **Recovery:** In case the standby Namenode is down when the active fails, the administrator can start the standby from cold. While this process is better than the non-HA scenario, it still requires standard operational procedures.

Advantages:

- **Reduced Downtime:** Provides high availability and reduces downtime by enabling a rapid failover mechanism.
- **Operational Efficiency:** Standardizes the failover process, making it more predictable and manageable.

Failover Process

Failover Controller:

- **Role:** Manages the transition between the active and standby Namenodes. It ensures that only one Namenode is active at a time.
- **Default Implementation:** Uses ZooKeeper, which coordinates the failover process by monitoring the health of the Namenodes and triggering failover if needed.
- **Process:**
    - **Heartbeat Mechanism:** Each Namenode runs a lightweight failover controller that sends heartbeats to check the status of the other Namenode.
    - **Graceful Failover:** Can be initiated manually by an administrator, such as during routine maintenance. This involves an orderly transition where both Namenodes switch roles smoothly.
    - **Ungraceful Failover:** Occurs automatically if the active Namenode fails unexpectedly. This can happen due to issues like network partitions or slow networks, where the active Namenode might still be running but is unreachable.

Client Failover Handling:

- **Transparent to Clients:** The client library manages failover transparently. Clients are configured with a logical hostname that maps to a pair of Namenode addresses.
- **Failover Mechanism:** The client library attempts connections to each Namenode address in turn until it succeeds. This ensures continuous service availability even during failover.

Fencing Mechanisms

Purpose:

- **Prevent Data Corruption:** Ensures that the previously active Namenode, which might still be running or reachable, does not interfere with the cluster operations or cause data corruption.

Fencing Techniques:

- **SSH Fencing Command:** A simple method where an SSH command is used to kill the process of the previously active Namenode. This is effective if the failover controller is unsure if the old Namenode has stopped completely.
- **NFS Filer:** When using an NFS filer for shared edit logs, stronger fencing methods are required because NFS filers do not enforce exclusive write access as effectively as QJM.
    - **Revoking Access:** Commands to revoke the Namenode’s access to the shared storage directory can be used.
    - **Disabling Network Port:** The Namenode’s network port can be disabled via remote management commands to prevent it from accepting requests.

- **STONITH (Shoot The Other Node In The Head):** This drastic measure involves using a specialized power distribution unit (PDU) to forcibly power down the host machine of the failed Namenode, ensuring it can no longer affect the system.

Hadoop Filesystems: Overview and Usage

Hadoop's flexible filesystem abstraction allows it to interact with various storage systems, each designed for specific use cases. Understanding these filesystems can help in choosing the right one for your needs, whether for local testing, distributed processing, or cloud integration.

1. Local FileSystem (fs.LocalFileSystem)

- **URI Scheme:** file:///
- **Description:** Represents local disk storage. Ideal for small-scale testing or development on a single machine.
- **Purpose:**
  - Testing and development in a local environment.
  - When data integrity through client-side checksums is needed.
- **How to Use:**
  - Access local files directly using the file:/// scheme.
  - For environments where checksums are not required, use RawLocalFileSystem.

2. Hadoop Distributed File System (HDFS) (hdfs.DistributedFileSystem)

- **URI Scheme:** hdfs://
- **Description:** A distributed storage system designed for high-throughput access to large datasets. Provides fault tolerance through data replication.
- **Purpose:**
  - Handling large-scale data storage and processing.
  - Optimized for use with MapReduce and other Hadoop processing frameworks.
  - Ensures data reliability and fault tolerance.
- **How to Use:**
  - Access files using the hdfs:// scheme.
  - Ideal for processing large volumes of data across multiple nodes.

3. WebHDFS (hdfs.web.WebHdfsFileSystem)

- **URI Scheme:** webhdfs://

- **Description:** Allows access to HDFS over HTTP, enabling interaction through web interfaces and REST APIs.
- **Purpose:**
  - Web-based access to HDFS.
  - Integration with web applications and services.
- **How to Use:**
  - Interact with HDFS via HTTP requests using the webhdfs:// scheme.

4. Secure WebHDFS (hdfs.web.SWebHdfsFileSystem)

- **URI Scheme:** swebhdfs://
- **Description:** Secure version of WebHDFS using HTTPS for encrypted data transfers.
- **Purpose:**
  - Secure data access over the web.
  - Ensures encrypted communication with HDFS.
- **How to Use:**
  - Access HDFS with secure HTTPS connections using the swebhdfs:// scheme.

5. Hadoop Archives (HAR) (fs.HarFileSystem)

- **URI Scheme:** har://
- **Description:** Archives multiple small files into a single file to reduce memory usage on the namenode.
- **Purpose:**
  - Reduces namenode memory overhead by consolidating small files.
  - Useful for managing a large number of small files.
- **How to Use:**
  - Access archived files using the har:// scheme.

6. ViewFS (viewfs.ViewFileSystem)

- **URI Scheme:** viewfs://
- **Description:** Provides a unified view of multiple filesystems, often used to create mount points for federated namenodes.
- **Purpose:**
  - Aggregates multiple HDFS clusters or filesystems into a single namespace.
  - Facilitates data access across federated systems.
- **How to Use:**
  - Access a combined view of filesystems using the viewfs:// scheme.

7. FTP (fs.ftp.FTPFileSystem)

- **URI Scheme:** ftp://
- **Description:** Interacts with FTP servers, allowing file operations over FTP.
- **Purpose:**
  - Integration with FTP servers for data access and manipulation.
- **How to Use:**
  - Read and write files over FTP using the ftp:// scheme.

8. Amazon S3 (fs.s3a.S3AFileSystem)

- **URI Scheme:** s3a://
- **Description:** Accesses Amazon S3 storage, supporting larger file sizes and better performance than the older s3n implementation.
- **Purpose:**
  - Integration with Amazon S3 cloud storage.
  - Suitable for large-scale cloud storage requirements.
- **How to Use:**
  - Access files in S3 using the s3a:// scheme.

9. Microsoft Azure Blob Storage (fs.azure.NativeAzureFileSystem)

- **URI Scheme:** wasb://
- **Description:** Provides access to Azure Blob Storage, integrating with Hadoop for cloud-based storage.
- **Purpose:**
  - Storage integration with Microsoft Azure.
  - Suitable for cloud storage needs with Hadoop.
- **How to Use:**
  - Access Azure Blob Storage using the wasb:// scheme.

10. OpenStack Swift (fs.swift.snative.SwiftNativeFileSystem)

- **URI Scheme:** swift://
- **Description:** Accesses OpenStack Swift, an object storage service.
- **Purpose:**
  - Integration with OpenStack Swift for object storage.

  - Suitable for cloud storage with OpenStack.
- **How to Use:**
  - Access files in Swift using the swift:// scheme.

Choosing a Filesystem

- **HDFS** is recommended for large-scale data processing due to its optimization for data locality and fault tolerance.
- **Cloud-based filesystems** (S3, Azure, Swift) are used for integrating with cloud storage platforms and can be suitable depending on specific storage and processing requirements.
- **Local filesystems** are best for small-scale or development testing.

Each filesystem has distinct use cases depending on your needs for data volume, processing, and integration with other storage systems.

Interface

Hadoop provides several interfaces to interact with its filesystems, catering to different programming languages, environments, and access needs. Here's an overview of the key interfaces:

1. Java API

- **Purpose**: Hadoop is written in Java, so its primary filesystem interactions are handled through Java APIs.
- **Implementation**: The org.apache.hadoop.fs.FileSystem class provides methods for filesystem operations such as reading, writing, and listing files.
- **Usage**: Commonly used for Java applications and for running Hadoop commands.

2. HTTP (WebHDFS)

- **Purpose**: Provides access to HDFS via HTTP, enabling non-Java applications to interact with Hadoop.
- **Implementation**: The WebHDFS protocol exposes an HTTP REST API that allows file operations over HTTP or HTTPS.
- **Access Methods**:
  - **Direct Access**: Clients interact directly with HDFS daemons (namenodes and datanodes) via embedded web servers.

    - **Proxy Access**: Clients connect through an HttpFS proxy server, which forwards requests to HDFS. This approach can help with firewall and bandwidth management, and is useful for cross-data-center or cloud access.
- **Performance**: WebHDFS is slower than the native Java client, so it's less suitable for very large data transfers.

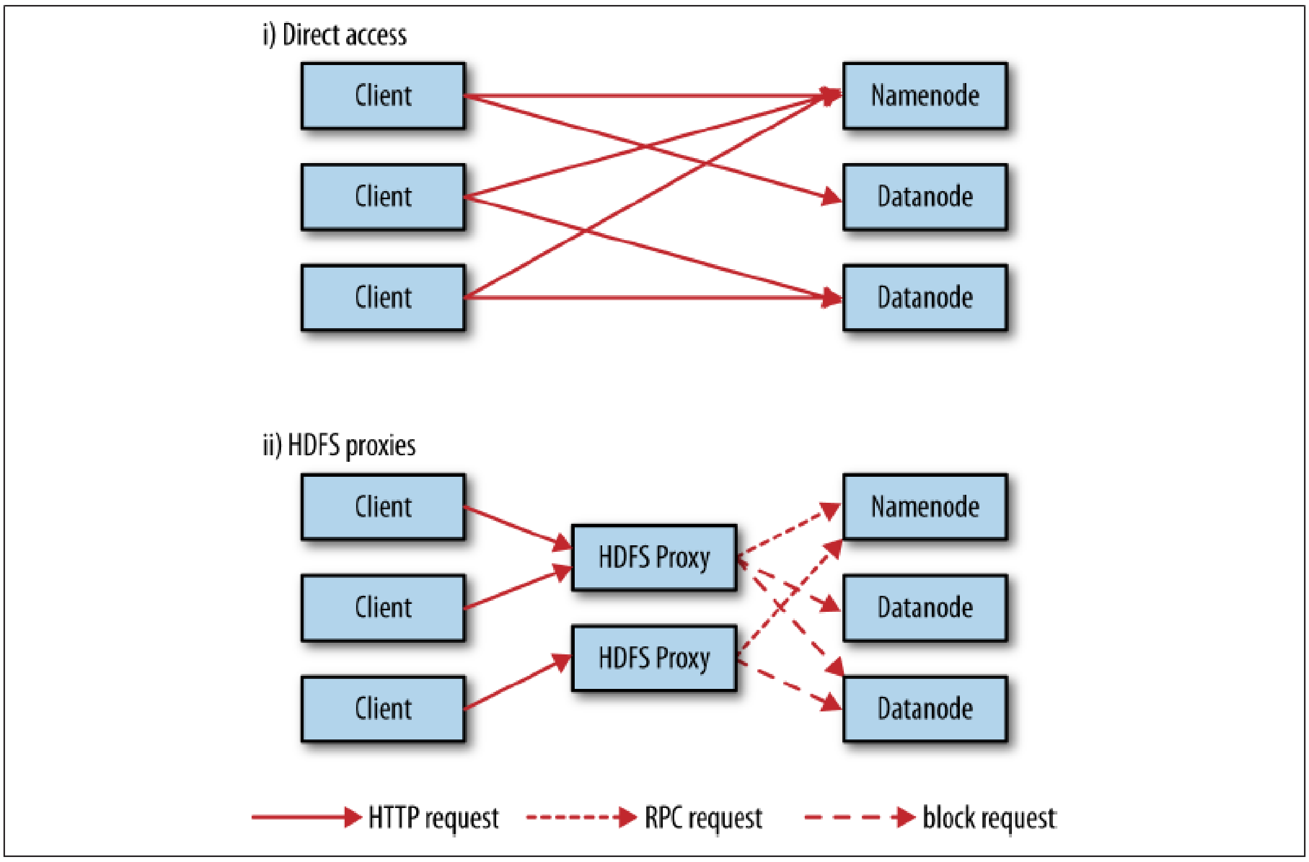


Figure 3-1. Accessing HDFS over HTTP directly and via a bank of HDFS proxies

3. C API

- **Purpose**: Provides a C library for interacting with Hadoop filesystems.
- **Implementation**:
    - **libhdfs**: Mirrors the Java FileSystem interface using Java Native Interface (JNI). It can access any Hadoop filesystem.
    - **libwebhdfs**: Uses the WebHDFS interface for accessing HDFS.
- **Usage**: Allows C applications to interact with Hadoop filesystems. Prebuilt binaries are available for 64-bit Linux, with other platforms requiring custom builds.

4. NFS (Network File System)

- **Purpose**: Allows HDFS to be mounted on a local client's filesystem, enabling interaction using standard Unix utilities.

- **Implementation**: Hadoop’s NFSv3 gateway makes HDFS appear as a local filesystem.
- **Usage**: Useful for Unix-like environments where standard tools and libraries are used for file operations.
- **Limitations**: Supports appending to files but not random modifications, as HDFS only allows appending to the end of a file.

5. FUSE (Filesystem in Userspace)

- **Purpose**: Integrates filesystems implemented in user space as standard Unix filesystems.
- **Implementation**: Hadoop’s Fuse-DFS module allows HDFS (or any Hadoop filesystem) to be mounted as a local filesystem.
- **Usage**: Provides a standard interface for interacting with Hadoop filesystems from Unix-like systems.
- **Limitations**: Generally, the Hadoop NFS gateway is preferred over Fuse-DFS for robustness.

Summary

These interfaces provide flexibility for accessing Hadoop filesystems from various programming environments and tools:

- **Java API**: Native and preferred for Hadoop’s Java-based ecosystem.
- **HTTP (WebHDFS)**: Provides web-based access and integration with non-Java applications.
- **C API**: Allows C applications to interact with Hadoop.
- **NFS**: Enables integration with Unix utilities and libraries via network file system access.
- **FUSE**: Integrates Hadoop filesystems with Unix-like systems as a local filesystem.

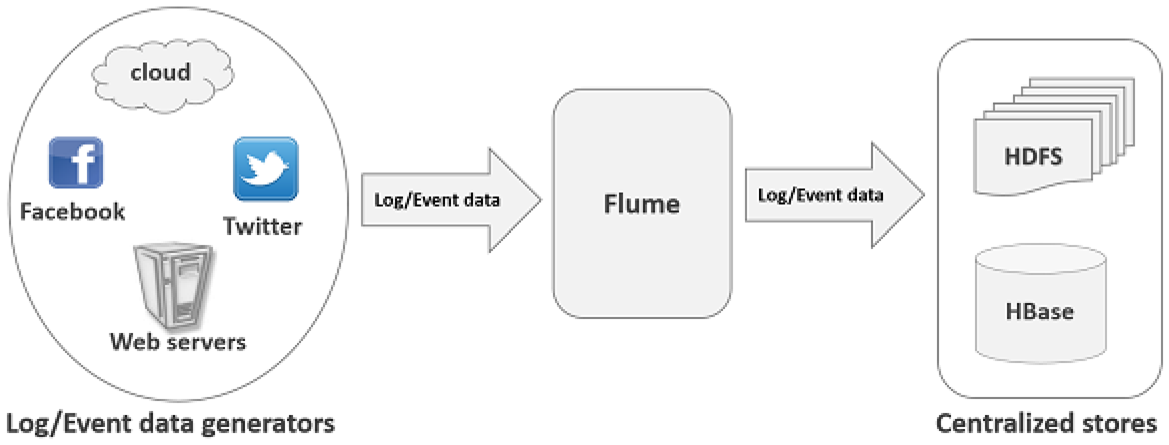
Each interface serves different use cases and provides different levels of access, performance, and compatibility.

1. FLUME – INTRODUCTION

What is Flume?

Apache Flume is a tool/service/data ingestion mechanism for collecting aggregating and transporting large amounts of streaming data such as log files, events (etc...) from various sources to a centralized data store.

Flume is a highly reliable, distributed, and configurable tool. It is principally designed to copy streaming data (log data) from various web servers to HDFS.



Applications of Flume

Assume an e-commerce web application wants to analyze the customer behavior from a particular region. To do so, they would need to move the available log data in to Hadoop for analysis. Here, Apache Flume comes to our rescue.

Flume is used to move the log data generated by application servers into HDFS at a higher speed.

Advantages of Flume

Here are the advantages of using Flume:

- Using Apache Flume we can store the data in to any of the centralized stores (HBase, HDFS).

- When the rate of incoming data exceeds the rate at which data can be written to the destination, Flume acts as a mediator between data producers and the centralized stores and provides a steady flow of data between them.

- Flume provides the feature of **contextual routing**.

- The transactions in Flume are channel-based where two transactions (one sender and one receiver) are maintained for each message. It guarantees reliable message delivery.

- Flume is reliable, fault tolerant, scalable, manageable, and customizable.

Features of Flume

Some of the notable features of Flume are as follows:

- Flume ingests log data from multiple web servers into a centralized store (HDFS, HBase) efficiently.

- Using Flume, we can get the data from multiple servers immediately into Hadoop.

- Along with the log files, Flume is also used to import huge volumes of event data produced by social networking sites like Facebook and Twitter, and e-commerce websites like Amazon and Flipkart.

- Flume supports a large set of sources and destinations types.

- Flume supports multi-hop flows, fan-in fan-out flows, contextual routing, etc.

- Flume can be scaled horizontally.

2. FLUME – DATA TRANSFER IN HADOOP

Big Data, as we know, is a collection of large datasets that cannot be processed using traditional computing techniques. Big Data, when analyzed, gives valuable results. **Hadoop** is an open-source framework that allows to store and process Big Data in a distributed environment across clusters of computers using simple programming models.

Streaming / Log Data

Generally, most of the data that is to be analyzed will be produced by various data sources like applications servers, social networking sites, cloud servers, and enterprise servers. This data will be in the form of **log files** and **events**.

Log file: In general, a log file is a **file** that lists events/actions that occur in an operating system. For example, web servers list every request made to the server in the log files.

On harvesting such log data, we can get information about:

- the application performance and locate various software and hardware failures.
- the user behavior and derive better business insights.

The traditional method of transferring data into the HDFS system is to use the **put** command. Let us see how to use the **put** command.

HDFS put Command

The main challenge in handling the log data is in moving these logs produced by multiple servers to the Hadoop environment.

Hadoop **File System Shell** provides commands to insert data into Hadoop and read from it. You can insert data into Hadoop using the **put** command as shown below.

```
$ Hadoop fs –put /path of the required file  /path in HDFS where to save the file
```

Problem with put Command

We can use the **put** command of Hadoop to transfer data from these sources to HDFS. But, it suffers from the following drawbacks:

- Using **put** command, we can transfer **only one file at a time** while the data generators generate data at a much higher rate. Since the analysis made on older data is less accurate, we need to have a solution to transfer data in real time.

- If we use **put** command, the data is needed to be packaged and should be ready for the upload. Since the webservers generate data continuously, it is a very difficult task.

What we need here is a solutions that can overcome the drawbacks of **put** command and transfer the "streaming data" from data generators to centralized stores (especially HDFS) with less delay.

Problem with HDFS

In HDFS, the file exists as a directory entry and the length of the file will be considered as zero till it is closed. For example, if a source is writing data into HDFS and the network was interrupted in the middle of the operation (without closing the file), then the data written in the file will be lost.

Therefore we need a reliable, configurable, and maintainable system to transfer the log data into HDFS.

Note: In POSIX file system, whenever we are accessing a file (say performing write operation), other programs can still read this file (at least the saved portion of the file). This is because the file exists on the disc before it is closed.

Available Solutions

To send streaming data (log files, events etc..,) from various sources to HDFS, we have the following tools available at our disposal:

Facebook's Scribe

Scribe is an immensely popular tool that is used to aggregate and stream log data. It is designed to scale to a very large number of nodes and be robust to network and node failures.

Apache Kafka

Kafka has been developed by Apache Software Foundation. It is an open-source message broker. Using Kafka, we can handle feeds with high-throughput and low-latency.

Apache Flume

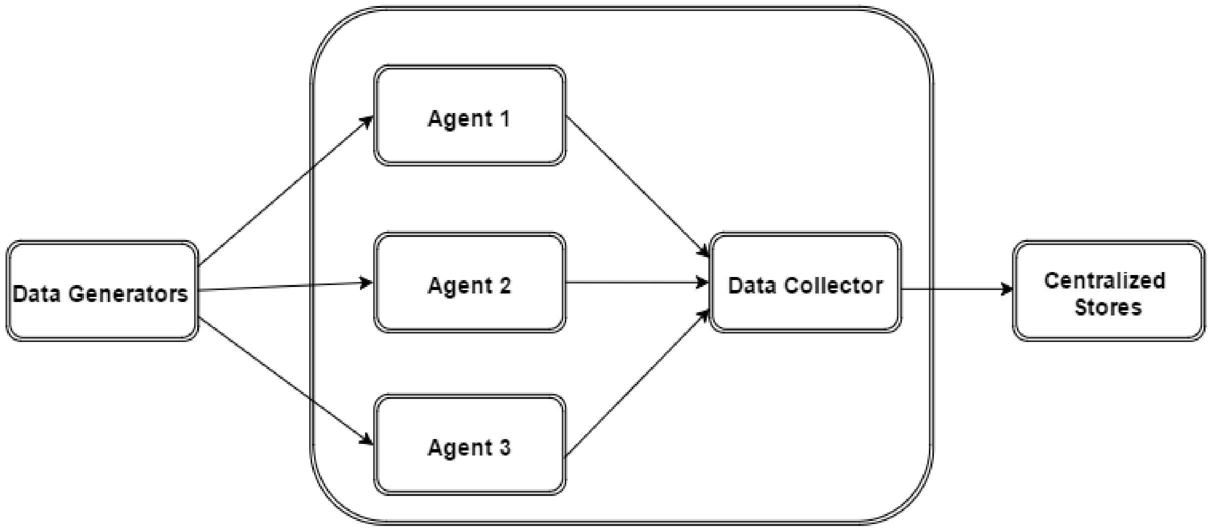
Apache Flume is a tool/service/data ingestion mechanism for collecting aggregating and transporting large amounts of streaming data such as log data, events (etc...) from various webserves to a centralized data store.

It is a highly reliable, distributed, and configurable tool that is principally designed to transfer streaming data from various sources to HDFS.

In this tutorial, we will discuss in detail how to use Flume with some examples.

3. FLUME – ARCHITECTURE

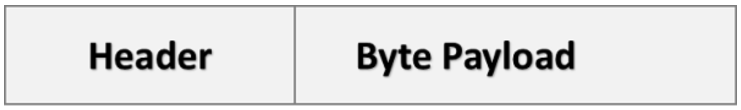
The following illustration depicts the basic architecture of Flume. As shown in the illustration, **data generators** (such as Facebook, Twitter) generate data which gets collected by individual Flume **agents** running on them. Thereafter, a **data collector** (which is also an agent) collects the data from the agents which is aggregated and pushed into a centralized store such as HDFS or HBase.



Flume

Flume Event

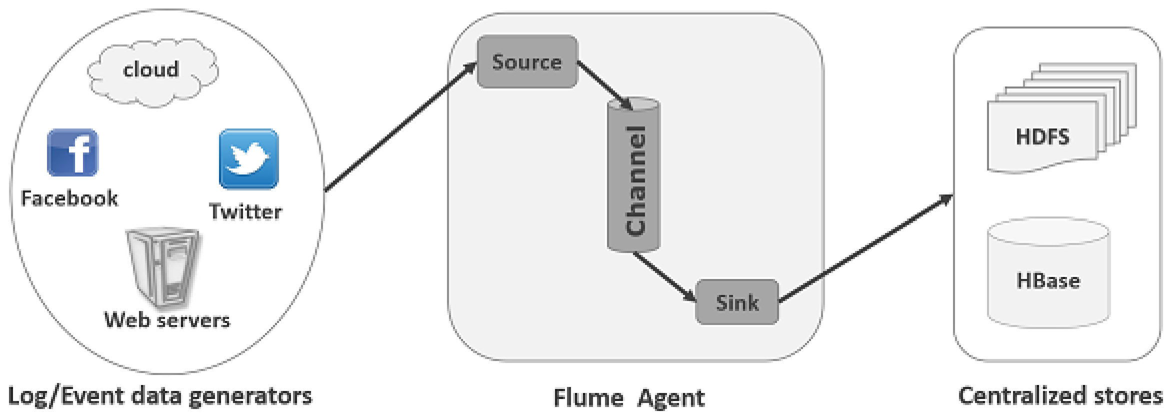
An **event** is the basic unit of the data transported inside **Flume**. It contains a payload of byte array that is to be transported from the source to the destination accompanied by optional headers. A typical Flume event would have the following structure:



Flume event

Flume Agent

Take a look at the following illustration. It shows the internal components of an agent and how they collaborate with each other.



An **agent** is an independent daemon process (JVM) in Flume. It receives the data (events) from clients or other agents and forwards it to their next destination.

A Flume Agent contains three main components namely, **source**, **channel**, and **sink**.

Source

A **source** receives data from the log/event data generators such as Facebook, Twitter, and other webservers, and transfers it to the channel in the form of Flume events.

Data generators like webservers generate data and deliver it to the agent. A **source** is a component of the agent which receives this data and transfers it to one or more channels.

Apache Flume supports several types of sources and each source receives events from a specified data generator. For example, Avro source receives data from the clients which generate data in the form of Avro files.

Flume supports the following sources: Avro, Exec, Spooling directory, Net Cat, Sequence generator, Syslog, Multiport TCP, Syslog UDP, and HTTP.

Channel

A **channel** is a transient store which receives the events from the source and buffers them till they are consumed by sinks. It acts as a bridge between the sources and the sinks.

These channels are fully transactional and they can work with any number of sources and sinks. **Example**: JDBC channel, File system channel, Memory channel, etc.

Sink

Finally, the **sink** stores the data into centralized stores like HBase and HDFS. It consumes the data (events) from the channels and delivers it to the destination. The destination of the sink might be another agent or the central stores. Example: HDFS sink.

Flume supports the following sinks: HDFS sink, Logger, Avro, Thrift, IRC, File Roll, Null sink, HBase, and Morphline solr.

Additional Components of Flume Agent

What we have discussed above are the primitive components of the agent. In addition to this, we have a few more components that play a vital role in transferring the events from the data generator to the centralized stores.

Interceptors

Interceptors are used to alter/inspect flume events which are transferred between source and channel.

Channel Selectors

These are used to determine which channel is to be opted to transfer the data in case of multiple channels. There are two types of channel selectors:

- **Default channel selectors**: These are also known as replicating channel selectors they replicates all the events in each channel.

- **Multiplexing channel selectors**: These decides the channel to send an event based on the address in the header of that event.

Sink Processors

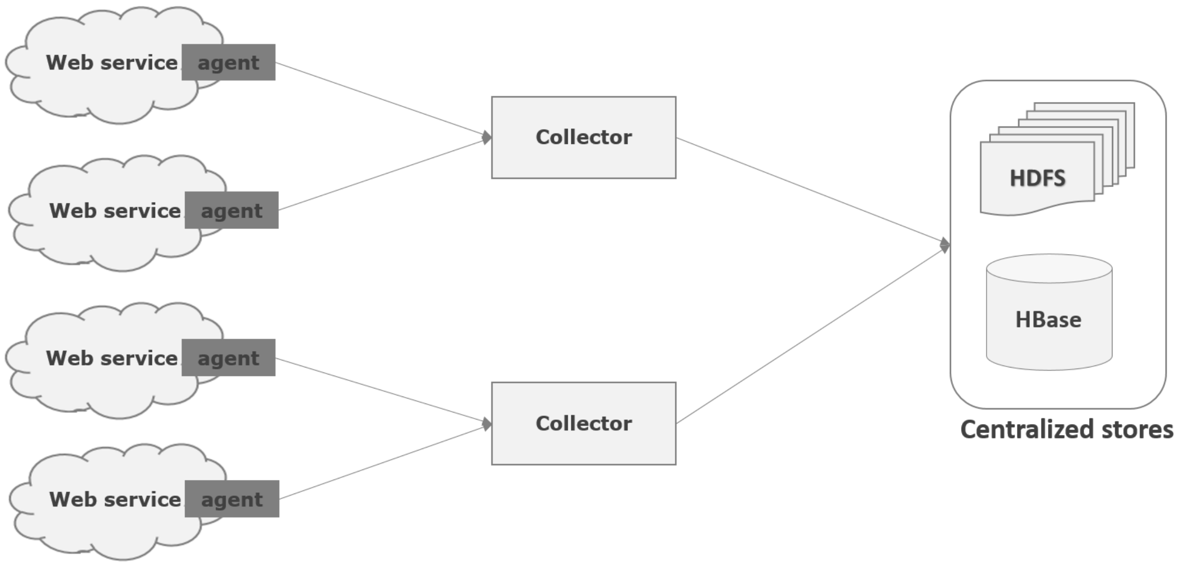
These are used to invoke a particular sink from the selected group of sinks. These are used to create failover paths for your sinks or load balance events across multiple sinks from a channel.

4. FLUME – DATA FLOW

Flume is a framework which is used to move log data into HDFS. Generally events and log data are generated by the log servers and these servers have Flume agents running on them. These agents receive the data from the data generators.

The data in these agents will be collected by an intermediate node known as **Collector**. Just like agents, there can be multiple collectors in Flume.

Finally, the data from all these collectors will be aggregated and pushed to a centralized store such as HBase or HDFS. The following diagram explains the data flow in Flume.



Multi-hop Flow

Within Flume, there can be multiple agents and before reaching the final destination, an event may travel through more than one agent. This is known as **multi-hop flow**.

Fan-out Flow

The dataflow from one source to multiple channels is known as **fan-out flow**. It is of two types:

- **Replicating:** The data flow where the data will be replicated in all the configured channels.

- **Multiplexing:** The data flow where the data will be sent to a selected channel which is mentioned in the header of the event.

Fan-in Flow

The data flow in which the data will be transferred from many sources to one channel is known as **fan-in flow**.

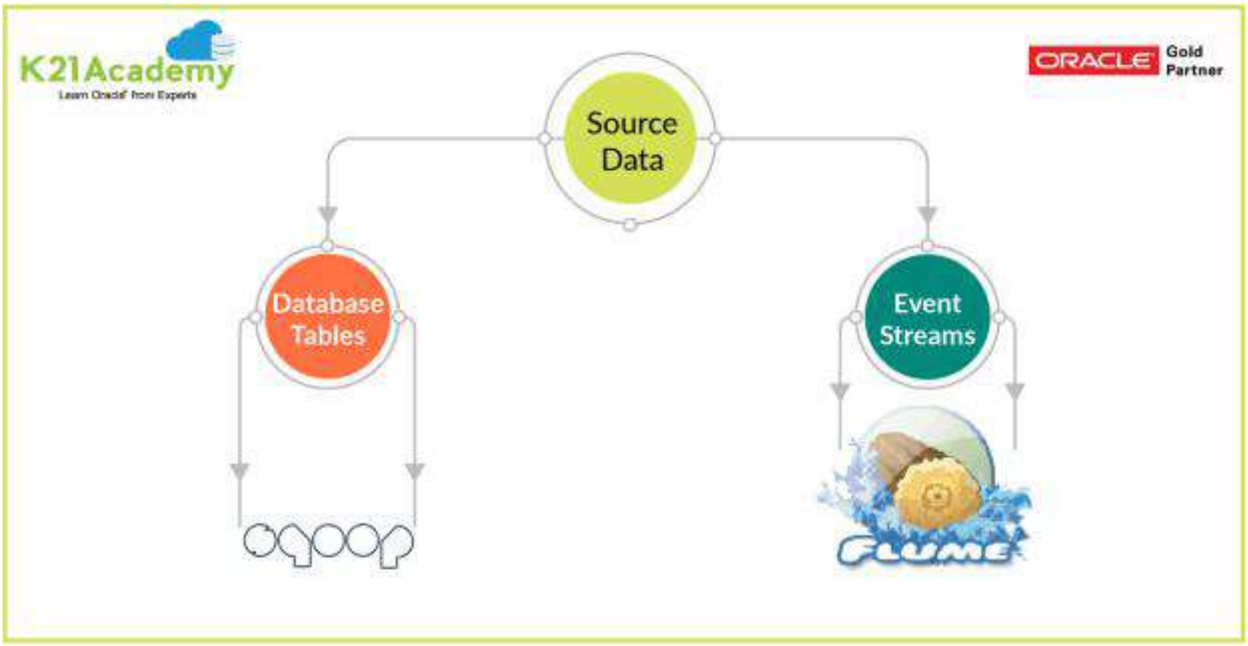
Failure Handling

In Flume, for each event, two transactions take place: one at the sender and one at the receiver. The sender sends events to the receiver. Soon after receiving the data, the receiver commits its own transaction and sends a "received" signal to the sender. After receiving the signal, the sender commits its transaction. (Sender will not commit its transaction till it receives a signal from the receiver.)

DATA INGESTION WITH SQOOP and FLUME

Introduction

Apache Hadoop is synonymous with big data for its cost-effectiveness and its attribute of scalability for processing petabytes of data. Data analysis using Hadoop is just half the battle won. Getting data into the Hadoop cluster plays a critical role in any big data deployment.



Data ingestion is important in any big data project because the volume of data is generally in petabytes or exabytes. **Hadoop Sqoop** and **Hadoop Flume** are the **two tools in Hadoop** which is used to gather data from different sources and load them into HDFS. **Sqoop in Hadoop** is mostly used to extract structured data from databases like Teradata, Oracle, etc., and **Flume in Hadoop** is used to sources data which is stored in various sources like and deals mostly with unstructured data.

Apache Sqoop and **Apache Flume** are two popular open-source tools for Hadoop that help organizations overcome the challenges encountered in data ingestion.
While working on **Hadoop**, there is always one question occurs that if both **Sqoop** and **Flum**e are used to gather data from different sources and load them into HDFS so why we are using both of them.

What is Apache Sqoop?

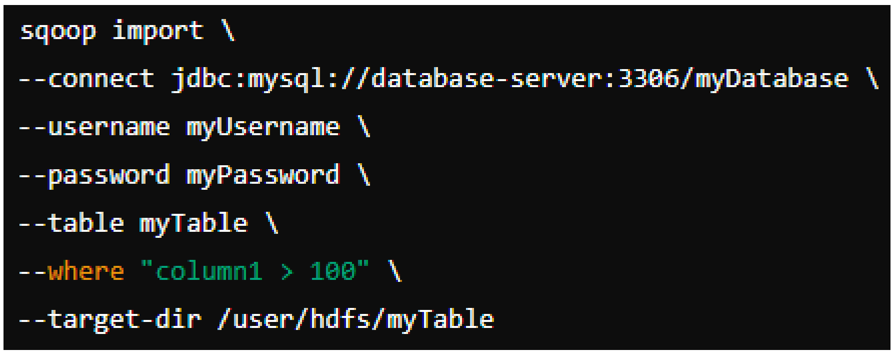
Apache Sqoop is a lifesaver in moving data from the data warehouse into the Hadoop environment. Interestingly it named Sqoop as **SQL-to-Hadoop**. Basically, for importing data from RDBMS's like MySQL, Oracle, etc. into HBase, Hive or HDFS. Apache Sqoop is an effective Hadoop tool.

Apache Sqoop (which is a portmanteau for “sql-to-hadoop”) is an open-source tool that allows users to extract data from a structured data store into Hadoop for further processing. This processing can be done with MapReduce programs or other higher-level tools such as Hive, Pig or Spark.

Sqoop can automatically create Hive tables from imported data from a RDBMS (Relational Database Management System) table.
Sqoop can also be used to send data from Hadoop to a relational database, useful for sending results processed in Hadoop to an operational transaction processing system.

SQOOP Operations
Sqoop includes tools for the following operations:

- Listing databases and tables on a database system
- Importing a single table from a database system, including specifying which columns to import and specifying which rows to import using a WHERE clause



```
sqoop import \
--connect jdbc:mysql://database-server:3306/myDatabase \
--username myUsername \
--password myPassword \
--table myTable \
--where "column1 > 100" \
--target-dir /user/hdfs/myTable
```

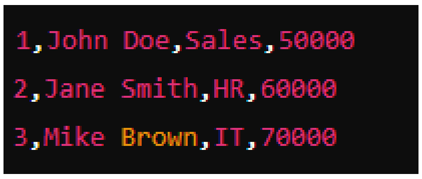
- Importing data from one or more tables using a SELECT statement
- Incremental imports from a table on a database system (importing only what has changed since a known previous state)
- Exporting of data from HDFS to a table on a remote database system.

Sqoop Connectors

Sqoop has an extension framework that makes it possible to **import** data from — and **export** data to — any external storage system that has **bulk data transfer** capabilities.
A Sqoop connector is a modular component that uses this framework to enable Sqoop imports and exports.
Sqoop ships with connectors for working with a range of popular databases, including MySQL, PostgreSQL, Oracle, SQL Server, DB2, and Netezza.
As well as the built-in Sqoop connectors, various third-party connectors are available for data stores, ranging from enterprise data warehouses (such as Teradata) to NoSQL stores (such as Couchbase).

Sqoop is capable of importing into a few different file formats.

By default, Sqoop will generate comma-delimited text files for our imported data. Delimiters can be specified explicitly.



```
1,John Doe,Sales,50000
2,Jane Smith,HR,60000
3,Mike Brown,IT,70000
```

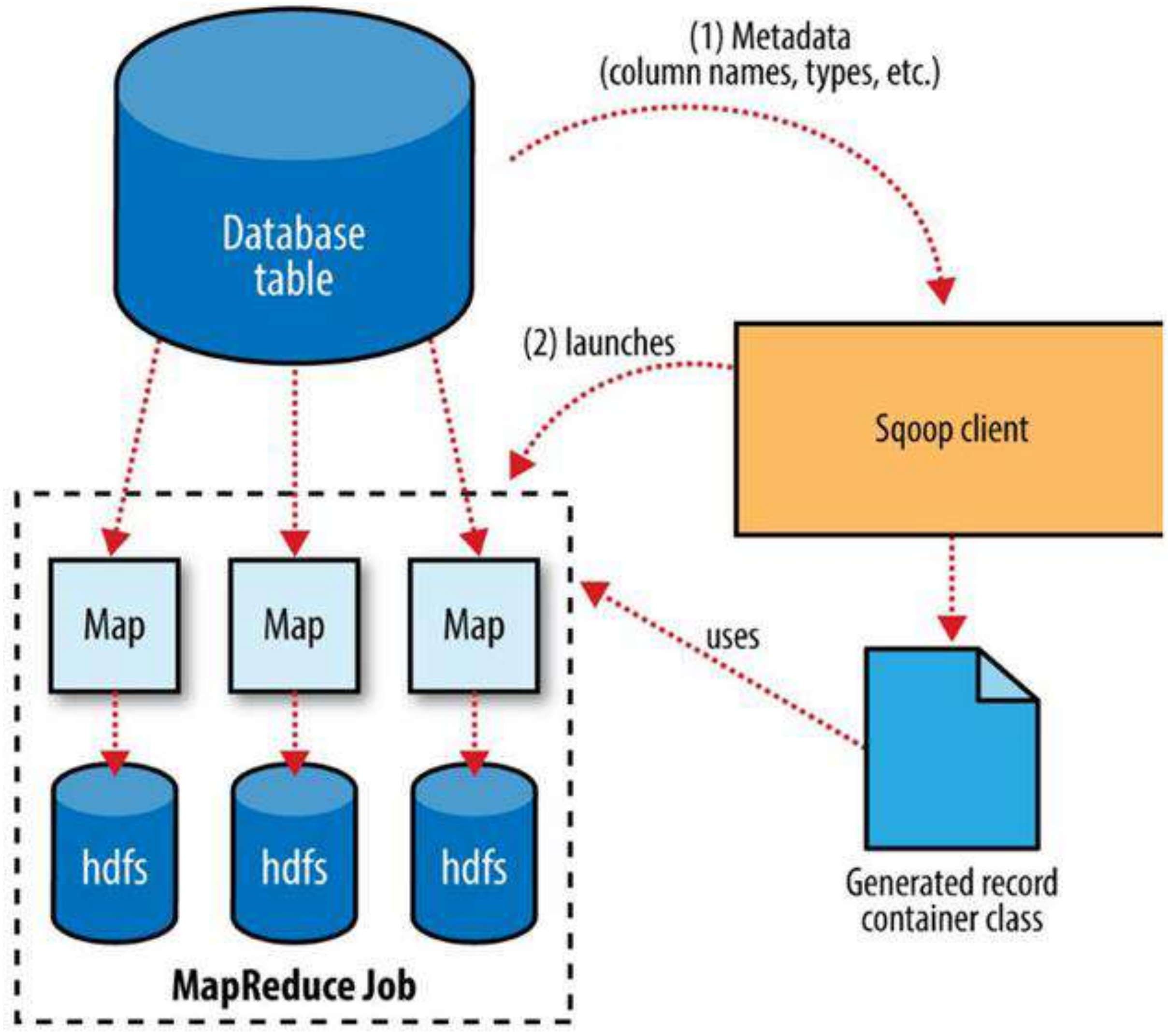
Sqoop also supports SequenceFiles, Avro datafiles, and Parquet files.

How Sqoop works

Sqoop is an abstraction for MapReduce, meaning it takes a command, such as a request to import a table from an RDBMS into HDFS, and implements this using a MapReduce processing routine. Specifically, Sqoop implements a **Map-only** MapReduce process.

Sqoop performs the following steps to complete an import operation:

1. Connect to the database system using JDBC or a customized connector.
2. Examine the table to be imported.
3. Create a Java class to represent the structure (schema) for the specified table. This class can then be reused for future import operations.
4. Execute a Map-only MapReduce job with a specified number of tasks (mappers) to connect to the database system and import data from the specified table in parallel.



Importing Data

This figure illustrates the steps Sqoop takes to import data from a database table into HDFS.

1. **Metadata Retrieval**:
   - Sqoop connects to the database and retrieves metadata about the table, including column names, types, and other schema details.
2. **MapReduce Job Launch**:
   - Sqoop launches a Map-only MapReduce job to perform the data import. This job consists of multiple mappers, each responsible for importing a portion of the table data in parallel.
3. **Data Import**:
   - Each mapper connects to the database, retrieves a portion of the data, and writes it to HDFS. The parallelism provided by multiple mappers speeds up the data import process.

4. **Generated Record Container Class**:
    - Sqoop generates a Java class that represents the structure of the table being imported. This class is used by the mappers to understand the schema and correctly import the data.

Sqoop Export Process

In Sqoop, exporting data involves moving data from HDFS to a relational database. This process is particularly useful for making the results of Hadoop-based analyses available to other tools or applications that work with relational databases.

Steps for Exporting Data:

1. **Prepare the Database:**
    - Before exporting data from HDFS to a database, ensure that the target table exists in the database and has the appropriate schema to receive the data. This step often involves manually creating the table with the correct column types.
2. **Connect to the Database:**
    - Sqoop connects to the database using JDBC or a custom connector.
3. **Generate Java Class:**

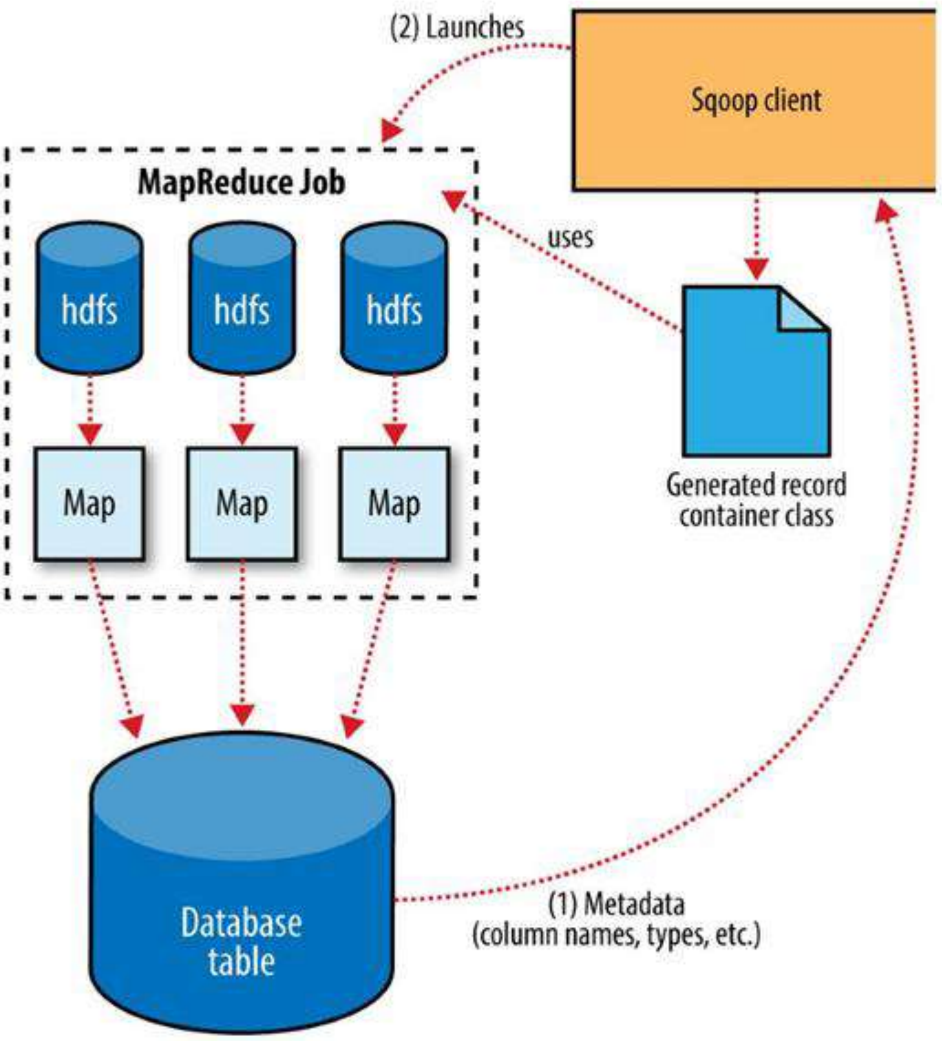
- Sqoop generates a Java class based on the target table definition. This class is capable of parsing records from the text files in HDFS and inserting values of the appropriate types into the table.

4. **Specify Delimiters:**

- When reading the tables directly from files, specify the field and record delimiters used in the text files stored in HDFS. Sqoop assumes records are newline-delimited by default.

5. **Launch MapReduce Job:**

- Sqoop launches a Map-only MapReduce job to read the source data files from HDFS, parse the records using the generated class, and execute the export strategy to insert the data into the database table.


(2) Launches
Sqoop client
MapReduce Job
uses
hdfs
hdfs
hdfs
Map
Map
Map
Generated record container class
Database table
(1) Metadata (column names, types, etc.)

Compression and Serialization

File Compression in Hadoop

File Compression Benefits:

1. **Storage Space Reduction:**
   - Compression reduces the amount of space needed to store files.
   - This is particularly beneficial when dealing with large volumes of data.
2. **Faster Data Transfer:**
   - Compressed files can be transferred more quickly across networks.
   - They also speed up reading and writing to and from disks.

Given the large datasets often handled in Hadoop, efficient compression can lead to significant storage and performance improvements.

Compression Formats, Tools, and Algorithms:

- Various compression formats, each with unique characteristics, can be used with Hadoop.

Common Compression Formats Used in Hadoop:

Compression Format	Tool	Algorithm	Filename Extension	Splittable
DEFLATE	N/A	DEFLATE	.deflate	No
gzip	gzip	DEFLATE	.gz	No
bzip2	bzip2	bzip2	.bz2	Yes
LZO	lzop	LZO	.lzo	No
LZ4	N/A	LZ4	.lz4	No
Snappy	N/A	Snappy	.snappy	No

Choosing the Right Compression Format:

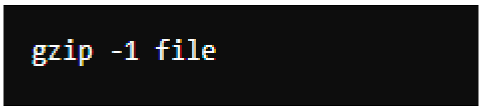
- The choice of compression format depends on the specific requirements of storage efficiency, speed, and whether the data needs to be splittable for parallel processing.
- For large files that need to be processed in parallel, splittable formats like bzip2 are advantageous.
- For scenarios where speed is critical, formats like LZO and Snappy might be more suitable.

Space/Time Trade-Off in Compression:

- All compression algorithms balance space and time:
    - **Faster compression/decompression**: Usually results in larger files.
    - **More effective compression (smaller files)**: Typically, slower compression/decompression speeds.

Control Over Compression Trade-Off:

- Tools offer different options for compression:
    - **-1 Option**: Optimize for speed.
    - **-9 Option**: Optimize for space.
    - Example: To create a compressed file using the fastest method:



```
gzip -1 file
```

Characteristics of Compression Tools:

- **gzip**: General-purpose compressor, balanced space/time trade-off.
- **bzip2**: Compresses more effectively than gzip but is slower. Decompression is faster than compression but still slower compared to others.
- **LZO, LZ4, Snappy**: Optimize for speed, much faster than gzip but less effective in compression.
    - **LZ4 and Snappy**: Faster decompression compared to LZO.

Splittable Compression Formats:

- **Splittable**: Allows seeking to any point in the stream and starting reading from there.
- Splittable formats are beneficial for MapReduce operations in Hadoop.

Codecs in Hadoop:

- **Codec**: Implementation of a compression-decompression algorithm.
- In Hadoop, a codec is represented by the CompressionCodec interface.
- **Common Hadoop Codecs:**

Compression Format	**Hadoop CompressionCodec**
DEFLATE	org.apache.hadoop.io.compress.DefaultCodec
gzip	org.apache.hadoop.io.compress.GzipCodec

Compression Format	Hadoop CompressionCodec
bzip2	org.apache.hadoop.io.compress.BZip2Codec
LZO	com.hadoop.compression.lzo.LzopCodec
LZ4	org.apache.hadoop.io.compress.Lz4Codec
Snappy	org.apache.hadoop.io.compress.SnappyCodec

Compressing and Decompressing Streams:

- **CompressionCodec Methods:**
  - createOutputStream(OutputStream out): Creates a CompressionOutputStream to write uncompressed data and store it in compressed form.
  - createInputStream(InputStream in): Creates a CompressionInputStream to read uncompressed data from a compressed input stream.
- CompressionOutputStream and CompressionInputStream are similar to java.util.zip.DeflaterOutputStream and java.util.zip.DeflaterInputStream.
  - They provide the ability to reset the underlying compressor or decompressor, useful for applications compressing sections of a data stream as separate blocks (e.g., in a SequenceFile).

```
OutputStream out = new FileOutputStream("output.txt");
CompressionCodec codec = new GzipCodec();
CompressionOutputStream compressedOut = codec.createOutputStream(out);
// Write compressed data to the file
compressedOut.write(data);
compressedOut.close();
```

Native Libraries for Compression in Hadoop

Performance Benefits:

- Native libraries for compression and decompression offer significant performance improvements.
  - **Decompression**: Up to 50% faster using native gzip libraries.
  - **Compression**: Around 10% faster compared to the built-in Java implementation.

Availability of Implementations:

- Not all compression formats have both Java and native implementations.

Compression Format	Java Implementation	Native Implementation
DEFLATE	Yes	Yes
gzip	Yes	Yes
bzip2	Yes	Yes
LZO	No	Yes
LZ4	No	Yes
Snappy	No	Yes

Compression and Input Splits in Hadoop

Importance of Splittable Compression Formats:

- Splittable formats allow for efficient MapReduce processing by enabling files to be divided into independent splits, processed by separate map tasks.

Example Scenario:

1. **Uncompressed File:**
    - Size: 1 GB.
    - HDFS Block Size: 128 MB.
    - Storage: The file is stored as eight 128 MB blocks in HDFS.
    - Processing: A MapReduce job creates eight input splits, each processed by a separate map task.
2. **gzip-Compressed File:**
    - Compressed Size: 1 GB.
    - Storage: Stored as eight blocks in HDFS.
    - Processing Issue: gzip does not support splitting because:
        - gzip (using DEFLATE) stores data in compressed blocks without identifiable start points.
        - A map task cannot independently read its split.
    - Solution: MapReduce processes the entire file as a single split, sacrificing data locality and increasing processing time.

Handling Non-Splittable Compression Formats:

- **LZO:**
    - Issue: Similar to gzip, LZO does not support splitting directly.

    - Solution: Preprocess LZO files using an indexer tool (available in Hadoop LZO libraries from Google and GitHub).
        - The tool builds an index of split points, making the files effectively splittable when using the appropriate MapReduce input format.

Splittable Compression Formats:

- **bzip2:**
    - Supports splitting by including a synchronization marker (a 48-bit approximation of pi) between blocks.
    - This marker allows a reader to synchronize and split the stream, enabling independent processing of each split by map tasks.

Considerations for MapReduce Jobs:

- **Non-Splittable Formats**: Result in fewer map tasks and can increase job runtime due to processing large splits and reduced data locality.
- **Splittable Formats**: Enable more efficient parallel processing by allowing data to be divided into independent splits.

Serialization in Hadoop

Definition:

- **Serialization**: The process of converting structured objects into a byte stream for network transmission or persistent storage.
- **Deserialization**: The reverse process of converting a byte stream back into structured objects.

Uses in Distributed Data Processing:

In distributed data processing, such as with Hadoop, serialization and deserialization are crucial for two main uses: communicating between different parts of the system and storing data.

1. **Interprocess Communication:**
    - **Remote Procedure Calls (RPCs)**: Nodes communicate by sending serialized messages.
    - **Deserialization**: The remote node deserializes the binary stream back into the original message.
2. **Persistent Storage:**
    - Data is serialized for storage and deserialized when read back.

Desirable Properties of an RPC Serialization Format:

1. **Compact:**
    - Efficient use of network bandwidth.
    - Essential in a data center where bandwidth is a scarce resource.
2. **Fast:**
    - Minimal performance overhead for serialization/deserialization.
    - Crucial for maintaining the backbone performance of a distributed system.
3. **Extensible:**
    - Protocols should evolve easily to meet new requirements.
    - Possible to add new arguments to method calls without breaking compatibility with older clients/servers.
4. **Interoperable:**
    - Support for clients written in different languages than the server.
    - Format design must facilitate cross-language compatibility.

Requirements for Persistent Storage Formats:

- **Compact**: Efficient use of storage space.
- **Fast**: Minimal overhead for reading/writing large volumes of data.

- **Extensible**: Ability to transparently read data written in older formats.
- **Interoperable**: Compatibility with different languages for reading/writing data.

Hadoop's Serialization Format:

Hadoop primarily uses its own serialization format called Writables for its internal data transfer and storage. However, it also supports other serialization frameworks such as Avro, Thrift, and Protocol Buffers.

Writables:

Writable is the core serialization format in Hadoop. It is used for both interprocess communication (e.g., RPC calls) and persistent storage within the Hadoop ecosystem.

- **Compact and Efficient**: Writables are designed to be compact and efficient, minimizing the overhead of serialization and deserialization.
- **Binary Format**: Data is stored in a binary format, which is both space-efficient and quick to read/write.
- **Java-centric**: Writables are designed primarily for use with Java, and the types are represented as Java objects.

Limitations:

    - Not easily extensible.
    - Limited interoperability with languages other than Java.

Importance of Writables in Hadoop:

- Central to Hadoop's operations, especially in MapReduce programs where they are used for key and value types.

Other Serialization Frameworks in Hadoop:

- **Avro**:
    - Designed to address some limitations of Writables.

Serialization Frameworks in Hadoop

Overview:

- While Hadoop's Writable types are commonly used in MapReduce programs, any type can be used as long as it can be serialized to and deserialized from a binary representation.
- Hadoop supports pluggable serialization frameworks through its Serialization API.

Key Components:

1. **Serialization Frameworks:**
    - Represented by an implementation of the Serialization interface in the org.apache.hadoop.io.serializer package.
    - Example: WritableSerialization is the default implementation for Writable types.
2. **Serialization and Deserialization:**
    - A Serialization framework maps types to Serializer instances (to turn objects into byte streams) and Deserializer instances (to turn byte streams into objects).
3. **Configuration:**
    - The io.serializations property can be set to a comma-separated list of class names to register custom Serialization implementations.
    - Default value includes:
        - org.apache.hadoop.io.serializer.WritableSerialization
        - Avro Specific and Reflect serializations
    - This means out-of-the-box support for Writable or Avro objects.
4. **Java Serialization:**
    - The JavaSerialization class uses Java Object Serialization.
    - While it allows the use of standard Java types (e.g., Integer, String), it is less efficient than Writables.
5. **Other Serialization Frameworks:**
    - **Apache Thrift** and **Google Protocol Buffers**:
        - Both are popular for defining types in a language-neutral, declarative fashion using an Interface Description Language (IDL).
        - Useful for interoperability and versioning.
        - Limited support in MapReduce formats but used internally in Hadoop for RPC and data exchange.

Why Consider Alternative Serialization Frameworks:

- **Efficiency**: Writable serialization is optimized for performance, but other frameworks like Apache Thrift or Protocol Buffers can offer better interoperability and versioning support.
- **Interoperability**: These frameworks are designed to support multiple languages and systems, making them ideal for cross-language data exchange.
- **Versioning**: They typically have built-in mechanisms for evolving data structures over time.

Writable Interface

This is the interface in Hadoop which provides methods for serialization and deserialization. The following table describes the methods:

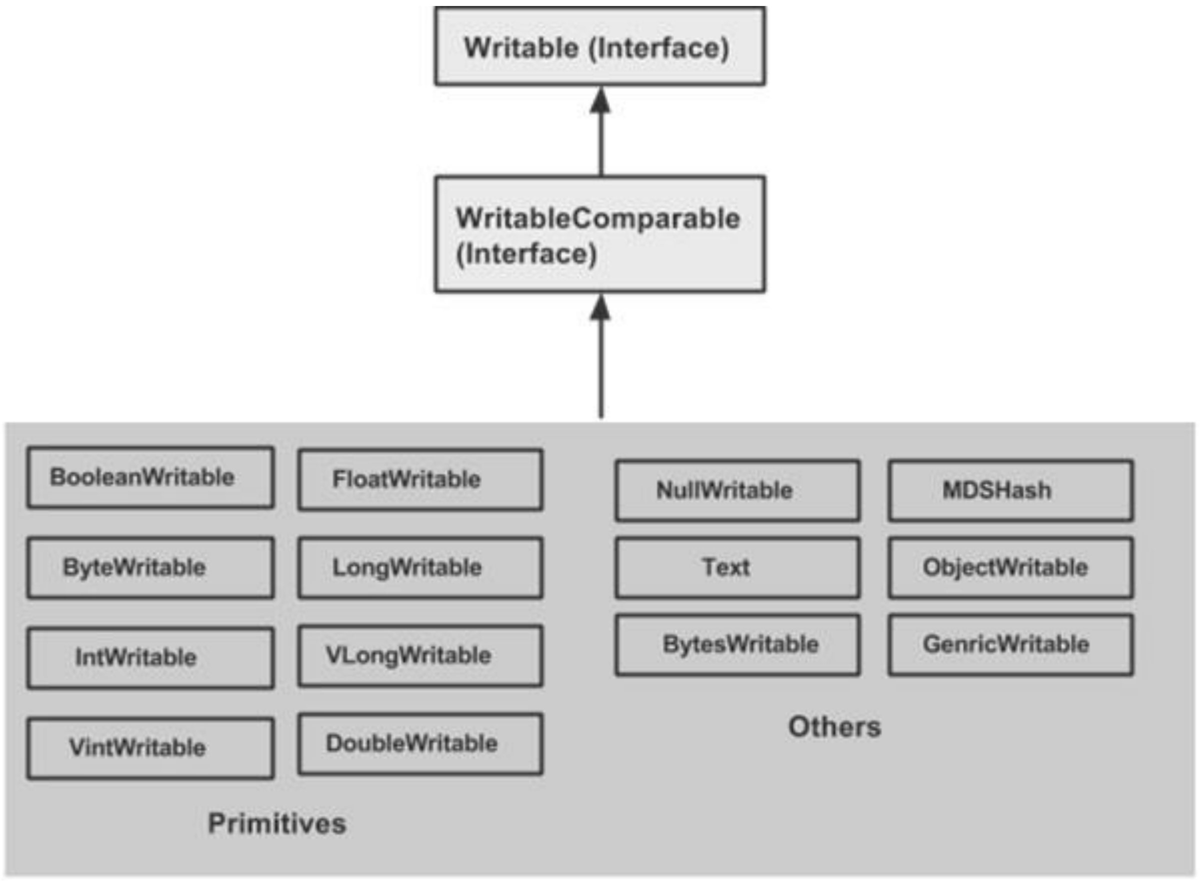
S. No.	Methods and Description
1	**void readFields(DataInput in)** This method is used to deserialize the fields of the given object.
2	**void write(DataOutput out)** This method is used to serialize the fields of the given object.

Writable Comparable Interface

It is the combination of **Writable** and **Comparable** interfaces. This interface inherits **Writable** interface of Hadoop as well as **Comparable** interface of Java. Therefore it provides methods for data serialization, deserialization, and comparison.

S. No.	Methods and Description
1	**int compareTo(class obj)** This method compares current object with the given object obj.

In addition to these classes, Hadoop supports a number of wrapper classes that implement WritableComparable interface. Each class wraps a Java primitive type. The class hierarchy of Hadoop serialization is given below:



These classes are useful to serialize various types of data in Hadoop. For instance, let us consider the **IntWritable** class. Let us see how this class is used to serialize and deserialize the data in Hadoop.

IntWritable Class

This class implements **Writable**, **Comparable**, and **WritableComparable** interfaces. It wraps an integer data type in it. Shortly, it provides methods used to serialize and deserialize integer type of data.

Constructors

S. No.	Summary
1	**IntWritable()**
2	**IntWritable(int value)**

Methods

S. No.	Summary
1	**int get()** Using this method you can get the integer value present in the current object.
2	**void readFields(DataInput in)** This method is used to deserialize the data in the given **DataInput** object.
3	**void set(int value)** This method is used to set the value of the current **IntWritable** object.
4	**void write(DataOutput out)** This method is used to serialize the data in the current object to the given **DataOutput** object.

Serializing the Data in Hadoop

The procedure to serialize the integer type of data is discussed below.

1. Instantiate **IntWritable** class by wrapping an integer value in it.

2. Instantiate **ByteArrayOutputStream** class.

3. Instantiate **DataOutputStream** class and pass the object of **ByteArrayOutputStream** class to it.

4. Serialize the integer value in IntWritable object using **write()** method. This method needs an object of DataOutputStream class.
5.
6. The serialized data will be stored in the byte array object which is passed as parameter to the **DataOutputStream** class at the time of instantiation. Convert the data in the object to byte array.

Example

The following example shows how to serialize data of integer type in Hadoop:

```
import java.io.ByteArrayOutputStream;

import java.io.DataOutputStream;

import java.io.IOException;


import org.apache.hadoop.io.IntWritable;


public class Serialization {
   public byte[] serialize() throws IOException{

     //Instantiating the IntWritable object
     IntWritable intwritable = new IntWritable(12);

     //Instantiating ByteArrayOutputStream object
     ByteArrayOutputStream byteoutputStream = new ByteArrayOutputStream();

     //Instantiating DataOutputStream object
     DataOutputStream dataOutputStream = new
                       DataOutputStream(byteoutputStream);

     //Serializing the data
     intwritable.write(dataOutputStream);

     //storing the serialized object in bytearray
     byte[] byteArray = byteoutputStream.toByteArray();

     //Closing the OutputStream
     dataOutputStream.close();

     return(byteArray);
     }
     public static void main(String args[]) throws IOException{
           Serialization serialization= new Serialization();
           serialization.serialize();
           System.out.println();
     }
}
```

Deserializing the Data in Hadoop

The procedure to deserialize the integer type of data is discussed below:

1. Instantiate **IntWritable** class by wrapping an integer value in it.

2. Instantiate **ByteArrayInputStream** class.

3. Instantiate **DataInputStream** class, and pass the object of **ByteArrayInputStream** class to it.

4. Deserialize the data in the object of **DataInputStream** using **readFields()** method of IntWritable class.

5. The deserialized data will be stored in the object of IntWritable class. You can retrieve this data using **get()** method of this class.

Example

The following example shows how to deserialize the data of integer type in Hadoop:

```
import java.io.ByteArrayInputStream;
import java.io.DataInputStream;
import org.apache.hadoop.io.IntWritable;

public class Deserialization {
   public void deserialize(byte[]byteArray) throws Exception{
     //Instantiating the IntWritable class
     IntWritable intwritable =new IntWritable();

     //Instantiating ByteArrayInputStream object
     ByteArrayInputStream InputStream = new ByteArrayInputStream(byteArray);

     //Instantiating DataInputStream object
     DataInputStream datainputstream=new DataInputStream(InputStream);

     //deserializing the data in DataInputStream
     intwritable.readFields(datainputstream);

     //printing the serialized data
     System.out.println((intwritable).get());
     }
     public static void main(String args[]) throws Exception {

           Deserialization dese = new Deserialization();
           dese.deserialize(new Serialization().serialize());
```

```

      }
}
```

Advantage of Hadoop over Java Serialization

Hadoop's Writable-based serialization is capable to reduce the object-creation overhead by reusing the Writable objects, which is not possible with the Java's native serialization framework.

Disadvantages of Hadoop Serialization

To serialize Hadoop data, there are two ways:

- You can use the **Writable** classes, provided by Hadoop's native library.
- You can also use **Sequence Files** which store the data in binary format.

The main drawback of these two mechanisms is that **Writables** and **SequenceFiles** have only a Java API and they cannot be written or read in any other language.

Therefore any of the files created in Hadoop with above two mechanisms cannot be read by any other third language, which makes Hadoop as a limited box. To address this drawback, Doug Cutting created **Avro,** which is a **language independent data structure.**

1. Avro – Overview

To transfer data over a network or for its persistent storage, you need to serialize the data. Prior to the **serialization APIs** provided by Java and Hadoop, we have a special utility, called **Avro**, a schema-based serialization technique.

This tutorial teaches you how to serialize and deserialize the data using Avro. Avro provides libraries for various programming languages. In this tutorial, we demonstrate the examples using Java library.

What is Avro?

Apache Avro is a language-neutral data serialization system. It was developed by Doug Cutting, the father of Hadoop. Since Hadoop writable classes lack language portability, Avro becomes quite helpful, as it deals with data formats that can be processed by multiple languages. Avro is a preferred tool to serialize data in Hadoop.

Avro has a schema-based system. A language-independent schema is associated with its read and write operations. Avro serializes the data which has a built-in schema. Avro serializes the data into a compact binary format, which can be deserialized by any application.

Avro uses JSON format to declare the data structures. Currently it supports languages such as Java, C, C++, C#, Python, and Ruby.

Avro Schemas

Avro depends heavily on its **schema**. It allows every data to be written with no prior knowledge of the schema. It serializes fast and the resulting serialized data is lesser in size. Schema is stored along with the Avro data in a file for any further processing.

In RPC, the client and the server exchange schemas during the connection. This exchange helps in the communication between same named fields, missing fields, extra fields, etc. Both the old and new schemas are always present to resolve any differences.

Avro schemas are defined with JSON that simplifies its implementation in languages with JSON libraries.

Like Avro, there are other serialization mechanisms in Hadoop such as **Sequence Files**, **Protocol Buffers**, and **Thrift**.

Comparison with Thrift and Protocol Buffers

Thrift and **Protocol Buffers** are the most competent libraries of Avro. Avro differs from these frameworks in the following ways:

- Avro supports both dynamic and static *types* as per the requirement. Protocol Buffers and Thrift use Interface Definition Languages (IDLs) to specify schemas and their types. These IDLs are used to generate code for serialization and deserialization.

- Avro is built in the Hadoop ecosystem. Thrift and Protocol Buffers are not built in Hadoop ecosystem.

Unlike Thrift and Protocol Buffer, Avro's schema definition is in JSON and not in any proprietary IDL; that makes it language neutral.

Property	Avro	Thrift and Protocol Buffer
Dynamic schema	Yes	No
Built into Hadoop	Yes	No
Schema in JSON	Yes	No
No need to compile	Yes	No
No need to declare IDs	Yes	No
Bleeding edge	Yes	No

Features of Avro

Listed below are some of the prominent features of Avro:

- Avro is a **language-neutral** data serialization system.

- It can be processed by many languages (currently C, C++, C#, Java, Python, and Ruby).

- Avro creates binary structured format that is both **compressible** and **splittable**. Hence it can be efficiently used as the input to Hadoop MapReduce jobs.

- Avro provides **rich data structures.** For example, you can create a record that contains an array, an enumerated type, and a sub record. These datatypes can be created in any language, can be processed in Hadoop, and the results can be fed to a third language.

- Avro **schemas** defined in **JSON** facilitate implementation in the languages that already have JSON libraries.

- Avro creates a self-describing file named *Avro Data File*, in which it stores data along with its schema in the metadata section.

- Avro is also used in Remote Procedure Calls (RPCs). During RPC, client and server exchange schemas in the connection handshake.

- Avro does not need code generation. The data is always accompanied by schemas, which permit full processing on the data.

General Working of Avro

To use Avro, you need to follow the given workflow:

- **Step 1**: Create schemas. Here you need to design Avro schema according to your data.

- **Step 2**: Read the schemas into your program. It is done in two ways:

    - **By Generating a Class Corresponding to Schema –** Compile the schema using Avro. This generates a class file corresponding to the schema.

    - **By Using Parsers Library –** You can directly read the schema using parsers library.

- **Step 3**: Serialize the data using the serialization API provided for Avro, which is found in the package **org.apache.avro.specific.**

- **Step 4**: Deserialize the data using deserialization API provided for Avro, which is found in the package **org.apache.avro.specific.**

Avro Serialization and Deserialization

1. Defining the Schema

The schema defines the structure of the data. For Avro, this is usually in JSON format.

Schema Definition (JSON):

json Copy code

```json
{
  "type": "record",
  "name": "Person",
  "fields": [
    {"name": "name", "type": "string"},
    {"name": "age", "type": "int"},
    {"name": "email", "type": "string"}
  ]
}
```

Pseudo Code for Serialization (Python):

python Copy code

```python
import avro.schema
import avro.io
import io

# Load the schema
schema = avro.schema.parse(open("person.avsc", "r").read())

# Create an Avro writer
writer = avro.io.DatumWriter(schema)

# Create a buffer to hold the serialized data
buffer = io.BytesIO()

# Create an Avro encoder
encoder = avro.io.BinaryEncoder(buffer)
```

```
# Create a sample Person record
person = {
    "name": "Alice",
    "age": 30,
    "email": "alice@example.com"
}

# Serialize the record
writer.write(person, encoder)

# Get the serialized data
serialized_data = buffer.getvalue()
```

Pseudo Code for Deserialization (Java):

```java
import org.apache.avro.Schema;
import org.apache.avro.io.DecoderFactory;
import org.apache.avro.io.DatumReader;
import org.apache.avro.io.DatumReader;
import org.apache.avro.generic.GenericDatumReader;
import org.apache.avro.generic.GenericData;

import java.io.ByteArrayInputStream;

// Load the schema
Schema schema = new Schema.Parser().parse(new File("person.avsc"));

// Create an Avro reader
DatumReader<GenericData.Record> reader = new GenericDatumReader<>(schema);

// Create a buffer with serialized data
ByteArrayInputStream inputStream = new ByteArrayInputStream(serializedData);
```

```java
// Create a decoder
Decoder decoder = DecoderFactory.get().binaryDecoder(inputStream, null);

// Deserialize the record
GenericData.Record person = reader.read(null, decoder);

// Access the fields
String name = person.get("name").toString();
int age = (Integer) person.get("age");
String email = person.get("email").toString();
```

File-Based Data Structures in Hadoop

For some applications, you need a specialized data structure to hold your data. For doing MapReduce-based processing, putting each blob of binary data into its own file doesn't scale, so Hadoop developed a number of higher-level containers for these situations.

SequenceFile

SequenceFiles are a specialized data structure in Hadoop designed to handle binary key-value pairs efficiently. They are particularly useful when dealing with large-scale data processing where traditional text formats are inadequate.

Key Concepts

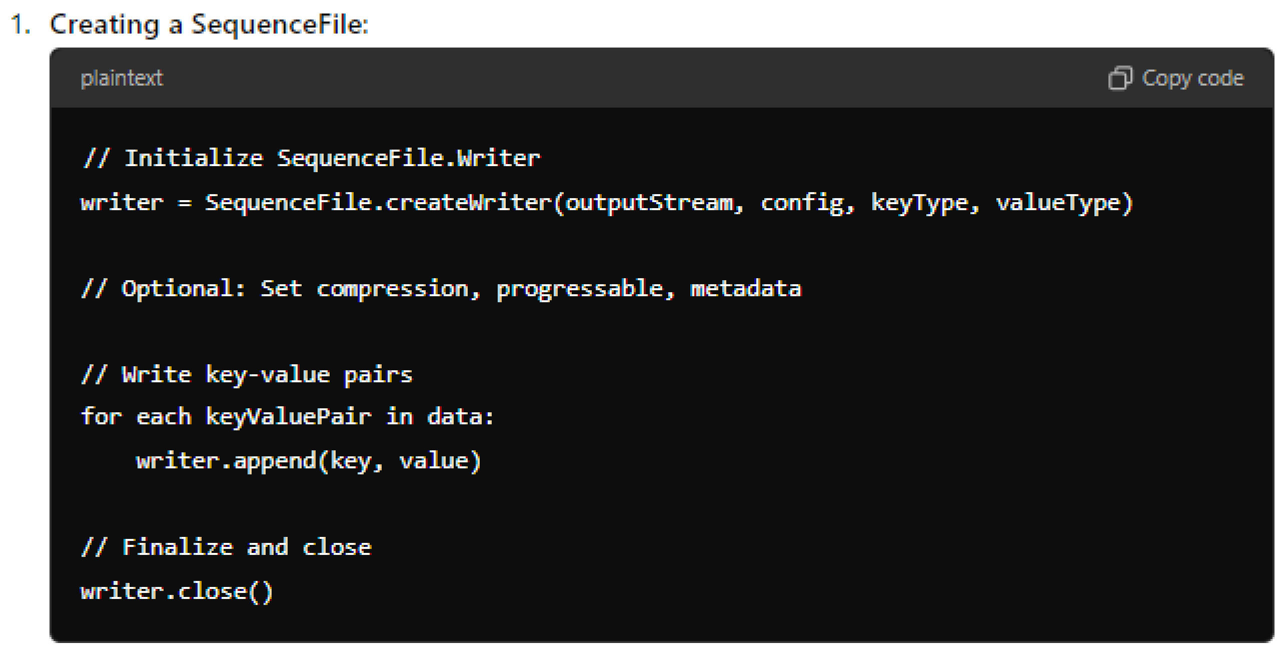
1. **Purpose**:
    - **Logfiles**: Suitable for storing binary types, as opposed to plain text formats.
    - **Containers for Smaller Files**: Packs multiple smaller files into a single SequenceFile, improving storage and processing efficiency.
2. **Structure**:
    - **Key-Value Pairs**: Each record in a SequenceFile is a pair consisting of a key and a value.
    - **Binary Format**: Both keys and values are stored in a binary format.
3. **Use Case**:
    - **Timestamp as Key**: For log files, a key like a timestamp (represented by LongWritable) can be used.
    - **Writable Value**: The value is a Writable type representing the quantity or data being logged.

Creating a SequenceFile

1. **Initialization**:
    - Use createWriter() methods to create an instance of SequenceFile.Writer.
    - Specify the following parameters:
        - **Output Stream**: A FSDataOutputStream or FileSystem and Path.
        - **Configuration**: A Configuration object specifying Hadoop settings.
        - **Key and Value Types**: Define the types for keys and values.
2. **Optional Parameters**:
    - **Compression**: Type and codec for compressing the SequenceFile.
    - **Progressable**: Callback to monitor the progress of writing.
    - **Metadata**: Additional information to be stored in the SequenceFile header.

Writing Data

1. **Writing Key-Value Pairs**:
    - Use the append() method to add key-value pairs to the SequenceFile.
    - After adding all records, call the close() method to finalize and close the file.



1. Creating a SequenceFile:

plaintext   Copy code

```plaintext
// Initialize SequenceFile.Writer
writer = SequenceFile.createWriter(outputStream, config, keyType, valueType)

// Optional: Set compression, progressable, metadata

// Write key-value pairs
for each keyValuePair in data:
    writer.append(key, value)

// Finalize and close
writer.close()
```

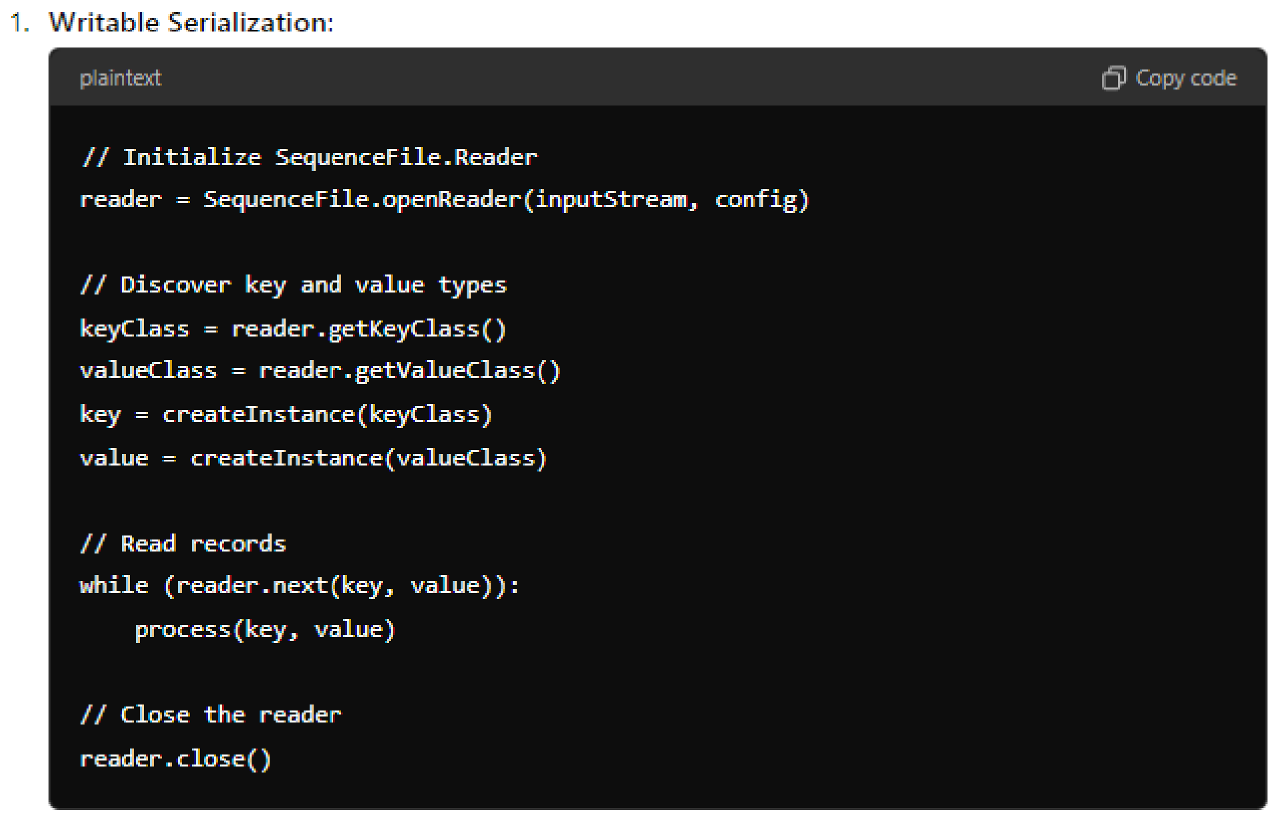
Reading a SequenceFile in Hadoop

When reading a SequenceFile, you need to utilize the SequenceFile.Reader class, which allows you to iterate over the records. The methods used for reading depend on whether you're working with Writable types or a different serialization framework.

Key Concepts

1. **Reading Process**:
    - **Create Instance**: Initialize SequenceFile.Reader to start reading.
    - **Iterate Records**: Use methods to read key-value pairs from the file until the end is reached.
2. **Methods for Reading**:
    - **Writable Serialization**:
        - public boolean next (Writable key, Writable val): Reads the next key-value pair into the provided Writable objects. Returns true if a record was read, and false if the end of the file is reached.
    - **Non-Writable Serialization** (e.g., Apache Thrift):

    - public Object next(Object key) throws IOException: Retrieves the next key from the stream.
    - public Object getCurrentValue(Object val) throws IOException: Retrieves the current value associated with the key.
3. **Serialization Framework**:
    - Ensure that the correct serialization framework is set in the io.serializations property.


1. Writable Serialization:

plaintext Copy code

```plaintext
// Initialize SequenceFile.Reader
reader = SequenceFile.openReader(inputStream, config)

// Discover key and value types
keyClass = reader.getKeyClass()
valueClass = reader.getValueClass()
key = createInstance(keyClass)
value = createInstance(valueClass)

// Read records
while (reader.next(key, value)):
    process(key, value)

// Close the reader
reader.close()
```

Sorting and Merging SequenceFiles in Hadoop

Sorting and merging SequenceFiles can be effectively managed using Hadoop's MapReduce framework. This method leverages Hadoop's parallel processing capabilities to sort and merge files efficiently.

Overview

1. **Sorting**: Organizes data in a specified order based on keys.
2. **Merging**: Combines multiple SequenceFiles into a single output file, maintaining the sort order.

Using MapReduce for Sorting and Merging

MapReduce is particularly suited for this task due to its parallel processing and distributed nature. The process involves:

1. **Mapper Phase**:
    - **Input**: Reads data from one or more SequenceFiles.
    - **Processing**: Emits key-value pairs, which Hadoop then sorts.
2. **Reducer Phase**:
    - **Input**: Receives sorted key-value pairs from the mapper.
    - **Processing**: Writes the sorted data to a new SequenceFile.

Steps to Sort and Merge SequenceFiles

1. **Specify Input and Output Formats**:
    - **Input**: SequenceFiles to be sorted and merged.
    - **Output**: A new SequenceFile with sorted data.
2. **Define Key and Value Types**:
    - Ensure that the key and value types used in the MapReduce job match those in the SequenceFiles.
3. **Configure the Job**:
    - Set up the MapReduce job to handle SequenceFiles.
    - Specify the number of reducers to control the number of output partitions.
4. **Run the Job**:
    - Execute the MapReduce job to perform the sorting and merging.

SequenceFile Format in Hadoop

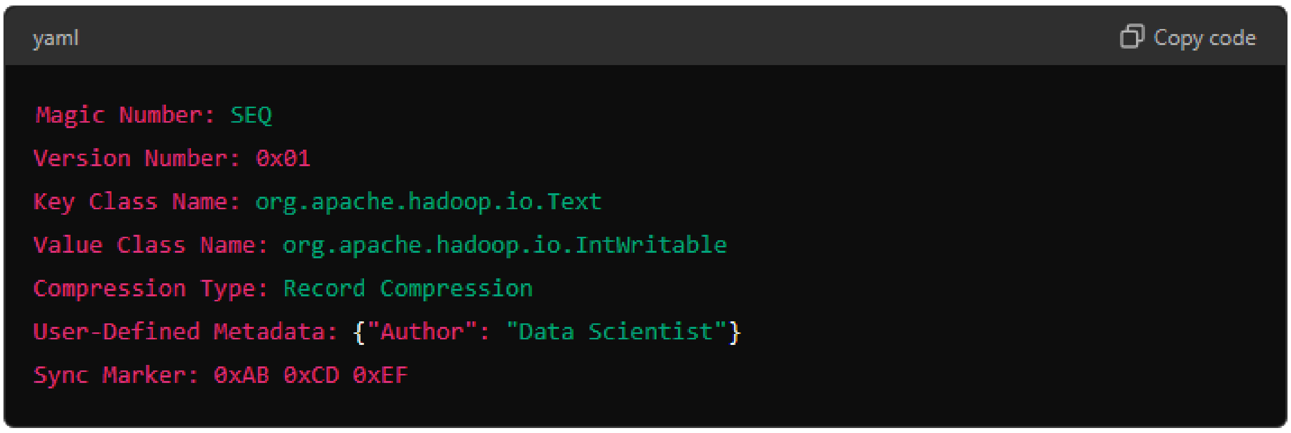
The SequenceFile format is a binary file format designed to store sequences of key-value pairs efficiently. It is optimized for high performance in Hadoop's MapReduce framework. The file format includes a header and records, and it supports various compression options.

Structure of a SequenceFile

1. **Header**:
    - **Magic Number**: The first three bytes are SEQ, which serve as a magic number to identify the file format.
    - **Version Number**: A single byte following the magic number indicates the version of the SequenceFile format.
    - **Additional Fields**:
        - **Key and Value Class Names**: Names of the classes for keys and values.
        - **Compression Details**: Information about any compression used (none, record, or block compression).

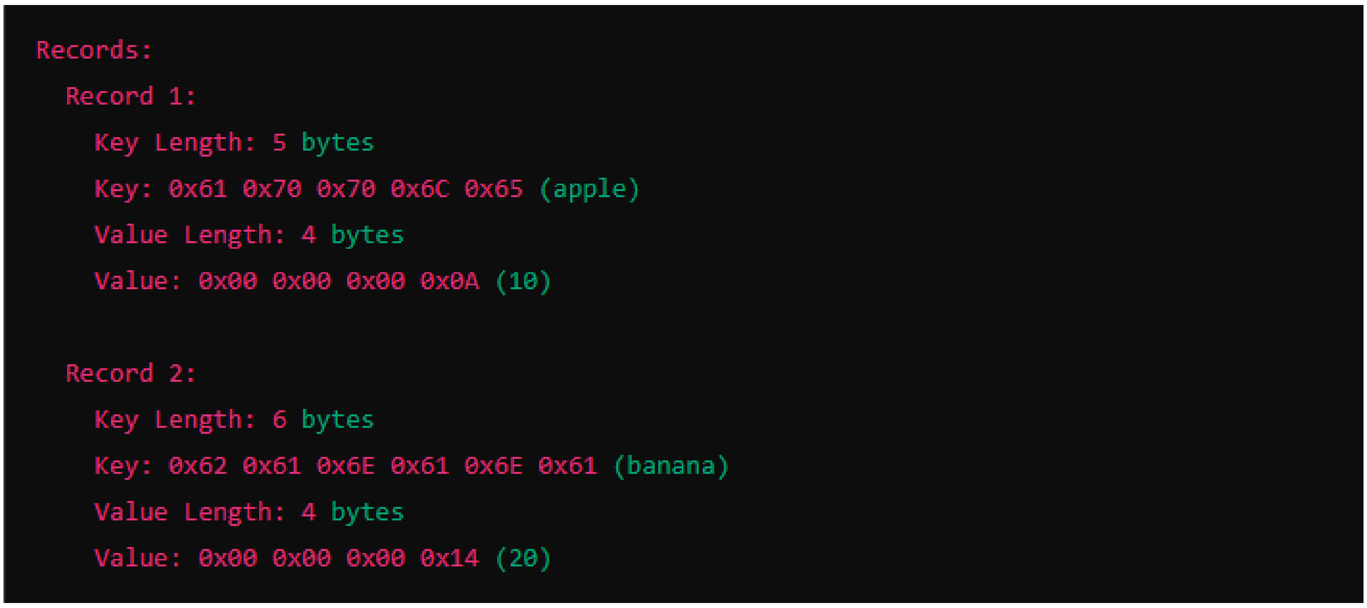
        - **User-Defined Metadata**: Additional metadata specified by the user.
        - **Sync Marker**: Used for synchronizing to a record boundary from any position in the file. Sync markers appear between records but not necessarily between every pair.

Header Example (Conceptual):



```yaml
Magic Number: SEQ
Version Number: 0x01
Key Class Name: org.apache.hadoop.io.Text
Value Class Name: org.apache.hadoop.io.IntWritable
Compression Type: Record Compression
User-Defined Metadata: {"Author": "Data Scientist"}
Sync Marker: 0xAB 0xCD 0xEF
```

2. **Records**:
    - **No Compression**:
        - **Record Length**: The total length of the record in bytes.
        - **Key Length**: The length of the key in bytes.
        - **Key**: The key itself.
        - **Value**: The value itself.
    - **Record Compression**:
        - Similar to no compression, but the value bytes are compressed using a codec defined in the header. Keys are not compressed.
    - **Block Compression**:
        - **Block Format**: Records are grouped into blocks, allowing for more efficient compression.
        - **Block Header**: Includes the number of records in the block.
        - **Compressed Fields**:
            - **Key Lengths**: Compressed lengths of all keys in the block.
            - **Keys**: Compressed key data.
            - **Value Lengths**: Compressed lengths of all values in the block.
            - **Values**: Compressed value data.
        - **Sync Marker**: A sync marker is written before the start of each block.

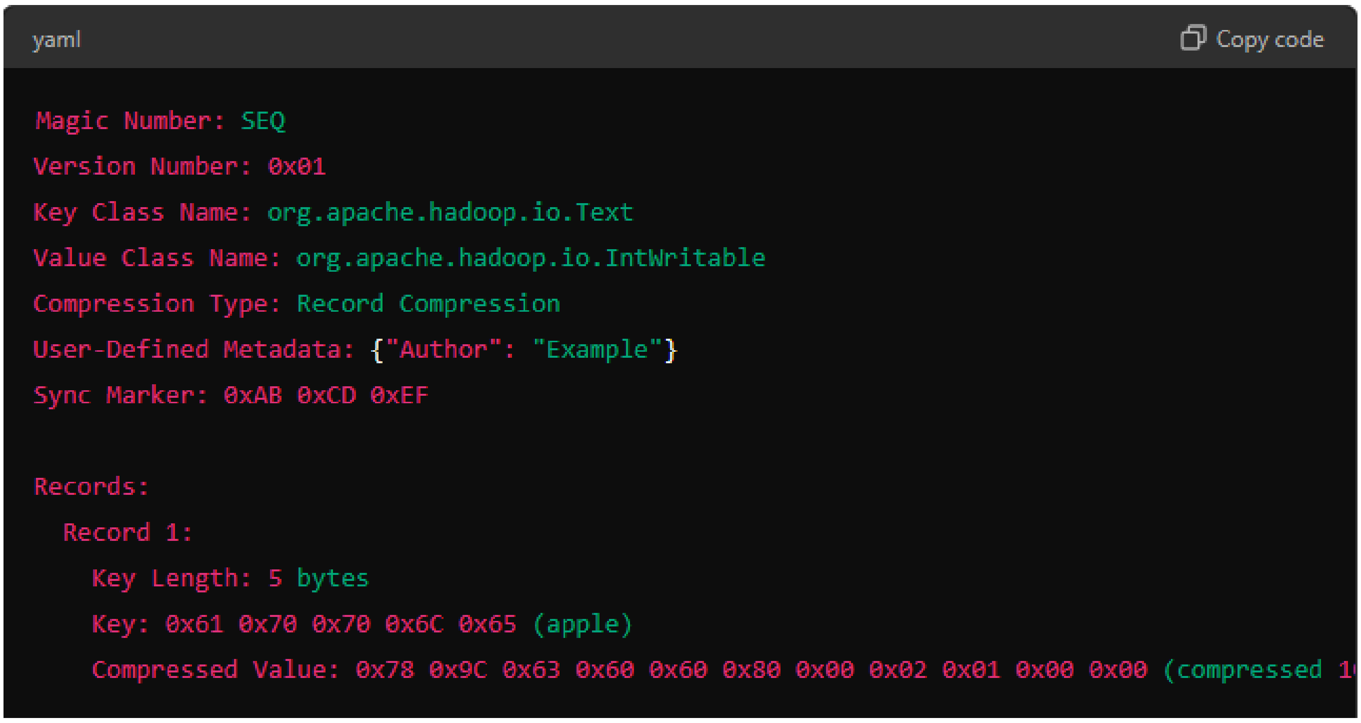


```
Records:
  Record 1:
    Key Length: 5 bytes
    Key: 0x61 0x70 0x70 0x6C 0x65 (apple)
    Value Length: 4 bytes
    Value: 0x00 0x00 0x00 0x0A (10)

  Record 2:
    Key Length: 6 bytes
    Key: 0x62 0x61 0x6E 0x61 0x6E 0x61 (banana)
    Value Length: 4 bytes
    Value: 0x00 0x00 0x00 0x14 (20)
```

Compression Details

1. **No Compression**:
   - Records are stored as plain binary data without compression.
2. **Record Compression**:
   - Compresses only the value bytes within each record.
   - Key bytes remain uncompressed.



```yaml
Magic Number: SEQ
Version Number: 0x01
Key Class Name: org.apache.hadoop.io.Text
Value Class Name: org.apache.hadoop.io.IntWritable
Compression Type: Record Compression
User-Defined Metadata: {"Author": "Example"}
Sync Marker: 0xAB 0xCD 0xEF

Records:
  Record 1:
    Key Length: 5 bytes
    Key: 0x61 0x70 0x70 0x6C 0x65 (apple)
    Compressed Value: 0x78 0x9C 0x63 0x60 0x60 0x80 0x00 0x02 0x01 0x00 0x00 (compressed 1
```

3. **Block Compression**:

- Compresses multiple records together, offering better compression efficiency by exploiting similarities between records.
- Blocks are filled up to a minimum size (default is 1 million bytes) as defined by the io.seqfile.compress.blocksize property.
- A sync marker is placed before each block, allowing readers to seek efficiently within the file.

```
Block:
  Block Header:
    Number of Records: 2

  Compressed Fields:
    Key Lengths: 0x05 0x06 (lengths of "apple" and "banana")
    Keys: 0x61 0x70 0x70 0x6C 0x65 0x62 0x61 0x6E 0x61 0x6E 0x61
    Value Lengths: 0x04 0x04
    Compressed Values: 0x78 0x9C 0x63 0x60 0x60 0x80 0x00 0x02 0x01 0x00 0x00 0x78 0x9C 0x
    Sync Marker: 0xAB 0xCD 0xEF
```

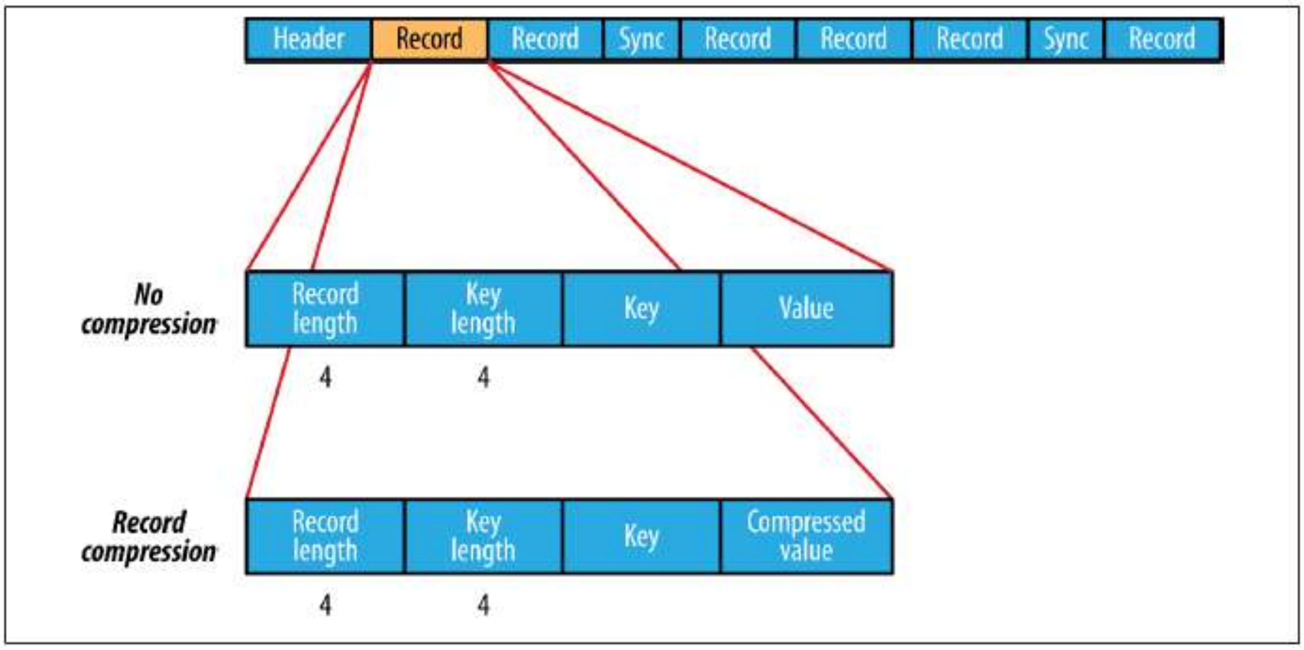


Figure 5-2. The internal structure of a sequence file with no compression and with record compression

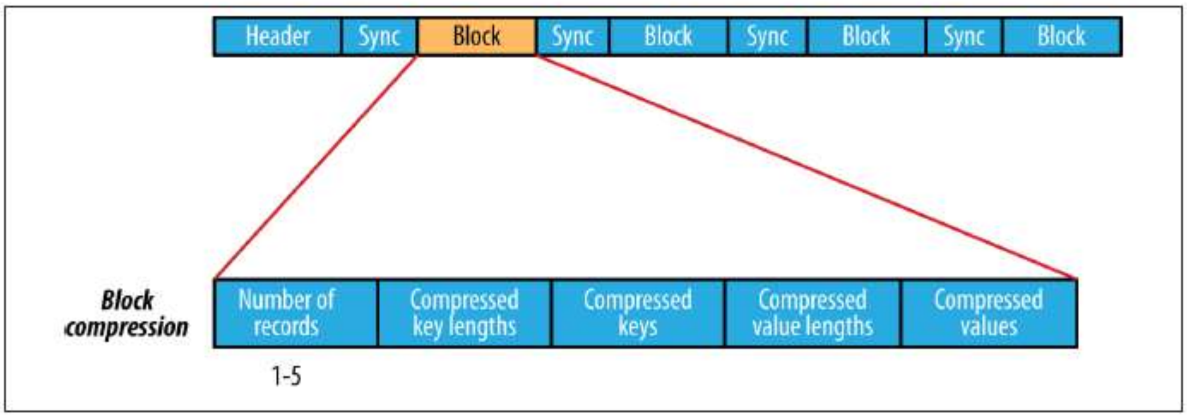


Figure 5-3. The internal structure of a sequence file with block compression

MapFile in Hadoop

A MapFile is a specialized type of SequenceFile designed for efficient key-value lookups. It combines a sorted SequenceFile with an index file to facilitate quick access to data by key.

Structure and Functionality

1. **Sorted SequenceFile**:
   - **Main Data File**: Contains all map entries sorted by key.
   - **Index File**: A secondary SequenceFile that contains a subset of keys (by default, every 128th key) and is used to speed up lookups.
2. **Index**:
   - The index file is loaded into memory to provide fast access to the data in the main file.
   - Allows quick lookups without scanning the entire data file.
3. **Writing to MapFile**:
   - When using MapFile.Writer to write data, entries must be added in sorted order.
   - Adding entries out of order will result in an IOException.
4. **Interface**:
   - The interface for reading and writing MapFiles is similar to SequenceFiles, but with additional constraints on the order of entries during writing.

Variants of MapFile

1. **SetFile**:
   - **Purpose**: Designed for storing a set of Writable keys.
   - **Requirement**: Keys must be added in sorted order.
   - **Differences**: Focuses on key storage without associated values.

2. **ArrayFile**:
    - **Purpose**: A specialized MapFile where the key represents an index in an array and the value is a Writable.
    - **Structure**:
        - **Key**: An integer index.
        - **Value**: A Writable value.
    - **Use Case**: Useful for storing arrays where the index is known and fixed.
3. **BloomMapFile**:
    - **Purpose**: Enhances the get() method for sparsely populated files using a Bloom filter.
    - **Bloom Filter**:
        - A probabilistic data structure that tests whether a key is in the map with a non-zero probability of false positives.
        - Provides fast in-memory lookups.
    - **Operation**:
        - The Bloom filter is used to quickly check if a key is likely present.
        - If the filter indicates the key may be present, the regular get() method is called to confirm.