CHENNAI
INSTITUTE OF TECHNOLOGY
(Autonomous)

UNIT III BASICS OF HADOOP

Data format – analyzing data with Hadoop – scaling out – Hadoop streaming – Hadoop pipes –design of Hadoop distributed file system (HDFS) – HDFS concepts – Java interface – data flow –Hadoop I/O – data integrity – compression – serialization – Avro – file-based data structures - Cassandra – Hadoop integration.

3.1 Data Format

In Hadoop, data can be processed in various formats depending on the nature of the data and the processing requirements. Here are some commonly used data formats in Hadoop:

1. Text/CSV Files
2. JSON Records
3. Avro Files
4. Sequence Files
5. RC Files
6. ORC Files
7. Parquet Files

Text/CSV Files

Text and CSV files are quite common and frequently Hadoop developers and data scientists received text and CSV files to work upon.

However CSV files do not support block compression, thus compressing a CSV file in Hadoop often comes at a significant read performance cost.

That is why if you are working with text or CSV files, don't include header ion the file else it will give you null value while computing the data.

Each line in these files should have a record and so there is no metadata stored in these files.

You must know how the file was written in order to make use of it.

JSON Records

JSON records contain JSON files where each line is its own JSON datum. In the case of JSON files, metadata is stored and the file is also splittable but again it also doesn't support block compression.

The only issue is there is not much support in Hadoop for JSON file but thanks to the third party tools which helps a lot. Just do the experiment and get your work done.

AVRO Files

Avro is quickly becoming the top choice for the developers due to its multiple benefits. Avro stores metadata with the data itself and allows specification of an independent schema for reading the file.

You can rename, add, delete and change the data types of fields by defining a new independent schema. Also, Avro files are splittable, support block compression and enjoy broad, relatively mature, tool support within the Hadoop ecosystem.

Sequence File

Sequence file stores data in binary format and has a similar structure to CSV file with some differences. It also doesn't store metadata and so only schema evolution option is appending new fields but it supports block compression.

Due to complexity, sequence files are mainly used in flight data as an intermediate storage.

RC File (Record Columnar Files)

RC file was the first columnar file in Hadoop and has significant compression and query performance benefits.

But it doesn't support schema evaluation and if you want to add anything to RC file you will have to rewrite the file. Also, it is a slower process.

ORC File (Optimized RC Files)

ORC is the compressed version of RC file and supports all the benefits of RC file with some enhancements like ORC files compress better than RC files, enabling faster queries.

But it doesn't support schema evolution. Some benchmarks indicate that ORC files compress to be the smallest of all file formats in Hadoop.

Parquet Files

Parquet file is another columnar file given by Hadoop founder Doug Cutting during his Trevni project. Like another Columnar file RC & ORC, Parquet also enjoys the features like compression and query performance benefits but is generally slower to write than noncolumnar file formats.

In Parquet format, new columns can be added at the end of the structure. This format was mainly optimized for Cloudera Impala but aggressively getting popularity in other ecosystems as well. One thing you should note here is, if you are working with Parquet file in Hive then you should take some precautions.

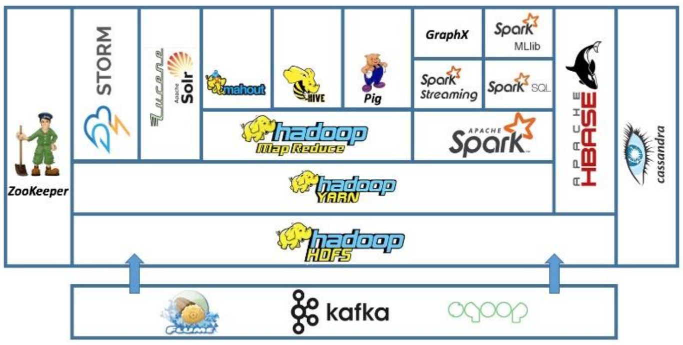
Note: *In Hive Parquet column names should be lowercase. If it is of mixed cases then hive will not read it and will give you null value.*

3.2 Analyzing data with Hadoop

Hadoop is an open-source framework that provides distributed storage and processing of large data sets. It consists of two main components: Hadoop Distributed File System (HDFS) and MapReduce. HDFS is a distributed file system that allows data to be stored across multiple machines, while MapReduce is a programming model that enables large-scale distributed data processing.

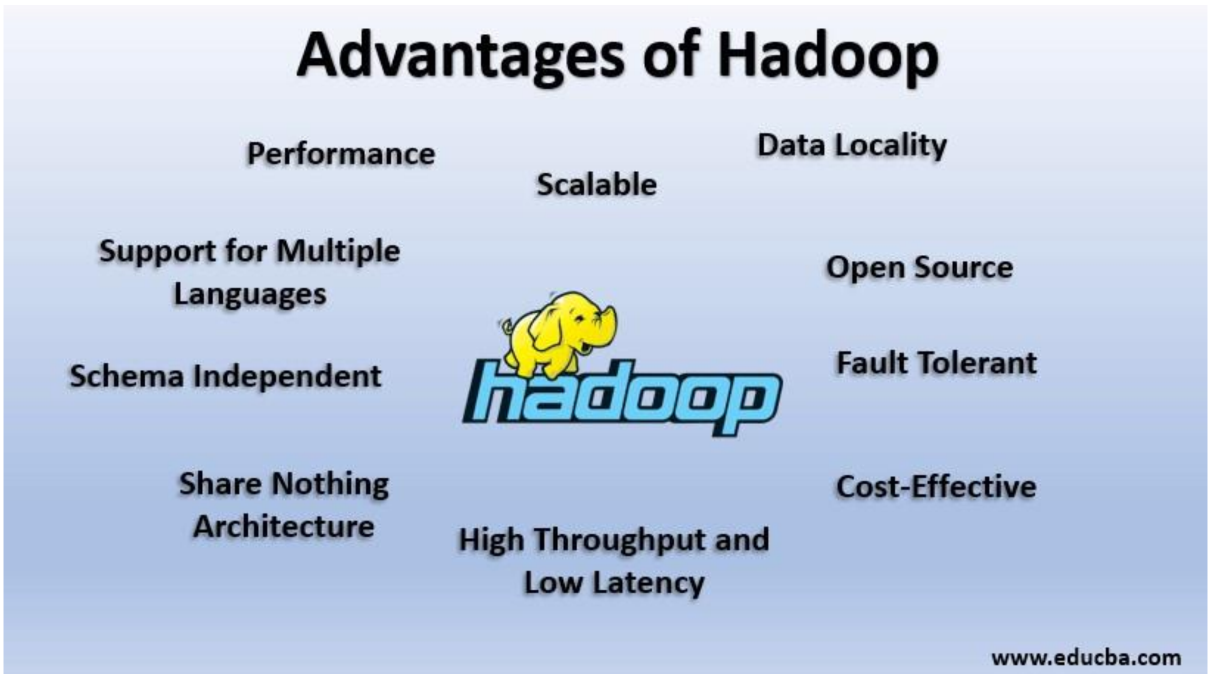
To analyze data with Hadoop, you first need to store your data in HDFS. This can be done by using the Hadoop command line interface or through a web-based graphical interface like Apache Ambari or Cloudera Manager.

Once your data is stored in HDFS, you can use MapReduce to perform distributed data processing. MapReduce breaks down the data processing into two phases: the map phase and the reduce phase.



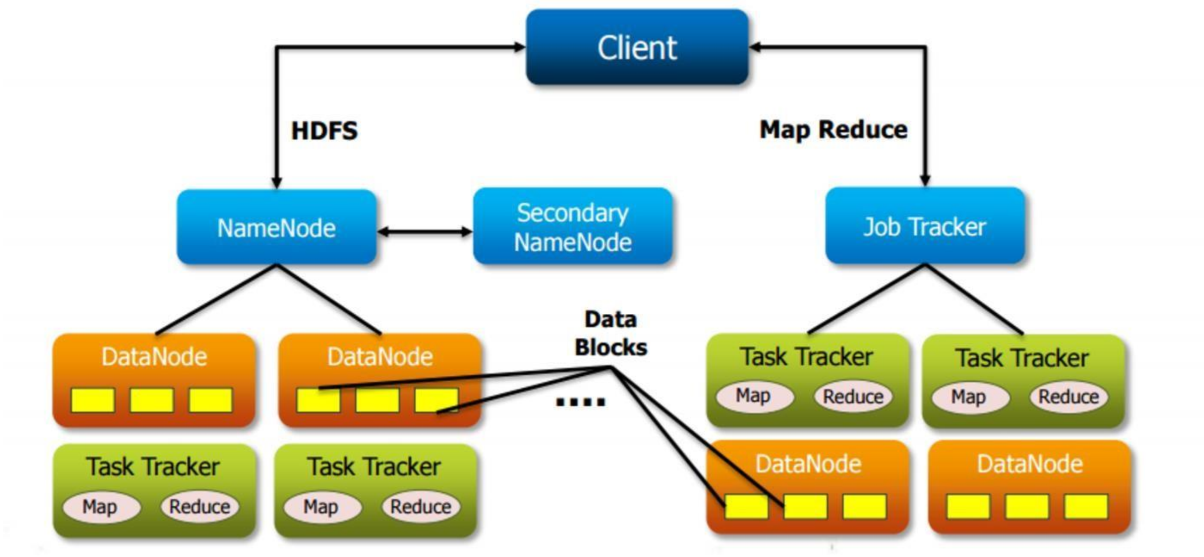
In the map phase, the input data is divided into smaller chunks and processed independently by multiple mapper nodes in parallel. The output of the map phase is a set of key-value pairs.

In the reduce phase, the key-value pairs produced by the map phase are aggregated and processed by multiple reducer nodes in parallel. The output of the reduce phase is typically a summary of the input data, such as a count or an average.



Hadoop also provides a number of other tools for analyzing data, including Apache Hive, Apache Pig, and Apache Spark. These tools provide higher-level abstractions that simplify the process of data analysis.

Apache Hive provides a SQL-like interface for querying data stored in HDFS. It translates SQL queries into MapReduce jobs, making it easier for analysts who are familiar with SQL to work with Hadoop.



Apache Pig is a high-level scripting language that enables users to write data processing pipelines that are translated into MapReduce jobs. Pig provides a simpler syntax than MapReduce, making it easier to write and maintain data processing code.

Apache Spark is a distributed computing framework that provides a fast and flexible way to process large amounts of data. It provides an API for working with data in various formats, including SQL, machine learning, and graph processing.

In summary, Hadoop provides a powerful framework for analyzing large amounts of data. By storing data in HDFS and using MapReduce or other tools like Apache Hive, Apache Pig, or Apache Spark, you can perform distributed data processing and gain insights from your data that would be difficult or impossible to obtain using traditional data analysis tools.

Here are the key steps involved in analyzing data with Hadoop:

1. Data Ingestion: The first step is to ingest the data into the Hadoop ecosystem. This can involve loading data from various sources such as files, databases, streaming platforms, or external systems. Hadoop supports different ingestion mechanisms, including batch processing and real-time streaming, depending on the nature of the data.
2. Data Storage: Hadoop offers a distributed file system called Hadoop Distributed File System (HDFS) for storing large-scale data across a cluster of nodes. Data is typically stored in HDFS in a fault-tolerant manner, ensuring replication and data availability.

   Additionally, Hadoop also supports other storage systems like Apache HBase, Apache Cassandra, and cloud-based storage solutions.
3. Data Processing: Hadoop provides the MapReduce programming model for distributed data processing. In this model, data is processed in parallel across the nodes in the Hadoop cluster. MapReduce breaks down the data processing into two main phases: the Map phase, where data is transformed into key-value pairs, and the Reduce phase, where the processed data is aggregated to generate the final results.
4. MapReduce Framework: To analyze data with Hadoop, developers write MapReduce programs using languages like Java, Python, or other Hadoop-compatible languages. These programs define the logic for the Map and Reduce phases, specifying how data is processed and transformed. The MapReduce framework handles the distribution of data and tasks across the cluster, ensuring fault tolerance and scalability.
5. Querying and Analysis: Hadoop offers various tools and frameworks for querying and analyzing data. Apache Hive, for example, provides a SQL-like interface that allows you to write queries to extract insights from data stored in Hadoop. Apache Pig is another data processing tool that uses a high-level scripting language for data transformation and analysis. Additionally, frameworks like Apache Spark and Apache Flink provide advanced analytics capabilities, including machine learning, graph processing, and real-time stream processing.
6. Data Visualization and Reporting: After processing and analyzing the data, the results can be visualized and presented in a meaningful way. Hadoop integrates with visualization tools and business intelligence platforms, such as Apache Zeppelin, Tableau, or Power BI, to create interactive dashboards, reports, and visual representations of the analyzed data.

3.3 Scaling out

Scalability is a system's ability to swiftly enlarge or reduce the power or size of computing, storage, or networking infrastructure. With the evolution of the requirements and resource demands of applications, scaling storage infrastructure provides a means of adapting to resource demands, optimizing costs, and improving the operations team's efficiency.

Scaling up (vertical scaling) and scaling out (horizontal scaling) are key methods organizations use to add capacity to their infrastructure. To an end user, these two concepts may seem to perform the same function. However, they each handle specific needs and solve specific capacity issues for the system's infrastructure in different ways.

Scaling up

Scaling up storage infrastructure aims to add resources supporting an application to improve or maintain ample performance. Both virtual and hardware resources can be scaled up.

In the context of hardware, it may be as straightforward as using a larger hard drive to greatly increase storage capacity. Note, though, that scaling up does not necessarily require changes to your system architecture.

Scaling up infrastructure is viable until individual components are impossible to scale anymore — making this a rather short-term solution.

When to scale up infrastructure

- **When there's a performance impact:** A good indicator of when to scale up is when your workloads start reaching performance limits, resulting in increased latency and performance bottlenecks caused by I/O and CPU capacity.
- **When storage optimization doesn't work:** Whenever the effectiveness of optimization solutions for performance and capacity diminishes, it may be time to scale up.
- **When your application struggles to handle the complexities of a distributed system:** You should consider scaling up when your application either struggles to distribute processes across multiple servers or to handle the complexities of a distributed system.

Example situations of when to scale up

A ERP system that manages various business processes

Consider a scenario where a large manufacturing company uses an enterprise resource planning (ERP) system to manage a variety of business processes. The ERP system should be capable of handling high I/O operations due to the large volumes of data being processed every day, including inventory, orders, payroll and more.

As the company grows and the amount of data increases, system performance may reduce, leading to inefficient operations. To counter these performance issues, the company can choose to scale up by adding more RAM, CPU, and storage resources to its existing server. This would increase the server's capacity to handle the additional data and operations, resulting in improved system performance.

Strengths

- **Relative speed:** Replacing a resource such as a single processor with a dual processor means that the throughput of the CPU is doubled. The same can be done to resources such as dynamic random access memory (DRAM) to improve dynamic memory performance.
- **Simplicity:** Increasing the size of an existing system means that network connectivity and software configuration do not change. As a result, the time and effort saved ensure the scaling process is much more straightforward compared to scaling out architecture.
- **Cost-effectiveness:** A scale-up approach is cheaper compared to scaling out, as networking hardware and licensing cost much less. Additionally, operational costs such as cooling are lower with scale-up architectures.
- **Limited energy consumption:** As less physical equipment is needed in comparison to scaling out, the overall energy consumption related to scaling up is significantly lessened.

Weaknesses

- **Latency:** Introducing higher capacity machines may not guarantee that a workload runs faster. Latency may be introduced in scale-up architecture for a use case such as video processing, which in turn may lead to worse performance.

- **Labor and risks:** Upgrading the system can be cumbersome, as you may, for instance, have to copy data to a new server. Switchover to a new server may result in downtime and poses a risk of data loss during the process.
- **Aging hardware:** The constraints of aging equipment lead to diminishing effectiveness and efficiency with time. Backup and recovery times are examples of functionality that is negatively impacted by diminishing performance and capacity.

Scaling out

Scale-out infrastructure replaces hardware to scale functionality, performance, and capacity. Scaling out addresses some of the limitations of scale-up infrastructure, as it is generally more efficient and effective. Furthermore, scaling out using the cloud ensures you do not have to buy new hardware whenever you want to upgrade your system.

While scaling out allows you to replicate resources or services, one of its key differentiators is fluid resource scaling. This allows you to respond to varying demand quickly and effectively.

When to scale out infrastructure

- **When you need a long-term scaling strategy:** The incremental nature of scaling out allows you to scale your infrastructure for expected long-term data growth. Components can be added or removed depending on your goals.
- **When upgrades need to be flexible:** Scaling out avoids the limitations of depreciating technology, as well as vendor lock-in for specific hardware technologies.

- **When storage workloads need to be distributed:** Scaling out is perfect for use cases that require workloads to be distributed across several storage nodes.

Example situations of when to scale out

Streaming services ensuring they provide a smooth user experience

A company like Netflix or YouTube, which provides streaming services to millions of users worldwide, faces unique challenges. With a growing global user base, it's not practical to rely on a single server or a cluster in one location.

In such a scenario, the company would scale out, adding servers in various global regions. This strategy would improve content delivery, reduce latency, and provide a consistent, smooth user experience. This is often executed in conjunction with content delivery networks (CDNs) that help distribute the content across various regions.

Strengths

- **Embraces newer server technologies:** Because the architecture is not constrained by older hardware, scale-out infrastructure is not affected by capacity and performance issues as much as scale-up infrastructure.
- **Adaptability to demand changes:** Scale-out architecture makes it easier to adapt to changes in demand, since services and hardware can be removed or added to satisfy demand needs. This also makes it easy to carry out resource scaling.

- **Cost management:** Scaling out follows an incremental model, which makes costs more predictable. Furthermore, such a model allows you to pay for the resources required as you need them.

Weaknesses

- **Limited rack space:** Scale-out infrastructure poses the risk of running out of rack space. Theoretically, rack space can get to a point where it cannot support increasing demand, showing that scaling out is not always the approach to handle greater demand.
- **Increased operational costs:** The introduction of more server resources introduces additional costs, such as licensing, cooling, and power.
- **Higher upfront costs:** Setting up a scale-out system requires a sizable investment, as you're not just upgrading existing infrastructure.

Scale up or scale out? How to decide

So, should you scale up or scale out your infrastructure? The decision tree below will help you more clearly answer this question.

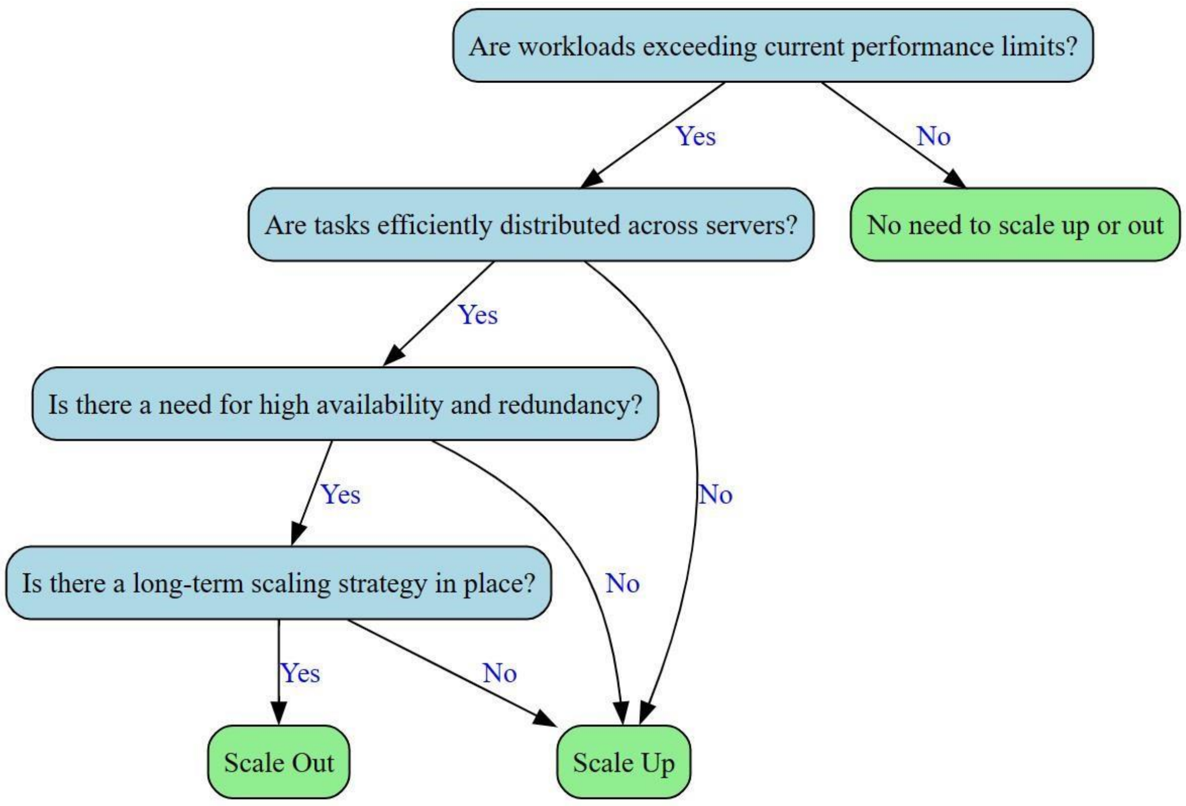
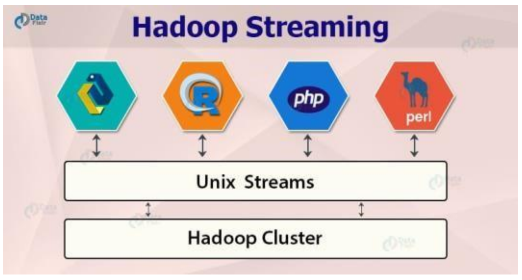

Are workloads exceeding current performance limits?
Yes
No
Are tasks efficiently distributed across servers?
No need to scale up or out
Yes
Is there a need for high availability and redundancy?
No
Yes
Is there a long-term scaling strategy in place?
No
Yes
No
Scale Out
Scale Up

Figure A. Decision tree directing whether to scale up or scale out.

Scaling out, in the context of Hadoop, refers to the ability to expand the computational and storage resources of a Hadoop cluster to accommodate larger workloads and handle increasing amounts of data. Scaling out is essential for achieving high performance, fault tolerance, and efficient processing in distributed computing environments.

3.4 Hadoop streaming

Hadoop streaming is the utility that enables us to create or run MapReduce scripts in any language either, java or non-java, as mapper/reducer.



By default, the Hadoop MapReduce framework is written in Java and provides support for writing map/reduce programs in Java only. But Hadoop provides API for writing MapReduce programs in languages other than Java.

Hadoop Streaming is the utility that allows us to create and run MapReduce jobs with any script or executable as the mapper or the reducer. It uses Unix streams as the interface between the

Hadoop and our MapReduce program so that we can use any language which can read standard input and write to standard output to write for writing our MapReduce program.

Hadoop Streaming supports the execution of Java, as well as non-Java, programmed MapReduce jobs execution over the **Hadoop cluster**. It supports the Python, Perl, R, PHP, and C++ programming languages.

Syntax for Hadoop Streaming

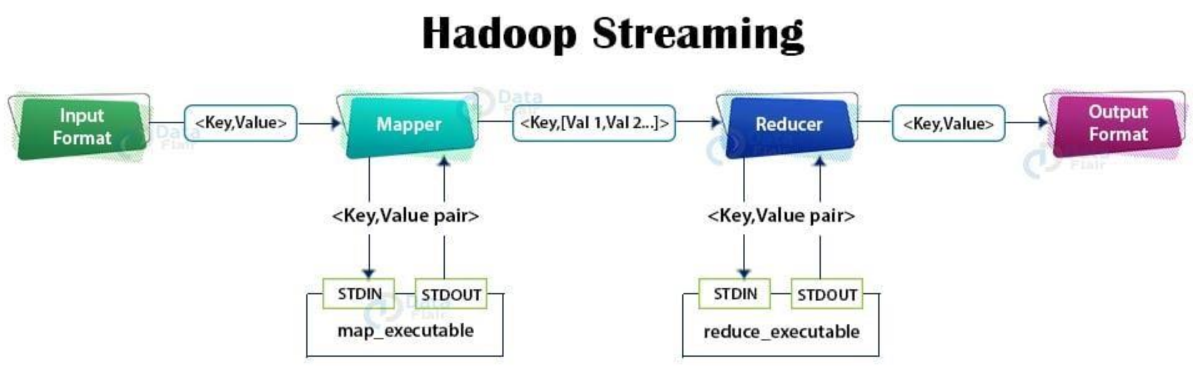
You can use the below syntax to run MapReduce code written in a language other than JAVA to process data using the Hadoop MapReduce framework.

```
$HADOOP_HOME/bin/hadoop jar $HADOOP_HOME/hadoop-streaming.jar

-input myInputDirs \

-output myOutputDir \

-mapper /bin/cat \

-reducer /usr/bin/wc
```

Parameters Description

Parameter	Description
-input myInputDirs \	Input location for mapper
-output myOutputDir \	Output location for reducer
-mapper /bin/cat \	Mapper executable
-reducer /usr/bin/wc	Reducer executable

How Streaming Works



Let us now see how Hadoop Streaming works.

- The mapper and the reducer (in the above example) are the scripts that read the input line-by-line from stdin and emit the output to stdout.
- The utility creates a Map/Reduce job and submits the job to an appropriate cluster and monitor the job progress until its completion.
- When a script is specified for mappers, then each mapper task launches the script as a separate process when the mapper is initialized.
- The mapper task converts its inputs (key, value pairs) into lines and pushes the lines to the standard input of the process. Meanwhile, the mapper collects the line oriented outputs from the standard output and converts each line into a (key, value pair) pair, which is collected as the result of the mapper.
- When **reducer** script is specified, then each reducer task launches the script as a separate process, and then the reducer is initialized.
- As reducer task runs, it converts its input key/values pairs into lines and feeds the lines to the standard input of the process. Meantime, the reducer gathers the line-oriented outputs from the stdout of the process and converts each line collected into a key/value pair, which is then collected as the result of the reducer.
- For both mapper and reducer, the prefix of a line until the first tab character is the key, and the rest of the line is the value except the tab character. In the case of no tab character in the line, the entire line is considered as key, and the value is considered null. This is customizable by setting -inputformat command option for mapper and outputformat option for reducer that we will see later in this article.

Let us now study some of the streaming command options.

Streaming Command Options

Hadoop Streaming supports some streaming command options. It also supports generic command option which we will see later in this article.

The general command line syntax is:

```
mapred streaming [genericOptions] [streamingOptions]
```

The Streaming command options are:

1. -input directoryname or filename (Required)

It specifies the input location for mapper.

2. -output directoryname (Required)

This streaming command option specifies the output location for reducer.

3. -mapper executable or JavaClassName (Optional)

It specifies the Mapper executable. If it is not specified then IdentityMapper is used as the default.

4. -reducer executable or JavaClassName (Optional)

It specifies the Reducer executable. If it is not specified then IdentityReducer is used as the default.

5. -inputformat JavaClassName (Optional)

The class you supply should return key/value pairs of Text class. If not specified then TextInputFormat is used as the default.

6. -outputformat JavaClassName (Optional)

The class you supply should take key/value pairs of Text class. If not specified then TextOutputformat is used as the default.

7. -numReduceTasks (Optional)

It specifies the number of reducers.

8. -file filename (Optional)

It makes the mapper, reducer, or combiner executable available locally on the compute nodes.

9. -mapdebug (Optional)

It is the script which is called when map task fails.

10. -reducedebug (Optional)

It is the script to call when reduce task fails.

11. -partitioner JavaClassName (Optional)

This option specifies the class that determines which reducer a key is sent to.

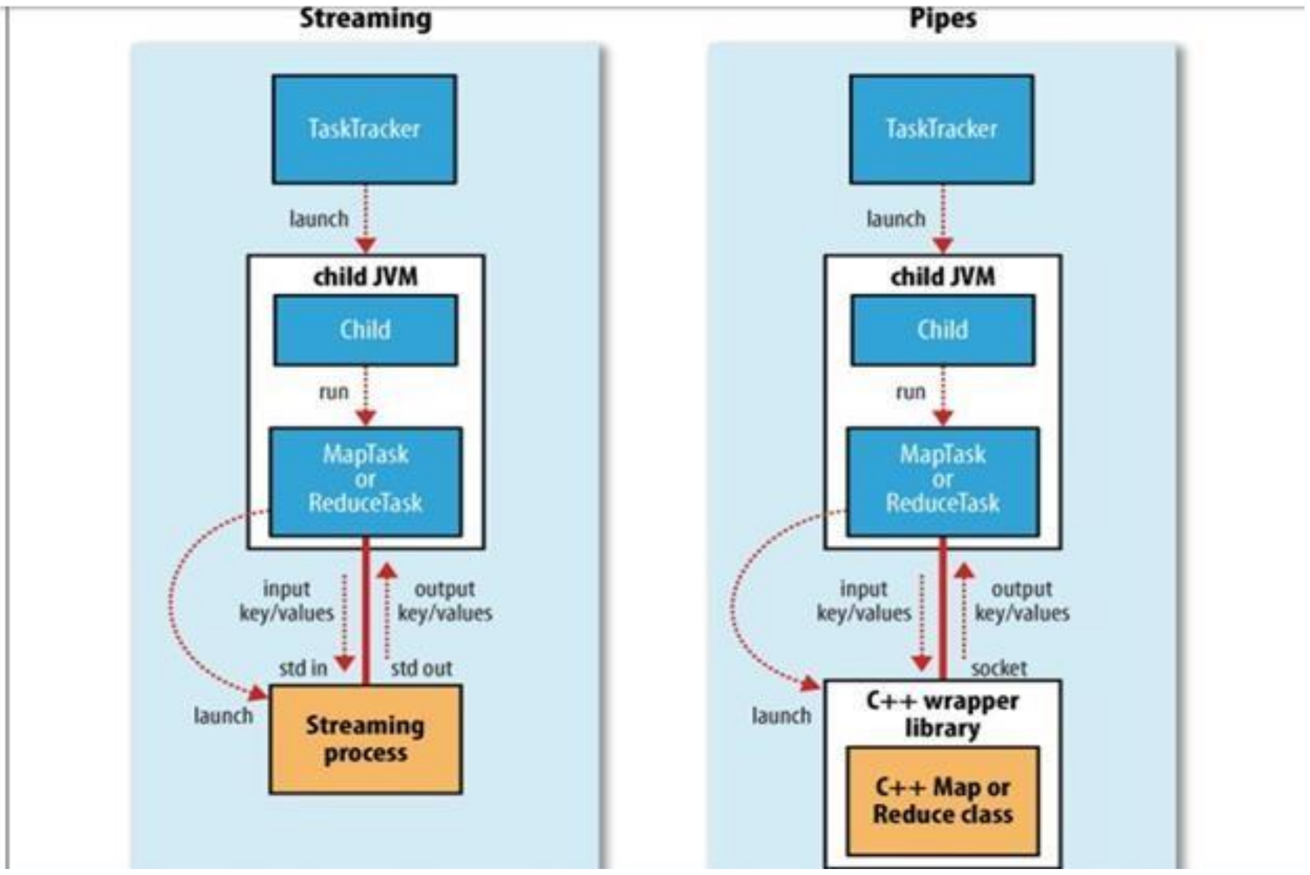
Thus these are some Hadoop streaming command options.

3.5 Hadoop pipes

Hadoop Pipes is the name of the C++ interface to Hadoop MapReduce.

Unlike Streaming, this uses standard input and output to communicate with the map and reduce code.

Pipes uses sockets as the channel over which the task tracker communicates with the process running the C++ map or reduce function.



Hadoop Pipes is another utility in the Hadoop ecosystem that allows developers to write MapReduce programs in C++ or other languages that can interface with C++. It provides a way to use non-Java languages for Hadoop MapReduce processing, similar to Hadoop Streaming. Hadoop Pipes offers a lower-level interface compared to Hadoop Streaming, allowing for direct integration with the Hadoop framework.

1. Input and Output Format: Similar to Hadoop Streaming, Hadoop Pipes expects the input and output data formats to be in the form of standard input/output streams. The input data is read from the standard input stream (stdin), and the output is written to the standard output stream (stdout).
2. Mapper and Reducer Programs: Developers write the mapper and reducer programs in C++ or a language that can interface with C++. These programs read input from stdin, process the data, and write the output to stdout. The mapper program performs the initial data transformation and emits intermediate key-value pairs. The reducer program aggregates and processes the intermediate data and produces the final output.
3. API and Library: Hadoop Pipes provides an API and a C++ library that developers can use to write MapReduce programs. The API includes classes and functions for

interacting with the Hadoop framework, accessing input data, emitting intermediate data, and writing output data.

4. Compilation and Execution: To use Hadoop Pipes, the developer compiles the mapper and reducer programs along with the Hadoop Pipes library. The resulting executable is then submitted to the Hadoop cluster for execution. The Hadoop framework handles the distribution of data and the execution of the mapper and reducer tasks across the cluster.
5. Data Streaming and Serialization: Similar to Hadoop Streaming, Hadoop Pipes enables data streaming between the mapper and reducer tasks. The data is serialized and sent as key-value pairs over the network. The streaming framework handles the distribution of data and the coordination of task execution.
6. Input Splitting and Task Execution: The input data is split into input splits, and each split is assigned to a mapper task for processing. The mapper tasks read the input data, apply the logic defined in the mapper program, and emit intermediate key-value pairs. The intermediate data is then shuffled and sorted by key before being passed to the reducer tasks.
7. Reducer Input and Output: The reducer tasks receive the shuffled and sorted intermediate data as input. They apply the logic defined in the reducer program to process and aggregate the data. The output of the reducer tasks is written to stdout, which becomes the final output of the MapReduce job.

Hadoop Pipes provides a lower-level and more efficient interface for non-Java languages to interact with the Hadoop MapReduce framework. It allows developers to leverage their existing C++ code or use other languages that can interface with C++ to process data in Hadoop.

3.6 Design of Hadoop distributed file system (HDFS)

Hadoop Distributed File System(HDFS) is the world’s most reliable storage system. It is best known for its **fault tolerance** and **high availability**.

What is Hadoop HDFS?

HDFS stores very large files running on a cluster of commodity hardware. It works on the principle of storage of less number of large files rather than the huge number of small files. HDFS stores data reliably even in the case of hardware failure. It provides high throughput by providing the data access in parallel.

HDFS Assumption and Goals

I. Hardware failure

Hardware failure is no more exception; it has become a regular term. HDFS instance consists of hundreds or thousands of server machines, each of which is storing part of the file system’s data. There exist a huge number of components that are very susceptible to hardware failure. This means that there are some components that are always non-functional. So the core architectural goal of HDFS is quick and automatic fault detection/recovery.

II. Streaming data access

HDFS applications need streaming access to their datasets. Hadoop HDFS is mainly designed for batch processing rather than interactive use by users. The force is on high throughput of data access rather than low latency of data access. It focuses on how to retrieve data at the fastest possible speed while analyzing logs.

III. Large datasets

HDFS works with large data sets. In standard practices, a file in HDFS is of size ranging from gigabytes to petabytes. The architecture of HDFS should be design in such a way that it should be best for storing and retrieving huge amounts of data. HDFS should provide high aggregate data bandwidth and should be able to scale up to hundreds of nodes on a single cluster. Also, it should be good enough to deal with tons of millions of files on a single instance.

IV. Simple coherency model

It works on a theory of **write-once-read-many** access model for files. Once the file is created, written, and closed, it should not be changed. This resolves the data coherency issues and enables high throughput of data access. A **MapReduce**-based application or web crawler application perfectly fits in this model. As per apache notes, there is a plan to support appending writes to files in the future.

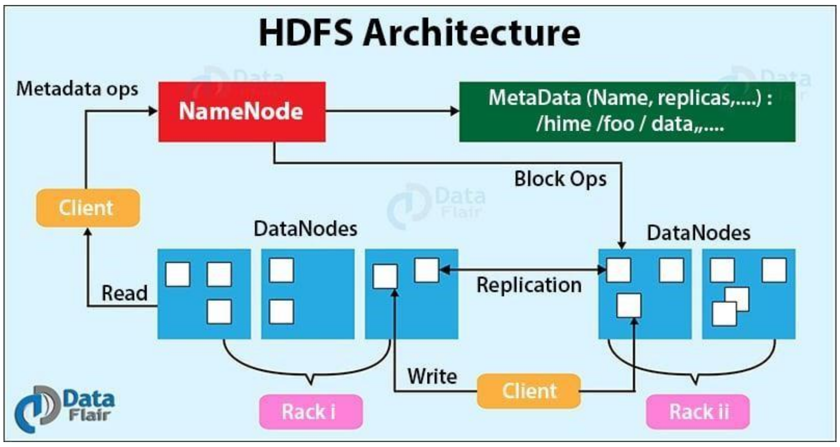
V. Moving computation is cheaper than moving data

If an application does the computation near the data it operates on, it is much more efficient than done far of. This fact becomes stronger while dealing with large data set. The main advantage of this is that it increases the overall throughput of the system. It also minimizes network congestion. The assumption is that it is better to move computation closer to data instead of moving data to computation.

VI. Portability across heterogeneous hardware and software platforms

HDFS is designed with the portable property so that it should be portable from one platform to another. This enables the widespread adoption of HDFS. It is the best platform while dealing with a large set of data.

Introduction to HDFS Architecture



Hadoop Distributed File System follows the **master-slave architecture**. Each cluster comprises a **single master node** and **multiple slave nodes**. Internally the files get divided into one or more **blocks**, and each block is stored on different slave machines depending on the **replication factor** (which you will see later in this article).

The master node stores and manages the file system namespace, that is information about blocks of files like block locations, permissions, etc. The slave nodes store data blocks of files.

The Master node is the NameNode and DataNodes are the slave nodes.

Let's discuss each of the nodes in the Hadoop HDFS Architecture in detail.

What is HDFS NameNode?

NameNode is the centerpiece of the Hadoop Distributed File System. It maintains and manages the **file system namespace** and provides the right access permission to the clients.

The NameNode stores information about blocks locations, permissions, etc. on the local disk in the form of two files:

- **Fsimage:** Fsimage stands for File System image. It contains the complete namespace of the Hadoop file system since the NameNode creation.
- **Edit log:** It contains all the recent changes performed to the file system namespace to the most recent Fsimage.

Functions of HDFS NameNode

1. It executes the file system namespace operations like opening, renaming, and closing files and directories.
2. NameNode manages and maintains the DataNodes.
3. It determines the mapping of blocks of a file to DataNodes.
4. NameNode records each change made to the file system namespace.
5. It keeps the locations of each block of a file.

6. NameNode takes care of the replication factor of all the blocks.
7. NameNode receives heartbeat and block reports from all DataNodes that ensure DataNode is alive.
8. If the DataNode fails, the NameNode chooses new DataNodes for new replicas.

Before Hadoop2, NameNode was the single point of failure. The **High Availability** Hadoop cluster architecture introduced in Hadoop 2, allows for two or more NameNodes running in the cluster in a hot standby configuration.

What is HDFS DataNode?

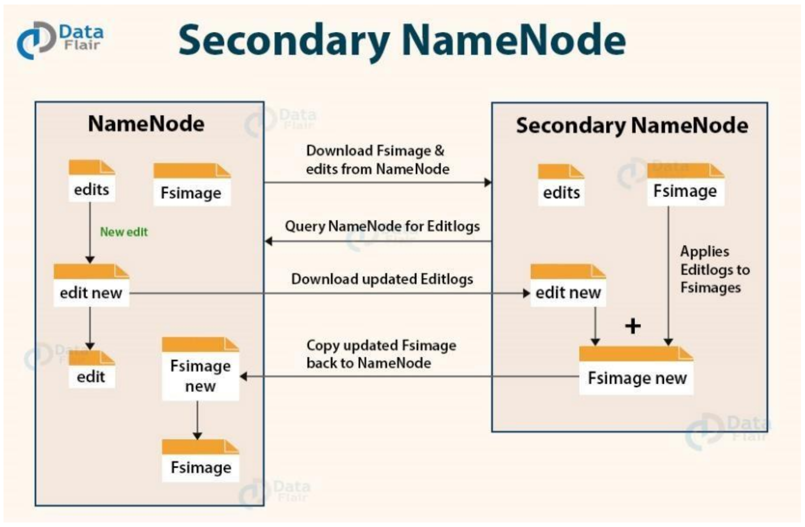
DataNodes are the slave nodes in Hadoop HDFS. DataNodes are **inexpensive commodity hardware**. They store blocks of a file.

Functions of DataNode 1. DataNode is responsible for serving the client

read/write requests.

2. Based on the instruction from the NameNode, DataNodes performs block creation, replication, and deletion.
3. DataNodes send a heartbeat to NameNode to report the health of HDFS.
4. DataNodes also sends block reports to NameNode to report the list of blocks it contains.

What is Secondary NameNode?


Secondary NameNode
NameNode
edits
Fsimage
New edit
edit new
edit
Fsimage new
Fsimage
Download Fsimage & edits from NameNode
Query NameNode for Editlogs
Download updated Editlogs
Copy updated Fsimage back to NameNode
Secondary NameNode
edits
Fsimage
Applies Editlogs to Fsimages
edit new
+
Fsimage new

Apart from DataNode and NameNode, there is another daemon called the **secondary NameNode**. Secondary NameNode works as a helper node to primary NameNode but doesn't replace primary NameNode.

When the NameNode starts, the NameNode merges the Fsimage and edit logs file to restore the current file system namespace. Since the NameNode runs continuously for a long time without any restart, the size of edit logs becomes too large. This will result in a long restart time for NameNode.

Secondary NameNode solves this issue.

Secondary NameNode downloads the Fsimage file and edit logs file from NameNode.

It periodically applies edit logs to Fsimage and refreshes the edit logs. The updated Fsimage is then sent to the NameNode so that NameNode doesn't have to re-apply the edit log records during its restart. This keeps the edit log size small and reduces the NameNode restart time.

If the NameNode fails, the last save Fsimage on the secondary NameNode can be used to recover file system metadata. The secondary NameNode performs regular checkpoints in HDFS.

What is Checkpoint Node?

The Checkpoint node is a node that periodically creates checkpoints of the namespace.

Checkpoint Node in Hadoop first downloads Fsimage and edits from the Active Namenode. Then it merges them (Fsimage and edits) locally, and at last, it uploads the new image back to the active NameNode.

It stores the latest checkpoint in a directory that has the same structure as the Namenode's directory. This permits the checkpointed image to be always available for reading by the NameNode if necessary. **What is Backup Node?**

A Backup node provides the same checkpointing functionality as the Checkpoint node.

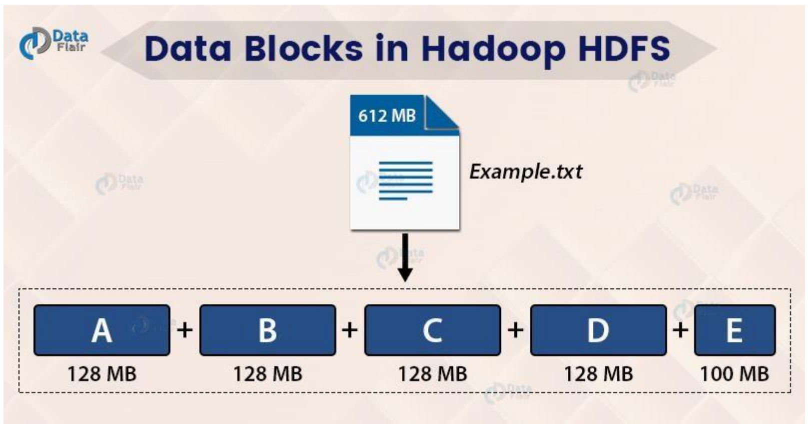
In Hadoop, Backup node keeps an **in-memory, up-to-date copy** of the file system namespace. It is always synchronized with the active NameNode state.

It is not required for the backup node in HDFS architecture to download **Fsimage** and **edits files** from the active NameNode to create a checkpoint. It already has an up-to-date state of the namespace state in memory.

The Backup node checkpoint process is more efficient as it only needs to save the namespace into the local Fsimage file and reset edits. **NameNode supports one Backup node at a time**.

This was about the different types of nodes in HDFS Architecture.

What are Blocks in HDFS Architecture?



Internally, HDFS split the file into block-sized chunks called a block. The size of the block is **128 Mb** by default. One can configure the block size as per the requirement.

For example, if there is a file of size 612 Mb, then HDFS will create four blocks of size 128 Mb and one block of size 100 Mb.

The file of a smaller size does not occupy the full block size space in the disk.

For example, the file of size 2 Mb will occupy only 2 Mb space in the disk.

The user doesn't have any control over the location of the blocks.

HDFS is highly fault-tolerant. **Now, look at what makes HDFS fault-tolerant.**

What is Replication Management?

For a distributed system, the data must be redundant to multiple places so that if one machine fails, the data is accessible from other machines.In Hadoop, HDFS stores replicas of a block on multiple DataNodes based on the replication factor.

The replication factor is the number of copies to be created for blocks of a file in HDFS architecture.If the replication factor is 3, then three copies of a block get stored on different DataNodes. So if one DataNode containing the data block fails, then the block is accessible from the other DataNode containing a replica of the block.

If we are storing a file of 128 Mb and the replication factor is 3, then (3*128=384) 384 Mb of disk space is occupied for a file as three copies of a block get stored.

This replication mechanism makes HDFS fault-tolerant.

What is Rack Awareness in HDFS Architecture?

Let us now talk about how HDFS store replicas on the DataNodes? What is a rack? What is rack awareness?

Rack is the collection of around 40-50 machines (DataNodes) connected using the same network switch. If the network goes down, the whole rack will be unavailable.

Rack Awareness is the concept of choosing the closest node based on the rack information.

To ensure that all the replicas of a block are not stored on the same rack or a single rack, NameNode follows a rack awareness algorithm to store replicas and provide latency and fault tolerance.

Suppose if the replication factor is 3, then according to the rack awareness algorithm:

- The first replica will get stored on the local rack.
- The second replica will get stored on the other DataNode in the same rack.
- The third replica will get stored on a different rack.

HDFS Read and Write Operation

1. Write Operation

When a client wants to write a file to HDFS, it communicates to the NameNode for metadata. The Namenode responds with a number of blocks, their location, replicas, and other details. Based on information from NameNode, the client directly interacts with the DataNode.

The client first sends block A to DataNode 1 along with the IP of the other two DataNodes where replicas will be stored. When Datanode 1 receives block A from the client, DataNode 1 copies the same block to DataNode 2 of the same rack. As both the DataNodes are in the same rack, so block transfer via rack switch. Now DataNode 2 copies the same block to DataNode 4 on a different rack. As both the DataNoNes are in different racks, so block transfer via an out-of-rack switch.

When DataNode receives the blocks from the client, it sends write confirmation to Namenode.

The same process is repeated for each block of the file.

2. Read Operation

To read from HDFS, the client first communicates with the NameNode for metadata. The Namenode responds with the locations of DataNodes containing blocks. After receiving the DataNodes locations, the client then directly interacts with the DataNodes.

The client starts reading data parallelly from the DataNodes based on the information received from the NameNode. The data will flow directly from the DataNode to the client.

When a client or application receives all the blocks of the file, it combines these blocks into the form of an original file.

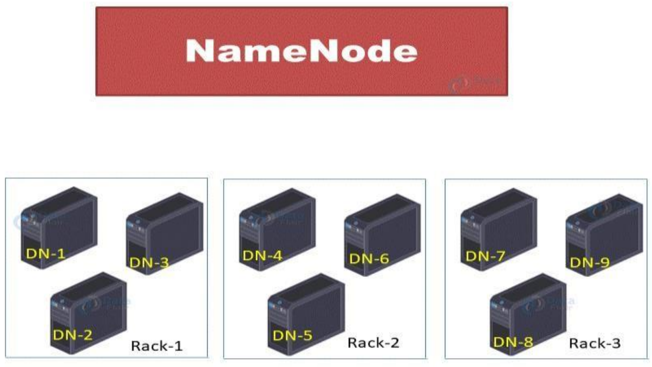
Overview Of HDFS Architecture

In Hadoop HDFS, NameNode is the master node and DataNodes are the slave nodes. The file in HDFS is stored as data blocks.

The file is divided into blocks (A, B, C in the below GIF). These blocks get stored on different DataNodes based on the Rack Awareness Algorithm. Block A on DataNode-1(DN1), block B on DataNode-6(DN-6), and block C on DataNode-7(DN-7).

To provide Fault Tolerance, replicas of blocks are created based on the replication factor.

In the below GIF, 2 replicas of each block is created (using default replication factor 3). Replicas were placed on different DataNodes, thus ensuring data availability even in the case of DataNode failure or rack failure.



Hadoop HDFS Architecture

So, This was all on HDFS Architecture Tutorial. Follow the following links to **master HDFS architecture.**

3.6 JAVA Interface

In the context of big data processing using Java, interfaces can play a significant role in defining the data flow and interactions between various components of a system. Let's explore how Java interfaces can be used to manage data flow in a big data environment.

1. InputFormat: In Apache Hadoop, InputFormat is an interface that defines how to read and parse input data into a format suitable for MapReduce processing. Implementing this interface allows you to define how your big data input will be split into chunks (input splits) and how to read and process each split.
2. OutputFormat: Similar to InputFormat, OutputFormat is an interface that defines how to write the output of a MapReduce job. Implementing this interface enables you to specify the format in which you want to store the results of your big data processing, such as writing to a file or a database.
3. Mapper and Reducer: In the MapReduce programming model, Mappers and Reducers are essential components for processing big data. They are defined as interfaces that need to be implemented by the user to specify how the data should be mapped and reduced. Mappers take input data and produce intermediate key-value pairs, while Reducers combine and aggregate the intermediate results to produce the final output.
4. Input and Output Interfaces: When dealing with big data frameworks like Apache Spark or Apache Flink, there are specific interfaces for handling input and output operations. These interfaces allow you to define data sources (e.g., files, databases,

streaming sources) and sinks (e.g., files, databases, messaging systems) for your big data processing jobs.

5. Connector Interfaces: In the big data ecosystem, there are various data processing and storage systems, such as Apache Kafka, Apache Cassandra, Apache HBase, and more. Many of these systems provide Java interfaces to interact with their APIs. By implementing these interfaces, you can establish connections and transfer data between different big data components.
6. Custom Interfaces: In addition to the standard interfaces provided by big data frameworks, you can define your custom interfaces to encapsulate specific data flow requirements or interactions between components in your big data application. These interfaces can be used to establish communication channels, define data structures, or handle specialized processing tasks.

By utilizing Java interfaces effectively, you can create modular and reusable components for big data processing. Interfaces enable loose coupling, making it easier to integrate and switch between different implementations without impacting the overall system functionality. They also help in organizing and abstracting the data flow, making the code more maintainable and extensible.

3.7 Hadoop I/O

Hadoop Input Output System Hadoop comes with a set of primitives. These primitive considerations, although generic in nature, go with the Hadoop IO system. Hadoop deals with multi-terabytes of datasets; a special consideration on these primitives will give an idea how Hadoop handles data input and output

Data Integrity

Data integrity means that data should remain accurate and consistent all across its storing, processing, and retrieval operations. To ensure that no data is lost or corrupted during persistence and processing, Hadoop maintains stringent data integrity constraints. Every read/write operation occurs in disks, more so through the network is prone to errors. And, the volume of data that Hadoop handles only aggravates the situation. The usual way to detect corrupt data is through checksums. A *checksum* is computed when data first enters into the system and is sent across the channel during the retrieval process. The retrieving end computes the checksum again and matches with the received ones. If it matches exactly then the data deemed to be error free else it contains error. But the problem is – what if the checksum sent itself is corrupt? This is highly unlikely because it is a small data, but not an undeniable possibility. Using the right kind of hardware such as ECC memory can be used to alleviate the situation.

This is mere detection. Therefore, to correct the error, another technique, called CRC (Cyclic Redundancy Check), is used.

Hadoop takes it further and creates a distinct checksum for every 512 (default) bytes of data. Because CRC-32 is 4 bytes only, the storage overhead is not an issue. All data that enters into the system is verified by the data nodes before being forwarded for storage or further processing. Data sent to the data node pipeline is verified through checksums and any corruption found is immediately notified to the client with *ChecksumException*. The client read from the data node also goes through the same drill. The data nodes maintain a log of checksum verification to keep track of the verified block. The log is updated by the data node upon receiving a block verification success signal from the client. This type of statistics helps in keeping the bad disks at bay.

Apart from this, a periodic verification on the block store is made with the help of *Data Block Scanner* running along with the data node thread in the background. This protects data from corruption in the physical storage media.

Hadoop maintains a copy or replicas of data. This is specifically used to recover data from massive corruption. Once the client detects an error while reading a block, it immediately reports to the data node about the bad block from the name node before throwing *Checksum Exception*. The name node then marks it as a bad block and schedules any further reference to the block to its replicas. In this way, the replica is maintained with other replicas and the marked bad block is removed from the system.

For every file created in the Hadoop *LocalFileSystem*, a hidden file with the same name in the same directory with the extension *.<filename>.crc* is created. This file maintains the checksum of each chunk of data (512 bytes) in the file. The maintenance of metadata helps in detecting read error before throwing *ChecksumException* by the *LocalFileSystem*.

Compression

Keeping in mind the volume of data Hadoop deals with, compression is not a luxury but a requirement. There are many obvious benefits of file compression rightly used by Hadoop. It economizes storage requirements and is a must-have capability to speed up data transmission over the network and disks. There are many tools, techniques, and algorithms commonly used by Hadoop. Many of them are quite popular and have been used in file compression over the ages. For example, gzip, bzip2, LZO, zip, and so forth are often used.

Serialization

The process that turns structured objects to stream of bytes is called *serialization*. This is specifically required for data transmission over the network or persisting raw data in disks. *Deserialization* is just the reverse process, where a stream of bytes is transformed into a structured object. This is particularly required for object implementation of the raw bytes. Therefore, it is not surprising that distributed computing uses this in a couple of distinct areas: inter-process communication and data persistence.

Hadoop uses RPC (Remote Procedure Call) to enact inter-process communication between nodes. Therefore, the RPC protocol uses the process of serialization and deserialization to render a message to the stream of bytes and vice versa and sends it across the network. However, the process must be compact enough to best use the network bandwidth, as well as fast, interoperable, and flexible to accommodate protocol updates over time.

Hadoop has its own compact and fast serialization format, *Writables*, that MapReduce programs use to generate keys and value types.

Data Structure of Files

There are a couple of high-level containers that elaborate the specialized data structure in Hadoop to hold special types of data. For example, to maintain a binary log, the *SequenceFile* container provides the data structure to persist binary key-value pairs. We then can use the key, such as a timestamp represented by *LongWritable* and value by *Writable*, which refers to logged quantity.

There is another container, a sorted derivation of *SequenceFile*, called *MapFile*. It provides an index for convenient lookups by key.

These two containers are interoperable and can be converted to and from each other.

Avro

Avro is a popular serialization framework in the Hadoop ecosystem that provides a compact and efficient data format along with schema evolution support. Avro is widely used in various Hadoop components and tools, including Hadoop MapReduce, Apache Spark, Apache Hive, and Apache Kafka.

1. Data Serialization: Avro allows you to define data schemas using the Avro schema definition language (usually in JSON format). The schema defines the structure of the data, including field names, types, and optional metadata. Avro schemas can represent complex data structures and support nested records, arrays, maps, enums, unions, and more.
2. Compact Binary Format: Avro uses a compact binary format to serialize data. The binary format is highly efficient in terms of storage and network transfer, as it uses techniques like data compression and schema resolution. Avro's compact format helps reduce the size of serialized data and improves performance during data processing and transmission.
3. Schema Evolution: Avro supports schema evolution, allowing you to evolve schemas over time while maintaining backward and forward compatibility. This means that you can modify schemas by adding, removing, or modifying fields without breaking compatibility with existing data. Avro handles schema evolution through schema resolution rules, enabling seamless data migration and system updates.
4. Interoperability: Avro provides support for multiple programming languages, making it highly interoperable. Avro schemas can be defined and shared between different programming languages, allowing seamless integration between components written in different languages. Avro's language independence enables data exchange and processing across the Hadoop ecosystem, regardless of the programming language used.
5. Integration with Hadoop Ecosystem: Avro is tightly integrated with various Hadoop tools and components. It is commonly used as a serialization format for data storage and retrieval in Hadoop Distributed File System (HDFS). Avro is also used in Hadoop MapReduce jobs as the default serialization framework. Additionally, Avro is often used

in conjunction with tools like Apache Spark, Apache Hive, and Apache Kafka for data processing and streaming applications.

6. Avro Data Files: Avro supports the concept of Avro data files, which are container files that can store multiple Avro-encoded data records along with their corresponding schemas. Avro data files provide efficient storage and retrieval of Avro-encoded data,

   and they support schema evolution within the same file. Avro data files can be used for batch processing, data exchange, and long-term data storage in Hadoop.

Overall, Avro provides a flexible, efficient, and schema-aware serialization framework for handling data in the Hadoop ecosystem. Its compact binary format, schema evolution support, and interoperability make it a popular choice for various data processing and storage use cases within the Hadoop ecosystem.

3.8 Cassandra – Hadoop integration.

Why Integrate Cassandra with Hadoop?

Integrating Cassandra with Hadoop provides several benefits, including:

1. **Scalability**: By integrating Cassandra with Hadoop, you can scale your data processing and storage capabilities to handle even larger data sets. Cassandra’s distributed architecture allows it to store and process data across multiple nodes, while Hadoop’s distributed computing framework enables parallel processing of large data sets.
2. **High Availability**: Cassandra’s distributed architecture also provides high availability, ensuring that your data remains accessible even in the event of a single node failure. This is especially important for organizations that require continuous availability of their data.
3. **Efficient Data Processing**: Hadoop’s MapReduce framework allows for efficient processing of large data sets. By integrating Cassandra with Hadoop, you can leverage Hadoop’s processing capabilities to analyze and process data stored in Cassandra.

How to Integrate Cassandra with Hadoop

Integrating Cassandra with Hadoop involves several steps, including:

Step 1: Install Hadoop and Cassandra

To integrate Cassandra with Hadoop, you need to have both technologies installed on your system. You can download the latest versions of Hadoop and Cassandra from their respective websites. Once downloaded, follow the installation instructions provided by each technology.

Step 2: Configure Hadoop

Once you have installed Hadoop, you need to configure it to work with Cassandra. This involves modifying the Hadoop configuration files to include the necessary settings for connecting to Cassandra.

First, navigate to the Hadoop installation directory and open the `core-site.xml` file. Add the following lines to the file:

```
<property>

<name>cassandra.input.thrift.address</na
me>
  <value>localhost</value>
</property>
```

Next, open the `hdfs-site.xml` file and add the following lines:

```
<property>

<name>dfs.client.use.datanode.hostname</
name>
  <value>true</value>
</property>
<property>

<name>cassandra.output.thrift.address</n
ame>
  <value>localhost</value> </property>
```

These settings configure Hadoop to use the Cassandra input and output formats.

Step 3: Configure Cassandra

Next, you need to configure Cassandra to work with Hadoop. This involves modifying the Cassandra configuration files to include the necessary settings for connecting to Hadoop.

First, navigate to the Cassandra installation directory and open the `cassandra.yaml` file. Add the following lines to the file:

```
hadoop_config:   fs.default.name:
hdfs://localhost:9000
```

This setting configures Cassandra to use the Hadoop file system.

Step 4: Create a Hadoop Job to Access Cassandra Data

Once you have configured Hadoop and Cassandra to work together, you can create a Hadoop job to access the data stored in Cassandra. This involves writing a MapReduce program that uses the Cassandra input and output formats to read and write data.

Here’s an example MapReduce program that reads data from a Cassandra table and writes it to a Hadoop file:

```
public class CassandraHadoopJob {
  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "Cassandra Hadoop Job");
job.setJarByClass(CassandraHadoopJob.class);
job.setInputFormatClass(CassandraInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);
job.setMapperClass(CassandraMapper.class);
job.setReducerClass(CassandraReducer.class);
job.setMapOutputKeyClass(Text.class);
job.setMapOutputValueClass(Text.class);
job.setOutputKeyClass(Text.class);     job.setOutputValueClass(Text.class);
    CassandraConfigHelper.setInputColumnFamily(job.getConfiguration(),
"keyspace", "table");
    CassandraConfigHelper.setOutputColumnFamily(job.getConfiguration(),
"keyspace", "table");
    CassandraConfigHelper.setInputInitialAddress(job.getConfiguration(),
"localhost");
    CassandraConfigHelper.setInputRpcPort(job.getConfiguration(), "9160");
    FileOutputFormat.setOutputPath(job, new Path("output"));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}
```

This program uses the `CassandraInputFormat` and `TextOutputFormat` classes to read and write data, respectively. The `CassandraMapper` and `CassandraReducer` classes define the Map and Reduce functions, respectively. The `CassandraConfigHelper` class is used to configure the input and output column families, as well as the initial address and RPC port for connecting to Cassandra.

Step 5: Run the Hadoop Job

Once you have written your MapReduce program, you can run the Hadoop job using the following command:

```
$ hadoop jar <path-to-jar-file> <main-class> <input-path> <output-path>
```

Replace `<path-to-jar-file>` with the path to your MapReduce program's JAR file, `<mainclass>` with the fully qualified name of your program's main class, `<input-path>` with the path to the input data, and `<output-path>` with the path to the output data.