8 vs

The breadth and nature of big data are defined by the eight traits collectively referred to as "the 8Vs of Big Data."

1. Volume: the amount of created and analysed data.

2. Variety: refers to the different types and arrangements of data, including structured, semi-structured, and unstructured data.

3. Velocity: defines the speed at which data are generated and analysed.

4. Veracity: This concept refers to the correctness, reliability, and ambiguity of the data.

The term "value" refers to the potential benefit or monetary value of the data.

6. Visualization: related to the need for safe, understandable, and accessible data.

The technical word "viscosity" refers to the resistance or friction that is felt while attempting to change how data is collected, saved, processed, or analysed. High viscosity might make it difficult to make changes to the data management process and limit the value that can be retrieved from the data.

8. "Virality" refers to data's potential to spread swiftly and unchecked, much like a virus does. Data sharing across networks and platforms, which may be challenging for enterprises to manage in terms of data privacy and security, can result in this.

HDFS

The HDFS (Hadoop Distributed File System) is a scalable and reliable storage system created for Hadoop's big data processing. In a master-slave architecture, one computer acts as the Namenode while a number of additional computers act as Datanodes.

The Namenode is responsible for maintaining the metadata, which includes the location of blocks for a given file, and the file system namespace. The Datanodes are in charge of maintaining the actual data blocks and answering read/write requests from clients.

The HDFS architecture is explained in more depth below:

1. Namenode: The Namenode, the master node of the HDFS cluster, is in responsible of managing file and directory operations in the file system namespace as well as maintaining the mapping of blocks to Datanodes. The Namenode also maintains the information for the file system, which includes block positions, replication factors, and access times. The meta data is linked to the following two folders:

I. Fs Image: From the creation of the name node, it contains all data pertaining to the file system namespace.

ii. Edit Logs: This section contains a record of all recent file system modifications done in connection with the most current FSImage.

It routinely requests a heartbeat and a block report from each DataNode in the cluster to verify that the DataNodes are active. If the required data is not received within 3 seconds, the other node with the identical data is activated.

2. Datanode: As the slave node of the HDFS cluster, the Datanode is responsible for storing the actual data blocks. When a client wants to read or write a file, it first speaks with the Namenode to learn the location of the data blocks before immediately interacting with the appropriate Datanode to obtain the data or save it.

3. Blocks: The blocks are retained on the Datanodes and can be stored in various chunk sizes. When a file is created, the user selects the replication factor, and the blocks are likewise duplicated for high availability and fault tolerance.

4. Replication: HDFS supports data replication to ensure that several copies of the same data block are preserved on distinct Datanodes for high availability and dependability. By keeping track of all the replicas of a block, the Namenode may direct clients to the replica that is closest for data retrieval. In the case of a Datanode failure, the Namenode may immediately divert clients to another copy of the same data block.

5. Data Integrity: HDFS provides checksums for data blocks to ensure data integrity and detect any mistakes that may have occurred during data transmission. The Namenode periodically checks the consistency of the data blocks and replaces corrupted blocks with trustworthy replicas.

In conclusion, the master-slave HDFS design provides a scalable and reliable storage system for large data processing in Hadoop, with its master-slave architecture for efficient data management and parallel processing. HDFS is a key component of the Hadoop infrastructure because it uses blocks and data replication, which ensures both data integrity and high availability.

• Secondary Name Node (Aide to Secondary Name Node)

The modified file is added to the FSImage, updated, and saved, after which the image is maintained in the name node once more.

Large dataset organisation and storage can be a challenging task to handle. HDFS Target 1. HDFS manages applications that must handle huge datasets. Each cluster of HDFS requires hundreds of servers to do this.

Detecting errors: Considering that HDFS made extensive use of generic hardware, it need to have technologies in place to quickly and effectively search for errors. Failure of the component occurs often.

3. Hardware efficiency: Processing is sped up when large datasets are involved since less network traffic is produced.

Benefits of HDFS

1. Fault tolerance: It was designed to recognise mistakes automatically and recover rapidly, ensuring reliability and continuity.

2. Speed: It can store 2GB of data every second due to its cluster design.

3. Access to more types of data

4. Portability and compatibility: Users have the choice to utilise HDFS in their own configuration because it is made to function with a variety of hardware setups and be compatible with a number of underlying OS.

5. Scalable: We can scale resources based on the size of your file system. Both vertical and lateral scaling occur.

6. Data locality: Data resides in data nodes rather than travelling to the location of the compute unit. Network congestion is decreased, and the system becomes more effective and efficient by shortening the distance between the data and the processing process.

7. Economical: Because HDFS data is virtual, the cost of keeping namespace and file system information may be drastically reduced.

8. Has a large amount of storage space for data of various sizes and types.

9. Versatile: Before being saved, data must be processed. can store almost any amount of information.

READ HDFS

The Hadoop Distributed File System (HDFS) is a scalable, fault-tolerant distributed file system designed to run on standard hardware. HDFS is used by Hadoop to store and manage a large amount of data. HDFS stores data in blocks that are duplicated several times to ensure data accessibility in the case of node failures.

The HDFS read method starts when a client asks to read a file that is stored in HDFS. This is a thorough explanation of the HDFS read process:

1. NameNode: The client communicates with the NameNode, the master node of the HDFS cluster. The NameNode keeps track of the metadata for each file in HDFS, including the size, the number of replicas, and the location of the blocks that make up the file.

2. Block addresses: The NameNode communicates the file block addresses to the client.

3. DataNode: The client next contacts the DataNode, which is in responsible of keeping the blocks of the file stored. Each block is stored in a different DataNode and has many replicates to ensure data availability in the case of a DataNode failure.

4. Read Data: The client receives the data from the desired block from the DataNode. If the requested block is not already in the local DataNode as a result of a client request, the DataNode obtains it from a replica node.

5. Merge Data: The client connects the data from each block to generate the whole file.

The very efficient and scalable HDFS read technique can read large volumes of data. The usage of blocks and replicas ensures high data availability even in the case of node failures. The HDFS read method additionally safeguards the data's security and confidentiality by allowing only authorised clients to access it.

In conclusion, the HDFS read process is an essential component of the Hadoop ecosystem and makes it possible to handle and archive data in a scalable and efficient manner. High data availability is achieved via blocks and replicas, and speedy and secure data access is provided through the NameNode and DataNode.

READ

To open the file it wants to see, the client first calls open() on the File System Object.

(which is a type of distributed file system used by HDFS).

Step 2: The Distributed File System (DFS) initiates a remote procedure call (RPC) to the name server to find the first few blocks in the file. The addresses of the data nodes that have copies of each block on them are provided by the name node for each block. The DFS sends the client an FSDataInputStream that may be used to obtain data. FSDataInputStream wraps a DFSInputStream, which manages the I/O for the data node and name node.

The client then employs the stream's read() function in step three. DFSInputStream stores the info node addresses for the first several blocks in the file and connects to the primary (closest) data node for the first block.

At step 4, the data node streams data back to the client, and the stream is regularly queried using the read() method.

Step 5: DFSInputStream selects the best data node for the subsequent block and stops communicating with it when the block is finished. Due to the client's perception that it is merely browsing an infinite stream, it is oblivious of this. Blocks are read as the client traverses the stream, and each time a new connection to a data node is made, it uses the DFSInputStream. It will also occasionally make a call to the name node to obtain the positions of the data nodes for the following set of blocks.

Step 6: The FSDataInputStream's method close() is called after the client has finished reading the file.

WRITE HDFS

The HDFS (Hadoop Distributed File System) is designed for Hadoop's big data processing and offers a scalable and reliable storage solution for storing enormous data sets. To publish a file in HDFS, do the following steps:

1. File Creation: The client must send a request to the Namenode with the file name, replication factor, and block size requested in order to create a new file. The Namenode returns a list of Datanodes that the client can use to store the data blocks.

2. Data Split: The client divides the data into blocks of equal size and sends the first block to the Datanode at the top of the list. The client then writes the next block to the second Datanode once all previous blocks have been written.

3. Data duplication: To ensure the data's high availability and reliability, HDFS uses data duplication. As a result, different Datanodes store different versions of the same data block. The user selects the duplication factor while creating a file.

4. File Closing: The client asks the Namenode to end the file after verifying that all blocks have been correctly written.

5. Metadata Update: The Namenode modifies its metadata to reflect the newly generated file as well as any related blocks and replication information. This information is used by the Namenode to control the file system namespace and to direct clients to the appropriate Datanode for data retrieval.

6. Data transfer: Data movement between the client and the Datanode is accomplished via a trustworthy, efficient protocol, such as Data Transfer Protocol. (DTP).

7. Checksums: HDFS provides checksums for data blocks to ensure data integrity and detect any mistakes that may have occurred during data transmission. Before being transferred to the Datanode, the client verifies the integrity of each block. The Datanode further calculates its own checksum for each block to verify data integrity and compares it to the checksum generated by the client.

8. Mistake Correction: In the event of a write mistake, HDFS may utilise the checksum data to quickly locate and correct the issue. The Namenode can also discover corrupted blocks and replace them with them if any good copies are available.

For storing massive volumes of data in Hadoop, the HDFS write operation provides a scalable, stable, and efficient solution. Because the use of data replication, checksums, and error correction procedures ensures high availability and data integrity, HDFS is an essential component of the Hadoop ecosystem.

WRITE

First, the client creates the file by using create() on the DistributedFileSystem.

(DFS).

Step 2: DFS makes an RPC call to the name node to create a new file in the namespace of the file system without any blocks attached to it. The name node performs a variety of checks to ensure that the file is not already present and that the client is authorised to create it. The name node generates a record of the new file if all of these checks are successful; otherwise, the client gets an IOException exception since the file cannot be created. The DFS provides an FSDataOutputStream, which the client may start using to transfer data to.

Step 3: The DFSOutputStream divides the client's inputted data into packets, which it then transfers to a private information queue. Before calling the name node to assign new blocks, the DataStreamer consumes the data queue and is in charge of choosing a list of suitable data nodes to store the replicas. The list of data nodes creates a pipeline; in this example, since the replication level is 3, we'll assume that there are three nodes in the pipeline. The packets are received from the DataStreamer by the main data node in the pipeline, where they are stored before being sent to the secondary data node.

Step 4: The second data node in the pipeline stores the packet in a similar manner, then passes it on to the third and last data node.

Step 5: The DFSOutputStream keeps a list of messages that are awaiting acknowledgement from data nodes in an internal "ack queue."

Step 6: After delivering the last packets to the data node's pipeline and waiting for acknowledgements, this process connects to the name node to determine if the file is complete or not.

REDUCE MAP

For managing big data sets on a cluster, the programming paradigm MapReduce and its associated implementation are used. Massive volumes of data stored in HDFS are managed using MapReduce, the central component of the Hadoop ecosystem (Hadoop Distributed File System).

The two main stages of the MapReduce paradigm are the Map stage and the Reduce stage.

Map Stage: Each component of the input data collection is subjected to a user-defined mapping function in the Map stage to create intermediate key-value pairs. The mapping function processes each incoming record to produce zero or more intermediate key-value pairs. The intermediate key-value pairs are grouped by key before proceeding to the Reduce stage.

Lower Stage: Each unique key in the intermediate key-value pairs generated by the Map stage is subjected to a user-defined reduction function in the Reduce stage. The reduction function processes each value associated with each unique key, producing zero or more output values.

Think about the following instance: You need to count the instances of each term in a sizable dataset of text documents. The following is a MapReduce solution that resolves this problem:

Reading each document, tokenizing it into words, and emitting a key-value pair for each word are all steps in the second stage of the mapping process. The following key-value pairs, for example, might be generated by the mapping function for a document containing the text "The swift brown fox jumps over the slow dog":

("The," "1," "quick," "1," "fox," "1," "jumps," "1," "over," "1," "lazy," "1") ("dog", 1)

Lower Stage: In the intermediate key-value pairs from the Map phase, the reduction function adds values for each unique key. With the aforementioned intermediate key-value combinations, for example, the reduction function may provide the results seen below:

The, 2 (Lazy, 1), (The), 2 (Fast, 1), (Brown, 1), (Fox), (Jumps, 1), and (Over, 1) ("dog", 1)

The example above demonstrates how MapReduce may be used to solve a simple word count problem. The MapReduce programming style may be used to handle a wide range of problems, including challenging data processing tasks like data aggregation, filtering, and transformations.

MRv1

For Hadoop's huge data processing, the MapReduce programming paradigm was initially implemented as MRv1 (MapReduce version 1). Its two main components, the JobTracker and the TaskTracker, are designed to process big data sets simultaneously over a cluster of commodity computers.

Here is a detailed examination of MRv1:

1. JobTracker: As the MRv1 architecture's master node, the JobTracker is in charge of managing the MapReduce jobs that clients deliver. It controls data dissemination, assigns tasks to certain cluster nodes, and monitors job accomplishment.

2. TaskTracker: A slave node in the MRv1 architecture, the TaskTracker is in responsibility of doing the tasks that the JobTracker assigned to it. When executing on a different system, each TaskTracker connects with the JobTracker to report the progress of its tasks.

3. Map Tasks: The Map Tasks are in charge of analysing the incoming data and generating intermediate key-value pairs. They are the initial step in the MapReduce process. Each block of data is handled by a different Map job, which executes on a per-block basis while running concurrently on the TaskTrackers.

4. Reduce Tasks: At the MapReduce process' second stage, Reduce tasks are in charge of merging the intermediate data that the Map tasks have generated. While they run concurrently on the TaskTrackers, the Reduce tasks use the intermediate data to create the final result.

5. Shuffling: The transfer of intermediary data generated by the Map tasks to the Reduce tasks is the process of shuffling, where this data is sorted and grouped by key to enable the Reduce task to aggregate the values for each key. The shuffling procedure is efficient and parallel because many Map processes send their intermediate data in parallel to multiple Reduce tasks.

6. Data Flow: The data flow of MRv1 is well optimised for large data processing, with input blocks being processed in parallel by several Map activities while intermediate data is jumbled and aggregated by many Reduce tasks. This makes it possible for MRv1 to analyse enormous data sets in a very scalable and efficient manner.

7. Scalability: MRv1 is designed to be incredibly scalable, enabling the cluster to grow as the size of the data set does. As a result, MRv1 now has a configurable big data processing solution that makes it simple to manage massive data volumes.

In conclusion, MRv1 implements the MapReduce programming paradigm for Hadoop's large data processing in a very effective and scalable way. Due to its master-slave architecture, parallel data processing, and efficient shuffling and aggregation of intermediate data, MRv1 is an essential component of the Hadoop ecosystem.

MapReduce is an associated implementation of a distributed, parallel method for handling large data sets on a cluster.

The following are the elements of the MapReduce version 1 design:

The task Tracker is the master node that manages the cluster's task execution and interacts with other task trackers.

Task Tracker: According to instructions from the Job Tracker, this cluster server performs tasks.

NameNode: This master node manages the file system domain and regulates client access to files.

Data in the Hadoop Distributed File System are stored on a node in the cluster called a "DataNode" (HDFS).

Here is a thorough description of how MapReduce works:

When a client sends a task to the Job Tracker, they must also provide the input and output locations as well as the MapReduce programme that will be utilised for the work.

A map job is sent to a job tracker for each chunk that the job tracker splits the data into.

The Task Tracker loads the data and then applies the map function to each element in the data. The map function processes each record, producing a set of intermediate key-value pairs in the process.

In the task tracker, the intermediate key-value pairs are sorted and organised by key. The intermediate key-value pairs that have been sorted are then passed on to the reducer tasks.

The task trackers get the reduce jobs from the job tracker.

After applying the reduce function to the intermediately sorted key-value combinations, the Task Tracker aggregates the values for each key to produce the final result.

After the final product is saved in the HDFS, the task is complete.

This process can be repeated for more map and reduce phases if the application requires it.

A new approach to the MapReduce programming paradigm for Hadoop's huge data processing is MRv2 MapReduce version 2 (MRv2). By addressing several of MRv1's shortcomings, such as scalability, fault tolerance, and resource management, it provides a more efficient and scalable choice for large data processing.

Here is a thorough evaluation of MRv2:

1. YARN: YARN is the new resource management tool for Hadoop that MRv2 introduced (Yet Another Resource Negotiator). Dynamic resource allocation is possible because to the scalable and flexible resource management technology known as YARN.

The NodeManager, a unique component of MRv2, is in charge of managing the resources on each cluster node. It notifies the ResourceManager about the CPU, memory, and disc use of each node.

3. ResourceManager: The master node in the MRv2 architecture, the ResourceManager is in charge of allocating resources among the cluster's many applications. Based on the needs of the various programmes, it chooses how to distribute resources after receiving information on resource utilisation from the NodeManager.

4. ApplicationMaster: New to MRv2, the ApplicationMaster is in charge of overseeing the MapReduce application and coordinating resource allocation with the ResourceManager. It communicates with the NodeManager to control how each node in the cluster completes its tasks.

5. Map and Reduce Tasks: Compared to MRv1, MRv2 Map and Reduce Tasks offer greater resource management and fault tolerance. The Map tasks process the incoming data to build intermediate key-value pairs, while the Reduce tasks put the intermediate data together to create the final output.

6. Better Shuffling: MRv2 adds a method of shuffling that is more efficient and resistant to data loss. The shuffling process is streamlined for parallel processing, and intermediate data is passed between Map and Reduce processes more consistently and successfully.

7. Scalability: MRv2 is designed to be extremely scalable, enabling the cluster to grow as the volume of the data gathering does. As a result, MRv2 now has a configurable big data processing solution that makes managing massive data sets simple.

MRv2 offers a more flexible, efficient, and scalable solution for Hadoop's large data processing, and is a significant improvement over MRv1. Being a key component of the Hadoop ecosystem, MRv2 now offers superior resource management, shuffling, and scalability in addition to YARN, NodeManager, ResourceManager, and ApplicationMaster.

With MapReduce version 2 (MRv2), also known as YARN (Yet Another Resource Negotiator), which allows more flexible resource management and supports a larger range of processing techniques besides MapReduce, the MapReduce architecture has been changed.

The components of the MRv2 architecture are as follows:

Resource Coordinator: The main node communicates with node managers and manages the cluster's resources.

The containers are managed by a node in the cluster known as the node manager, which also keeps track of how much resource each container uses and provides that data to the resource manager.

Master Application: It interacts with Node Managers to carry out and maintain track of obligations, and it bargains with the Resource Manager for resources.

Resources (such memory and CPU) are allotted in a container to carry out a task.

A detailed description of how MRv2 works is provided below:

The client notifies the resource management of a task request, including the resources required, the input and output locations, and the programme to be executed.

An application master, who is in charge of allocating resources and supervising tasks, is given a job by the resource manager.

When the input has been split into tasks, the Application Master negotiates the Resource Manager's resources.

After receiving the tasks from the Application Master, the Node Managers carry out the tasks in containers.

The Node Managers are in charge of carrying out the activities; they also monitor resource use and provide updates to the Application Master.

The application master combines the job output and saves the final product in the file system.

The job is deemed complete when all of the tasks have completed running and the final output has been stored to the file system.

Keep in mind that MRv2 supports more processing paradigms than only MapReduce and can execute a variety of applications including batch processing, interactive processing, graph processing, and streaming.

The main difference between MRv1 and MRv2 in terms of recovery is how they handle work failures.

When a process failed in MRv1, the entire process was restarted, which meant that any work done up until that point was lost. Because there was no coordination between the JobTracker and the TaskTracker to handle a task's progress, it was also difficult to recover from task failures.

On the other hand, the new component introduced by MRv2 called the ApplicationMaster is in charge of overseeing task execution and controlling the application. When a job fails, the ApplicationMaster can identify the problem and hand off control to another server so that it can continue where it left off. As a result, less effort is lost and task failure recovery is made easier.

The NodeManager, which is in charge of monitoring each node's state inside the network, is also a component of MRv2. If a node fails, the ResourceManager can provide resources to another node to complete the task, and the NodeManager can recognise the failure and notify it.

In general, MRv2's improved recovery method makes it a more reliable option for big data processing in Hadoop by increasing its resilience to task failures and reducing the amount of lost work.

+++++

Large data sets can be processed in parallel and distributed ways using the MapReduce version 1 (MRv1) and version 2 (MRv2), also known as YARN (Yet Another Resource Negotiator). Nonetheless, there are a few notable differences between MRv1 and MRv2:

Resource Management: When utilising MRv1, the Job Tracker is in charge of resource management, including delegating map and reduce tasks to Task Trackers. In MRv2, the Resource Manager is in charge of managing resources, while the Application Master is in charge of processing tasks and negotiating resources.

Scalability: The ability to support a significantly higher node count and a more flexible resource management system give MRv2 greater scalability than MRv1.

freedom: MRv2 provides more freedom because it supports a wider range of applications in addition to MapReduce, such as batch processing, interactive processing, graph processing, streaming, and other processing paradigms.

Processing Model: MRv1 is exclusively built for the MapReduce processing model, whereas MRv2 can support a wider range of processing models, including MapReduce and other processing paradigms.

Task management: In MRv1, it is the Job Tracker's responsibility to supervise the jobs and ensure their completion. In MRv2, it is the Application Master's responsibility to supervise the tasks and ensure their completion.

The major improvements that MRv2 offers over MRv1 are a more flexible resource management system, improved scalability, and more flexibility.

REDUCING MAP RECOVERY FAILURE

The MapReduce programming approach enables programmers to manage large datasets in a distributed environment. The Map and Reduce phases of a data processing operation are separated in MapReduce.

During the Map stage, the incoming data is divided into smaller chunks and processed simultaneously by a number of cluster nodes. The outcome of the Map phase is a collection of intermediate key-value pairs.

In the Reduce phase, the intermediate key-value pairs produced by the Map phase are merged, sorted, and processed to produce the final outcome.

In a distributed environment, failures are inevitable. Many factors, including as network difficulties, hardware or software faults, etc., might cause nodes to fail. To make sure the MapReduce operation is properly completed, a robust failure recovery mechanism is required.

Here are several techniques for recovering from MapReduce errors:

A failed Map or Reduce job can be retried by the MapReduce framework on a different node. A customizable number of times can pass before the job permanently succeeds or fails.

2. Speculative Execution: If a Map or Reduce job is taking longer than expected, the framework might start a duplicate task on a separate node. The job that is successfully finished first is utilised, and the alternative is dropped. The execution time of this procedure can be drastically reduced by making use of unused resources.

3. Another choice is to start the backup jobs simultaneously with the primary tasks. These tasks aren't used unless the main job fails. In the case that the primary job fails, the backup task takes over to ensure that the work is finished.

4. Checkpointing: Interim results may frequently be saved to a distributed file system, such as HDFS. The framework can pick up where it left off in order to minimise the amount of work that needs to be completed once a job fails.

5. Task Observation: The MapReduce architecture can track the status of each task and quickly spot errors. When a job fails, the framework has the ability to record the error message and notify the administrator.

These techniques enable MapReduce to handle failures in a distributed environment and ensure the successful completion of jobs involving enormous data processing.

Architecture by Apache Hive

A query language (HiveQL) similar to SQL is provided by the data warehousing tool Apache Hive for exploring and analysing big datasets stored in Hadoop. The aspects that make up Hive's design are as follows:

1. User Interface: Hive provides a range of platforms for users to interact with the system. The command-line interface (CLI), which is also used to control the system, is how HiveQL searches are executed. The Hive Web Interface provides a web-based graphical user interface (GUI) for interacting with Hive.

2. Driver: The Driver is in charge of managing the lifecycle of a HiveQL query. It converts a user interface query into an execution plan, which it then utilises the cluster to execute.

3. Compiler: The Compiler is in responsible of extracting the HiveQL query's processing plan from it. A logical plan that the compiler generates is converted into a physical plan that can be performed on the cluster.

4. Metastore: The Metastore is the primary repository for knowledge about the data stored in Hive. It maintains track of tables, divisions, columns, as well as the many types of data, their locations, and their formats.

5. Execution Engine: The Compiler's real plan must be carried out by the Execution Engine. Hive supports a wide range of processing engines, including MapReduce, Tez, and Spark.

6. SerDe: The SerDe (Serializer/Deserializer) serialises and deserializes data stored in Hive. It transforms the data between its internal representation and the format required by the storage device.

7. Storage Handler: In charge of interacting with the data storage device. Hive supports a number of storage drivers, including HDFS, HBase, and Amazon S3.

8. Hadoop Distributed File System (HDFS): The main file system for Hadoop is HDFS. For storage, it distributes the data among a number of cluster computers.

9. Apache ZooKeeper is a distributed coordination programme that is used in Hadoop to manage configuration data, synchronisation, and naming services. Hive uses it for global locking and coordination.

To provide a scalable and flexible data warehousing option that can be used to process sizable datasets stored in Hadoop, the Hive architecture was developed. The architecture allows users to use a number of execution engines and storage handlers to customise the system to their requirements.

SPARK APACHE

Apache Spark is an open-source massive data processing engine with an emphasis on speed, scalability, and usability. Spark provides a single framework for processing batch and streaming data, machine learning, graph processing, and interactive searches.

The design of Spark, which makes use of a cluster of computers to process data in parallel, is built on the distributed computing concept. The main Spark design components are as follows:

1. Driver Program: The Driver Program, which is the main programme, manages the Spark application. It maintains the cluster's distributed processing while also running the user's code.

2. Executors: Each cluster node has a running instance of a worker programme referred to as an executor. The driving software assigns them duties, and they are in charge of completing those tasks and managing the memory and resources those activities need.

3. Cluster Manager: The Cluster Manager is in responsible of managing the network traffic, RAM utilisation, and CPU use for the cluster. Spark allows other cluster managers in addition to Hadoop YARN, Apache Mesos, and its own standalone cluster manager.

4. Resilient Distributed Datasets (RDDs): The Spark data model is built on RDDs. These are immutable, partitioned data sets that the cluster is capable of managing concurrently.

5. Spark SQL: Spark SQL provides a structured data querying tool that is comparable to SQL. It enables data kept in many formats, such as Apache HBase, Hadoop Distributed File System, and Apache Cassandra, to be queried.

Spark Streaming: 6. Spark Streaming is a processing engine for streaming data that is adaptable and fault-tolerant. It provides help with ingesting and analysing real-time data streams, such as online logs, sensor data, and social media feeds.

7. MLlib: MLlib is the name of Spark's machine learning framework. It provides a set of high-level APIs for the creation of scalable machine learning algorithms and models.

8. GraphX: GraphX is the name of the network processing library used by Spark. It provides a variety of graph-building and graph-working APIs, including those for social networks, transportation networks, and biological networks.

Essentially, Spark's architecture is designed to provide a flexible and scalable framework for managing huge datasets and performing complex computations on them. Its modular architecture allows users to pick and select the components they need for their specific use cases, and its unified programming interface facilitates the creation and deployment of big data applications.

What separates HADOOP and SPARK?

Hadoop and Spark are two big data processing systems that can handle massive volumes of data. Yet, they differ greatly in terms of their use cases, performance, and architecture.

1. Architecture: Hadoop utilises a distributed file system called Hadoop Distributed File System (HDFS) for storing and managing data, whereas Spark does not and may function on top of other distributed file systems like HDFS, Amazon S3, or Apache Cassandra. Moreover, Hadoop employs a two-stage MapReduce processing model, whereas Spark uses an in-memory processing engine.

2. Performance: Spark often outperforms Hadoop, especially when dealing with large datasets. This is because Spark's in-memory processing engine allows it to run multiple calculations, in contrast to Hadoop's requirement to read and write data to disc between each step of processing.

3. Use Cases: Hadoop is more frequently used for batch processing and storing enormous volumes of data, whereas Spark is better suited for real-time stream processing, machine learning, and interactive data analysis. Additionally, Spark is usually utilised in circumstances that require speedier processing and more complex computations, whereas Hadoop is frequently used in data warehousing and ETL (Extract, Transform, Load) scenarios.

4. Ecosystem: Hadoop has a broader ecosystem than Spark since it has been around longer and has a more established community. This ecosystem's many different tools and technologies, like as Apache Pig, Apache Hive, and Apache Zookeeper, may all be utilised with Hadoop. GraphX, MLlib, and Spark SQL are already part of the community of tools and libraries that make up Spark, but this community is growing swiftly.

The architecture, performance, and use cases of Hadoop and Spark differ significantly from one another. Hadoop is more frequently used for batch processing and data storage, but Spark excels in real-time stream processing, machine learning, and interactive data analysis.

LIKENESS BETWEEN HADOOP AND SPARK

As massive data processing frameworks, Hadoop and Spark share a number of characteristics:

1. Parallel processing of large datasets over a cluster of nodes is made possible by the distributed computing capabilities of both Hadoop and Spark.

2. Open Source: Because Hadoop and Spark are both open-source initiatives, users are allowed to access and modify them as necessary.

3. By adding extra nodes to the network as needed, Hadoop and Spark both provide horizontal scaling.

4. Fault Tolerance: Both Hadoop and Spark are designed to be fault tolerant, meaning they can recover from hardware issues or other types of system faults without losing any data.

5. Batch Processing: Spark and Hadoop may also be used for batch processing, which involves managing enormous volumes of data in a predetermined or routine manner.

6. Ecosystem: Both Hadoop and Spark have robust ecosystems of tools, libraries, and technologies to carry out a range of large data processing tasks.

In conclusion, Hadoop and Spark are both excellent choices for addressing big data processing tasks in a distributed computing context due to their many architectural, functional, and ecosystem similarities.

Database for columns

A columnar database, also known as a column-oriented database or a columnar store, is a database management system (DBMS) that maintains data in columns rather than rows. In a columnar database, each column of data is kept independently and may have a unique data type.

Columnar databases are designed to maximise efficiency and performance for analytical workloads, particularly those involving complicated queries or the aggregation of huge datasets. Because each column is kept independently, columnar databases' ability to access only the exact columns needed by a given query minimises the quantity of data that must be processed and can significantly improve query performance.

Another advantage is that columnar databases may compress data more effectively than row-oriented databases. Each column contains just one type of data, making it easier to compress and store that data more efficiently. This compression can significantly lower the amount of data that has to be read from disc, which lowers storage costs and speeds up queries.

Columnar databases are commonly used in data warehousing, business intelligence, and analytics programmes where high performance and scalability are crucial. Some examples of columnar databases are Apache Cassandra, Apache HBase, Google Bigtable, Amazon Redshift, and Apache Kudu.

The tables of a columnar database are arranged in rows. Fast tabular operations like MIN, MAX, SUM, COUNT, and AVG execution as well as effective data write and read operations to and from hard disc storage are all advantages. Columnar databases allow for random read and write in Hadoop and meet the ACID criteria for a database.

ACUTE PIG

Pig analyses data in Hadoop using a language known as Pig Latin. It is a high-level language for processing data and provides a large range of operators and data types for handling data in various ways.

To carry out a given task while using Pig, programmers must write a Pig script in the Pig Latin language and run it using one of the execution methods (Grunt Shell, UDFs, Embedded). These scripts will undergo a variety of modifications after being executed by the Pig Framework in order to provide the desired output.

These scripts are internally converted by Apache Pig into a series of MapReduce jobs, which makes the programmer's job easier.

There are several components that make up the Apache Pig architecture.

Parser

The Pig Scripts are originally handled by the Parser. It conducts a variety of tests, including type checking and syntactic analysis of the script. The output of the parser will be a directed acyclic graph (DAG), which represents the logical operators and sentences in Pig Latin.

The DAG shows the logical operators of the script as nodes and the data processes as edges.

Optimizer The logical optimizer processes logical optimisations like projection and pushdown after receiving the logical plan (DAG).

Compiler The compiler transforms the optimised logical plan into a series of MapReduce tasks.

Execution engine

Prior to being delivered to Hadoop, the MapReduce tasks are sorted. The desired results are acquired when these MapReduce tasks are eventually executed on Hadoop.

CODING

HDFS

\*\*\* must be root for this

su root

\*\*\* mkdir

hdfs dfs -mkdir /myhdfs

\*\*\* copy from local to hdfs

hdfs dfs -put /home/cloudera/myfiles/datafile.txt /myhdfs/datafile.txt #(proper file path to be copied)

hdfs dfs -ls /myhdfs (list all the files)

\*\*\* delete file

hdfs dfs -rm /myhdfs/delfile.txt

\*\*\* delete directory

hdfs dfs -rmdir /myhdfs

PIG

\*\*\* Pig Commands - Interactive Mode\*\*\*

======================================

\*\*\* subscriber - count bytes exercise \*\*\*

\*\*\* =====================================

### start pig interative mode

pig

Grunt>

### quit pig interative mode

quit

### clear screen

clear

### hdfs commands

su root

hdfs dfs -mkdir /mypig

hdfs dfs -mkdir /mypig/subscriber

hdfs dfs -mkdir /mypig/subscriber/input

hdfs dfs -put /home/cloudera/myfiles/pig-subscriber.txt /mypig/subscriber/input

### pig commands for textfile

### sum bytes of Subscriber

A = load '/mypig/subscriber/input' as (line:chararray); (load file in pig)

B = foreach A generate (chararray)SUBSTRING(line,14,26) as id , (double)SUBSTRING(line,87,97) as bytes;

C = group B by id; (group by if mentioned)

D = foreach C generate group, SUM(B.bytes);

dump D; (to display)

store B into '/mypig/subscriber/output' using PigStorage(','); (to store txt file to csv file on browser)

\*\*\* customer - read csv & write csv exercise \*\*\*

\*\*\* ============================================

### hdfs commands

su root

hdfs dfs -mkdir /mypig

hdfs dfs -mkdir /mypig/customer

hdfs dfs -mkdir /mypig/customer/input

hdfs dfs -put /home/cloudera/myfiles/pig-customer.csv /mypig/customer/input

### pig commands

CustFile = load '/mypig/customer/input' using PigStorage(',') as ( CustId:int, FirstName:chararray, LastName:chararray, Phone:chararray, City:chararray );

dump CustFile;

store CustFile into '/mypig/customer/output' using PigStorage(','); (to store in csv we need to PigStorge(',')

Pig Practice

A = load '/mypractice/hr-prac.txt' as (line:chararray);

B = foreach A generate (chararray)SUBSTRING(line,0,2) as id ,(chararray)SUBSTRING(line,3,22) as name, (chararray)SUBSTRING(line,23,24) as gender,SUBSTRING(line,25,26) as dept,(double)SUBSTRING(line,27,31) as price;

dump B;

store B into '/mypractice/output' using PigStorage(',');

D = group B by id;

F = foreach D generate group, SUM(B.price);

E = FILTER F by (B.price> 7214.0);

dump E;

grunt> SPLIT F into students1 if (B.price>7214.0), students2 if (B.price<=7214.0);

grunt> H = foreach B generate (id,name),LOWER(name);

dump H;

HIVE

\*\*\* Hive Commands \*\*\*

=====================

### version

hive --version

beeline --version

### start

hive

beeline -u jdbc:hive2://

### quit

quit

!quit

### databases / tables

show databases;

use default;

show tables;

create database hr;

use hr;

create table employee (emp\_id string, emp\_name string, salary float, status int) row format delimited fields terminated by ',' lines terminated by '\n';

load data local inpath '/home/cloudera/myfiles/hive-employee.csv' overwrite into table employee; (Data loaded from local)

hive> LOAD DATA inpath 'hdfs:///myproject/output' OVERWRITE INTO TABLE insurance; #(data loaded from hdfs)

desc employee;

select \* from employee;

select \* from employee where salary >= 5000;

select \* from employee where salary = 2000;

select \* from employee where emp\_name like 'cyrus';

select \* from employee where emp\_name like 'cyrus%';

select \* from employee where emp\_name like 'Cyrus%';

select \* from employee where emp\_name like '%ta%';

select \* from employee order by emp\_name; #note - run as mr-job

select count(\*) from employee; #note - run as mr-job

select count(\*) from employee where salary = 2000; #note - run as mr-job

select salary, count(\*) from employee group by salary;

select salary, sum(salary) as sumsal from employee group by salary;

select max(salary) as maxsal, min(salary) as minsal from employee;

EXPLAIN select salary, count(\*) from employee group by salary;

SQOOP

to open sql: mysql -u root -p

# export data ... read from hdfs & store to mysql -- check map reduce job

linux>

hdfs dfs -mkdir /myhdfs/employee

hdfs dfs -put /home/cloudera/myfiles/hive-employee.csv /myhdfs/employee

mysql>

create database employee; #(if db does not exists)

use employee;

create table new\_emp (emp\_id VARCHAR(10), emp\_name VARCHAR(50), salary FLOAT, status INT);

linux>

sqoop export --connect jdbc:mysql://localhost/employee --username root -P --table new\_emp --export-dir /myhdfs/employee --input-fields-terminated-by ',' --lines-terminated-by '\n'8 vs

The breadth and nature of big data are defined by the eight traits collectively referred to as "the 8Vs of Big Data."

1. Volume: the amount of created and analysed data.

2. Variety: refers to the different types and arrangements of data, including structured, semi-structured, and unstructured data.

3. Velocity: defines the speed at which data are generated and analysed.

4. Veracity: This concept refers to the correctness, reliability, and ambiguity of the data.

The term "value" refers to the potential benefit or monetary value of the data.

6. Visualization: related to the need for safe, understandable, and accessible data.

The technical word "viscosity" refers to the resistance or friction that is felt while attempting to change how data is collected, saved, processed, or analysed. High viscosity might make it difficult to make changes to the data management process and limit the value that can be retrieved from the data.

8. "Virality" refers to data's potential to spread swiftly and unchecked, much like a virus does. Data sharing across networks and platforms, which may be challenging for enterprises to manage in terms of data privacy and security, can result in this.

HDFS

The HDFS (Hadoop Distributed File System) is a scalable and reliable storage system created for Hadoop's big data processing. In a master-slave architecture, one computer acts as the Namenode while a number of additional computers act as Datanodes.

The Namenode is responsible for maintaining the metadata, which includes the location of blocks for a given file, and the file system namespace. The Datanodes are in charge of maintaining the actual data blocks and answering read/write requests from clients.

The HDFS architecture is explained in more depth below:

1. Namenode: The Namenode, the master node of the HDFS cluster, is in responsible of managing file and directory operations in the file system namespace as well as maintaining the mapping of blocks to Datanodes. The Namenode also maintains the information for the file system, which includes block positions, replication factors, and access times. The meta data is linked to the following two folders:

I. Fs Image: From the creation of the name node, it contains all data pertaining to the file system namespace.

ii. Edit Logs: This section contains a record of all recent file system modifications done in connection with the most current FSImage.

It routinely requests a heartbeat and a block report from each DataNode in the cluster to verify that the DataNodes are active. If the required data is not received within 3 seconds, the other node with the identical data is activated.

2. Datanode: As the slave node of the HDFS cluster, the Datanode is responsible for storing the actual data blocks. When a client wants to read or write a file, it first speaks with the Namenode to learn the location of the data blocks before immediately interacting with the appropriate Datanode to obtain the data or save it.

3. Blocks: The blocks are retained on the Datanodes and can be stored in various chunk sizes. When a file is created, the user selects the replication factor, and the blocks are likewise duplicated for high availability and fault tolerance.

4. Replication: HDFS supports data replication to ensure that several copies of the same data block are preserved on distinct Datanodes for high availability and dependability. By keeping track of all the replicas of a block, the Namenode may direct clients to the replica that is closest for data retrieval. In the case of a Datanode failure, the Namenode may immediately divert clients to another copy of the same data block.

5. Data Integrity: HDFS provides checksums for data blocks to ensure data integrity and detect any mistakes that may have occurred during data transmission. The Namenode periodically checks the consistency of the data blocks and replaces corrupted blocks with trustworthy replicas.

In conclusion, the master-slave HDFS design provides a scalable and reliable storage system for large data processing in Hadoop, with its master-slave architecture for efficient data management and parallel processing. HDFS is a key component of the Hadoop infrastructure because it uses blocks and data replication, which ensures both data integrity and high availability.

• Secondary Name Node (Aide to Secondary Name Node)

The modified file is added to the FSImage, updated, and saved, after which the image is maintained in the name node once more.

Large dataset organisation and storage can be a challenging task to handle. HDFS Target 1. HDFS manages applications that must handle huge datasets. Each cluster of HDFS requires hundreds of servers to do this.

Detecting errors: Considering that HDFS made extensive use of generic hardware, it need to have technologies in place to quickly and effectively search for errors. Failure of the component occurs often.

3. Hardware efficiency: Processing is sped up when large datasets are involved since less network traffic is produced.

Benefits of HDFS

1. Fault tolerance: It was designed to recognise mistakes automatically and recover rapidly, ensuring reliability and continuity.

2. Speed: It can store 2GB of data every second due to its cluster design.

3. Access to more types of data

4. Portability and compatibility: Users have the choice to utilise HDFS in their own configuration because it is made to function with a variety of hardware setups and be compatible with a number of underlying OS.

5. Scalable: We can scale resources based on the size of your file system. Both vertical and lateral scaling occur.

6. Data locality: Data resides in data nodes rather than travelling to the location of the compute unit. Network congestion is decreased, and the system becomes more effective and efficient by shortening the distance between the data and the processing process.

7. Economical: Because HDFS data is virtual, the cost of keeping namespace and file system information may be drastically reduced.

8. Has a large amount of storage space for data of various sizes and types.

9. Versatile: Before being saved, data must be processed. can store almost any amount of information.

READ HDFS

The Hadoop Distributed File System (HDFS) is a scalable, fault-tolerant distributed file system designed to run on standard hardware. HDFS is used by Hadoop to store and manage a large amount of data. HDFS stores data in blocks that are duplicated several times to ensure data accessibility in the case of node failures.

The HDFS read method starts when a client asks to read a file that is stored in HDFS. This is a thorough explanation of the HDFS read process:

1. NameNode: The client communicates with the NameNode, the master node of the HDFS cluster. The NameNode keeps track of the metadata for each file in HDFS, including the size, the number of replicas, and the location of the blocks that make up the file.

2. Block addresses: The NameNode communicates the file block addresses to the client.

3. DataNode: The client next contacts the DataNode, which is in responsible of keeping the blocks of the file stored. Each block is stored in a different DataNode and has many replicates to ensure data availability in the case of a DataNode failure.

4. Read Data: The client receives the data from the desired block from the DataNode. If the requested block is not already in the local DataNode as a result of a client request, the DataNode obtains it from a replica node.

5. Merge Data: The client connects the data from each block to generate the whole file.

The very efficient and scalable HDFS read technique can read large volumes of data. The usage of blocks and replicas ensures high data availability even in the case of node failures. The HDFS read method additionally safeguards the data's security and confidentiality by allowing only authorised clients to access it.

In conclusion, the HDFS read process is an essential component of the Hadoop ecosystem and makes it possible to handle and archive data in a scalable and efficient manner. High data availability is achieved via blocks and replicas, and speedy and secure data access is provided through the NameNode and DataNode.

READ

To open the file it wants to see, the client first calls open() on the File System Object.

(which is a type of distributed file system used by HDFS).

Step 2: The Distributed File System (DFS) initiates a remote procedure call (RPC) to the name server to find the first few blocks in the file. The addresses of the data nodes that have copies of each block on them are provided by the name node for each block. The DFS sends the client an FSDataInputStream that may be used to obtain data. FSDataInputStream wraps a DFSInputStream, which manages the I/O for the data node and name node.

The client then employs the stream's read() function in step three. DFSInputStream stores the info node addresses for the first several blocks in the file and connects to the primary (closest) data node for the first block.

At step 4, the data node streams data back to the client, and the stream is regularly queried using the read() method.

Step 5: DFSInputStream selects the best data node for the subsequent block and stops communicating with it when the block is finished. Due to the client's perception that it is merely browsing an infinite stream, it is oblivious of this. Blocks are read as the client traverses the stream, and each time a new connection to a data node is made, it uses the DFSInputStream. It will also occasionally make a call to the name node to obtain the positions of the data nodes for the following set of blocks.

Step 6: The FSDataInputStream's method close() is called after the client has finished reading the file.

WRITE HDFS

The HDFS (Hadoop Distributed File System) is designed for Hadoop's big data processing and offers a scalable and reliable storage solution for storing enormous data sets. To publish a file in HDFS, do the following steps:

1. File Creation: The client must send a request to the Namenode with the file name, replication factor, and block size requested in order to create a new file. The Namenode returns a list of Datanodes that the client can use to store the data blocks.

2. Data Split: The client divides the data into blocks of equal size and sends the first block to the Datanode at the top of the list. The client then writes the next block to the second Datanode once all previous blocks have been written.

3. Data duplication: To ensure the data's high availability and reliability, HDFS uses data duplication. As a result, different Datanodes store different versions of the same data block. The user selects the duplication factor while creating a file.

4. File Closing: The client asks the Namenode to end the file after verifying that all blocks have been correctly written.

5. Metadata Update: The Namenode modifies its metadata to reflect the newly generated file as well as any related blocks and replication information. This information is used by the Namenode to control the file system namespace and to direct clients to the appropriate Datanode for data retrieval.

6. Data transfer: Data movement between the client and the Datanode is accomplished via a trustworthy, efficient protocol, such as Data Transfer Protocol. (DTP).

7. Checksums: HDFS provides checksums for data blocks to ensure data integrity and detect any mistakes that may have occurred during data transmission. Before being transferred to the Datanode, the client verifies the integrity of each block. The Datanode further calculates its own checksum for each block to verify data integrity and compares it to the checksum generated by the client.

8. Mistake Correction: In the event of a write mistake, HDFS may utilise the checksum data to quickly locate and correct the issue. The Namenode can also discover corrupted blocks and replace them with them if any good copies are available.

For storing massive volumes of data in Hadoop, the HDFS write operation provides a scalable, stable, and efficient solution. Because the use of data replication, checksums, and error correction procedures ensures high availability and data integrity, HDFS is an essential component of the Hadoop ecosystem.

WRITE

First, the client creates the file by using create() on the DistributedFileSystem.

(DFS).

Step 2: DFS makes an RPC call to the name node to create a new file in the namespace of the file system without any blocks attached to it. The name node performs a variety of checks to ensure that the file is not already present and that the client is authorised to create it. The name node generates a record of the new file if all of these checks are successful; otherwise, the client gets an IOException exception since the file cannot be created. The DFS provides an FSDataOutputStream, which the client may start using to transfer data to.

Step 3: The DFSOutputStream divides the client's inputted data into packets, which it then transfers to a private information queue. Before calling the name node to assign new blocks, the DataStreamer consumes the data queue and is in charge of choosing a list of suitable data nodes to store the replicas. The list of data nodes creates a pipeline; in this example, since the replication level is 3, we'll assume that there are three nodes in the pipeline. The packets are received from the DataStreamer by the main data node in the pipeline, where they are stored before being sent to the secondary data node.

Step 4: The second data node in the pipeline stores the packet in a similar manner, then passes it on to the third and last data node.

Step 5: The DFSOutputStream keeps a list of messages that are awaiting acknowledgement from data nodes in an internal "ack queue."

Step 6: After delivering the last packets to the data node's pipeline and waiting for acknowledgements, this process connects to the name node to determine if the file is complete or not.

REDUCE MAP

For managing big data sets on a cluster, the programming paradigm MapReduce and its associated implementation are used. Massive volumes of data stored in HDFS are managed using MapReduce, the central component of the Hadoop ecosystem (Hadoop Distributed File System).

The two main stages of the MapReduce paradigm are the Map stage and the Reduce stage.

Map Stage: Each component of the input data collection is subjected to a user-defined mapping function in the Map stage to create intermediate key-value pairs. The mapping function processes each incoming record to produce zero or more intermediate key-value pairs. The intermediate key-value pairs are grouped by key before proceeding to the Reduce stage.

Lower Stage: Each unique key in the intermediate key-value pairs generated by the Map stage is subjected to a user-defined reduction function in the Reduce stage. The reduction function processes each value associated with each unique key, producing zero or more output values.

Think about the following instance: You need to count the instances of each term in a sizable dataset of text documents. The following is a MapReduce solution that resolves this problem:

Reading each document, tokenizing it into words, and emitting a key-value pair for each word are all steps in the second stage of the mapping process. The following key-value pairs, for example, might be generated by the mapping function for a document containing the text "The swift brown fox jumps over the slow dog":

("The," "1," "quick," "1," "fox," "1," "jumps," "1," "over," "1," "lazy," "1") ("dog", 1)

Lower Stage: In the intermediate key-value pairs from the Map phase, the reduction function adds values for each unique key. With the aforementioned intermediate key-value combinations, for example, the reduction function may provide the results seen below:

The, 2 (Lazy, 1), (The), 2 (Fast, 1), (Brown, 1), (Fox), (Jumps, 1), and (Over, 1) ("dog", 1)

The example above demonstrates how MapReduce may be used to solve a simple word count problem. The MapReduce programming style may be used to handle a wide range of problems, including challenging data processing tasks like data aggregation, filtering, and transformations.

MRv1

For Hadoop's huge data processing, the MapReduce programming paradigm was initially implemented as MRv1 (MapReduce version 1). Its two main components, the JobTracker and the TaskTracker, are designed to process big data sets simultaneously over a cluster of commodity computers.

Here is a detailed examination of MRv1:

1. JobTracker: As the MRv1 architecture's master node, the JobTracker is in charge of managing the MapReduce jobs that clients deliver. It controls data dissemination, assigns tasks to certain cluster nodes, and monitors job accomplishment.

2. TaskTracker: A slave node in the MRv1 architecture, the TaskTracker is in responsibility of doing the tasks that the JobTracker assigned to it. When executing on a different system, each TaskTracker connects with the JobTracker to report the progress of its tasks.

3. Map Tasks: The Map Tasks are in charge of analysing the incoming data and generating intermediate key-value pairs. They are the initial step in the MapReduce process. Each block of data is handled by a different Map job, which executes on a per-block basis while running concurrently on the TaskTrackers.

4. Reduce Tasks: At the MapReduce process' second stage, Reduce tasks are in charge of merging the intermediate data that the Map tasks have generated. While they run concurrently on the TaskTrackers, the Reduce tasks use the intermediate data to create the final result.

5. Shuffling: The transfer of intermediary data generated by the Map tasks to the Reduce tasks is the process of shuffling, where this data is sorted and grouped by key to enable the Reduce task to aggregate the values for each key. The shuffling procedure is efficient and parallel because many Map processes send their intermediate data in parallel to multiple Reduce tasks.

6. Data Flow: The data flow of MRv1 is well optimised for large data processing, with input blocks being processed in parallel by several Map activities while intermediate data is jumbled and aggregated by many Reduce tasks. This makes it possible for MRv1 to analyse enormous data sets in a very scalable and efficient manner.

7. Scalability: MRv1 is designed to be incredibly scalable, enabling the cluster to grow as the size of the data set does. As a result, MRv1 now has a configurable big data processing solution that makes it simple to manage massive data volumes.

In conclusion, MRv1 implements the MapReduce programming paradigm for Hadoop's large data processing in a very effective and scalable way. Due to its master-slave architecture, parallel data processing, and efficient shuffling and aggregation of intermediate data, MRv1 is an essential component of the Hadoop ecosystem.

MapReduce is an associated implementation of a distributed, parallel method for handling large data sets on a cluster.

The following are the elements of the MapReduce version 1 design:

The task Tracker is the master node that manages the cluster's task execution and interacts with other task trackers.

Task Tracker: According to instructions from the Job Tracker, this cluster server performs tasks.

NameNode: This master node manages the file system domain and regulates client access to files.

Data in the Hadoop Distributed File System are stored on a node in the cluster called a "DataNode" (HDFS).

Here is a thorough description of how MapReduce works:

When a client sends a task to the Job Tracker, they must also provide the input and output locations as well as the MapReduce programme that will be utilised for the work.

A map job is sent to a job tracker for each chunk that the job tracker splits the data into.

The Task Tracker loads the data and then applies the map function to each element in the data. The map function processes each record, producing a set of intermediate key-value pairs in the process.

In the task tracker, the intermediate key-value pairs are sorted and organised by key. The intermediate key-value pairs that have been sorted are then passed on to the reducer tasks.

The task trackers get the reduce jobs from the job tracker.

After applying the reduce function to the intermediately sorted key-value combinations, the Task Tracker aggregates the values for each key to produce the final result.

After the final product is saved in the HDFS, the task is complete.

This process can be repeated for more map and reduce phases if the application requires it.

A new approach to the MapReduce programming paradigm for Hadoop's huge data processing is MRv2 MapReduce version 2 (MRv2). By addressing several of MRv1's shortcomings, such as scalability, fault tolerance, and resource management, it provides a more efficient and scalable choice for large data processing.

Here is a thorough evaluation of MRv2:

1. YARN: YARN is the new resource management tool for Hadoop that MRv2 introduced (Yet Another Resource Negotiator). Dynamic resource allocation is possible because to the scalable and flexible resource management technology known as YARN.

The NodeManager, a unique component of MRv2, is in charge of managing the resources on each cluster node. It notifies the ResourceManager about the CPU, memory, and disc use of each node.

3. ResourceManager: The master node in the MRv2 architecture, the ResourceManager is in charge of allocating resources among the cluster's many applications. Based on the needs of the various programmes, it chooses how to distribute resources after receiving information on resource utilisation from the NodeManager.

4. ApplicationMaster: New to MRv2, the ApplicationMaster is in charge of overseeing the MapReduce application and coordinating resource allocation with the ResourceManager. It communicates with the NodeManager to control how each node in the cluster completes its tasks.

5. Map and Reduce Tasks: Compared to MRv1, MRv2 Map and Reduce Tasks offer greater resource management and fault tolerance. The Map tasks process the incoming data to build intermediate key-value pairs, while the Reduce tasks put the intermediate data together to create the final output.

6. Better Shuffling: MRv2 adds a method of shuffling that is more efficient and resistant to data loss. The shuffling process is streamlined for parallel processing, and intermediate data is passed between Map and Reduce processes more consistently and successfully.

7. Scalability: MRv2 is designed to be extremely scalable, enabling the cluster to grow as the volume of the data gathering does. As a result, MRv2 now has a configurable big data processing solution that makes managing massive data sets simple.

MRv2 offers a more flexible, efficient, and scalable solution for Hadoop's large data processing, and is a significant improvement over MRv1. Being a key component of the Hadoop ecosystem, MRv2 now offers superior resource management, shuffling, and scalability in addition to YARN, NodeManager, ResourceManager, and ApplicationMaster.

With MapReduce version 2 (MRv2), also known as YARN (Yet Another Resource Negotiator), which allows more flexible resource management and supports a larger range of processing techniques besides MapReduce, the MapReduce architecture has been changed.

The components of the MRv2 architecture are as follows:

Resource Coordinator: The main node communicates with node managers and manages the cluster's resources.

The containers are managed by a node in the cluster known as the node manager, which also keeps track of how much resource each container uses and provides that data to the resource manager.

Master Application: It interacts with Node Managers to carry out and maintain track of obligations, and it bargains with the Resource Manager for resources.

Resources (such memory and CPU) are allotted in a container to carry out a task.

A detailed description of how MRv2 works is provided below:

The client notifies the resource management of a task request, including the resources required, the input and output locations, and the programme to be executed.

An application master, who is in charge of allocating resources and supervising tasks, is given a job by the resource manager.

When the input has been split into tasks, the Application Master negotiates the Resource Manager's resources.

After receiving the tasks from the Application Master, the Node Managers carry out the tasks in containers.

The Node Managers are in charge of carrying out the activities; they also monitor resource use and provide updates to the Application Master.

The application master combines the job output and saves the final product in the file system.

The job is deemed complete when all of the tasks have completed running and the final output has been stored to the file system.

Keep in mind that MRv2 supports more processing paradigms than only MapReduce and can execute a variety of applications including batch processing, interactive processing, graph processing, and streaming.

The main difference between MRv1 and MRv2 in terms of recovery is how they handle work failures.

When a process failed in MRv1, the entire process was restarted, which meant that any work done up until that point was lost. Because there was no coordination between the JobTracker and the TaskTracker to handle a task's progress, it was also difficult to recover from task failures.

On the other hand, the new component introduced by MRv2 called the ApplicationMaster is in charge of overseeing task execution and controlling the application. When a job fails, the ApplicationMaster can identify the problem and hand off control to another server so that it can continue where it left off. As a result, less effort is lost and task failure recovery is made easier.

The NodeManager, which is in charge of monitoring each node's state inside the network, is also a component of MRv2. If a node fails, the ResourceManager can provide resources to another node to complete the task, and the NodeManager can recognise the failure and notify it.

In general, MRv2's improved recovery method makes it a more reliable option for big data processing in Hadoop by increasing its resilience to task failures and reducing the amount of lost work.

+++++

Large data sets can be processed in parallel and distributed ways using the MapReduce version 1 (MRv1) and version 2 (MRv2), also known as YARN (Yet Another Resource Negotiator). Nonetheless, there are a few notable differences between MRv1 and MRv2:

Resource Management: When utilising MRv1, the Job Tracker is in charge of resource management, including delegating map and reduce tasks to Task Trackers. In MRv2, the Resource Manager is in charge of managing resources, while the Application Master is in charge of processing tasks and negotiating resources.

Scalability: The ability to support a significantly higher node count and a more flexible resource management system give MRv2 greater scalability than MRv1.

freedom: MRv2 provides more freedom because it supports a wider range of applications in addition to MapReduce, such as batch processing, interactive processing, graph processing, streaming, and other processing paradigms.

Processing Model: MRv1 is exclusively built for the MapReduce processing model, whereas MRv2 can support a wider range of processing models, including MapReduce and other processing paradigms.

Task management: In MRv1, it is the Job Tracker's responsibility to supervise the jobs and ensure their completion. In MRv2, it is the Application Master's responsibility to supervise the tasks and ensure their completion.

The major improvements that MRv2 offers over MRv1 are a more flexible resource management system, improved scalability, and more flexibility.

REDUCING MAP RECOVERY FAILURE

The MapReduce programming approach enables programmers to manage large datasets in a distributed environment. The Map and Reduce phases of a data processing operation are separated in MapReduce.

During the Map stage, the incoming data is divided into smaller chunks and processed simultaneously by a number of cluster nodes. The outcome of the Map phase is a collection of intermediate key-value pairs.

In the Reduce phase, the intermediate key-value pairs produced by the Map phase are merged, sorted, and processed to produce the final outcome.

In a distributed environment, failures are inevitable. Many factors, including as network difficulties, hardware or software faults, etc., might cause nodes to fail. To make sure the MapReduce operation is properly completed, a robust failure recovery mechanism is required.

Here are several techniques for recovering from MapReduce errors:

A failed Map or Reduce job can be retried by the MapReduce framework on a different node. A customizable number of times can pass before the job permanently succeeds or fails.

2. Speculative Execution: If a Map or Reduce job is taking longer than expected, the framework might start a duplicate task on a separate node. The job that is successfully finished first is utilised, and the alternative is dropped. The execution time of this procedure can be drastically reduced by making use of unused resources.

3. Another choice is to start the backup jobs simultaneously with the primary tasks. These tasks aren't used unless the main job fails. In the case that the primary job fails, the backup task takes over to ensure that the work is finished.

4. Checkpointing: Interim results may frequently be saved to a distributed file system, such as HDFS. The framework can pick up where it left off in order to minimise the amount of work that needs to be completed once a job fails.

5. Task Observation: The MapReduce architecture can track the status of each task and quickly spot errors. When a job fails, the framework has the ability to record the error message and notify the administrator.

These techniques enable MapReduce to handle failures in a distributed environment and ensure the successful completion of jobs involving enormous data processing.

Architecture by Apache Hive

A query language (HiveQL) similar to SQL is provided by the data warehousing tool Apache Hive for exploring and analysing big datasets stored in Hadoop. The aspects that make up Hive's design are as follows:

1. User Interface: Hive provides a range of platforms for users to interact with the system. The command-line interface (CLI), which is also used to control the system, is how HiveQL searches are executed. The Hive Web Interface provides a web-based graphical user interface (GUI) for interacting with Hive.

2. Driver: The Driver is in charge of managing the lifecycle of a HiveQL query. It converts a user interface query into an execution plan, which it then utilises the cluster to execute.

3. Compiler: The Compiler is in responsible of extracting the HiveQL query's processing plan from it. A logical plan that the compiler generates is converted into a physical plan that can be performed on the cluster.

4. Metastore: The Metastore is the primary repository for knowledge about the data stored in Hive. It maintains track of tables, divisions, columns, as well as the many types of data, their locations, and their formats.

5. Execution Engine: The Compiler's real plan must be carried out by the Execution Engine. Hive supports a wide range of processing engines, including MapReduce, Tez, and Spark.

6. SerDe: The SerDe (Serializer/Deserializer) serialises and deserializes data stored in Hive. It transforms the data between its internal representation and the format required by the storage device.

7. Storage Handler: In charge of interacting with the data storage device. Hive supports a number of storage drivers, including HDFS, HBase, and Amazon S3.

8. Hadoop Distributed File System (HDFS): The main file system for Hadoop is HDFS. For storage, it distributes the data among a number of cluster computers.

9. Apache ZooKeeper is a distributed coordination programme that is used in Hadoop to manage configuration data, synchronisation, and naming services. Hive uses it for global locking and coordination.

To provide a scalable and flexible data warehousing option that can be used to process sizable datasets stored in Hadoop, the Hive architecture was developed. The architecture allows users to use a number of execution engines and storage handlers to customise the system to their requirements.

SPARK APACHE

Apache Spark is an open-source massive data processing engine with an emphasis on speed, scalability, and usability. Spark provides a single framework for processing batch and streaming data, machine learning, graph processing, and interactive searches.

The design of Spark, which makes use of a cluster of computers to process data in parallel, is built on the distributed computing concept. The main Spark design components are as follows:

1. Driver Program: The Driver Program, which is the main programme, manages the Spark application. It maintains the cluster's distributed processing while also running the user's code.

2. Executors: Each cluster node has a running instance of a worker programme referred to as an executor. The driving software assigns them duties, and they are in charge of completing those tasks and managing the memory and resources those activities need.

3. Cluster Manager: The Cluster Manager is in responsible of managing the network traffic, RAM utilisation, and CPU use for the cluster. Spark allows other cluster managers in addition to Hadoop YARN, Apache Mesos, and its own standalone cluster manager.

4. Resilient Distributed Datasets (RDDs): The Spark data model is built on RDDs. These are immutable, partitioned data sets that the cluster is capable of managing concurrently.

5. Spark SQL: Spark SQL provides a structured data querying tool that is comparable to SQL. It enables data kept in many formats, such as Apache HBase, Hadoop Distributed File System, and Apache Cassandra, to be queried.

Spark Streaming: 6. Spark Streaming is a processing engine for streaming data that is adaptable and fault-tolerant. It provides help with ingesting and analysing real-time data streams, such as online logs, sensor data, and social media feeds.

7. MLlib: MLlib is the name of Spark's machine learning framework. It provides a set of high-level APIs for the creation of scalable machine learning algorithms and models.

8. GraphX: GraphX is the name of the network processing library used by Spark. It provides a variety of graph-building and graph-working APIs, including those for social networks, transportation networks, and biological networks.

Essentially, Spark's architecture is designed to provide a flexible and scalable framework for managing huge datasets and performing complex computations on them. Its modular architecture allows users to pick and select the components they need for their specific use cases, and its unified programming interface facilitates the creation and deployment of big data applications.

What separates HADOOP and SPARK?

Hadoop and Spark are two big data processing systems that can handle massive volumes of data. Yet, they differ greatly in terms of their use cases, performance, and architecture.

1. Architecture: Hadoop utilises a distributed file system called Hadoop Distributed File System (HDFS) for storing and managing data, whereas Spark does not and may function on top of other distributed file systems like HDFS, Amazon S3, or Apache Cassandra. Moreover, Hadoop employs a two-stage MapReduce processing model, whereas Spark uses an in-memory processing engine.

2. Performance: Spark often outperforms Hadoop, especially when dealing with large datasets. This is because Spark's in-memory processing engine allows it to run multiple calculations, in contrast to Hadoop's requirement to read and write data to disc between each step of processing.

3. Use Cases: Hadoop is more frequently used for batch processing and storing enormous volumes of data, whereas Spark is better suited for real-time stream processing, machine learning, and interactive data analysis. Additionally, Spark is usually utilised in circumstances that require speedier processing and more complex computations, whereas Hadoop is frequently used in data warehousing and ETL (Extract, Transform, Load) scenarios.

4. Ecosystem: Hadoop has a broader ecosystem than Spark since it has been around longer and has a more established community. This ecosystem's many different tools and technologies, like as Apache Pig, Apache Hive, and Apache Zookeeper, may all be utilised with Hadoop. GraphX, MLlib, and Spark SQL are already part of the community of tools and libraries that make up Spark, but this community is growing swiftly.

The architecture, performance, and use cases of Hadoop and Spark differ significantly from one another. Hadoop is more frequently used for batch processing and data storage, but Spark excels in real-time stream processing, machine learning, and interactive data analysis.

LIKENESS BETWEEN HADOOP AND SPARK

As massive data processing frameworks, Hadoop and Spark share a number of characteristics:

1. Parallel processing of large datasets over a cluster of nodes is made possible by the distributed computing capabilities of both Hadoop and Spark.

2. Open Source: Because Hadoop and Spark are both open-source initiatives, users are allowed to access and modify them as necessary.

3. By adding extra nodes to the network as needed, Hadoop and Spark both provide horizontal scaling.

4. Fault Tolerance: Both Hadoop and Spark are designed to be fault tolerant, meaning they can recover from hardware issues or other types of system faults without losing any data.

5. Batch Processing: Spark and Hadoop may also be used for batch processing, which involves managing enormous volumes of data in a predetermined or routine manner.

6. Ecosystem: Both Hadoop and Spark have robust ecosystems of tools, libraries, and technologies to carry out a range of large data processing tasks.

In conclusion, Hadoop and Spark are both excellent choices for addressing big data processing tasks in a distributed computing context due to their many architectural, functional, and ecosystem similarities.

Database for columns

A columnar database, also known as a column-oriented database or a columnar store, is a database management system (DBMS) that maintains data in columns rather than rows. In a columnar database, each column of data is kept independently and may have a unique data type.

Columnar databases are designed to maximise efficiency and performance for analytical workloads, particularly those involving complicated queries or the aggregation of huge datasets. Because each column is kept independently, columnar databases' ability to access only the exact columns needed by a given query minimises the quantity of data that must be processed and can significantly improve query performance.

Another advantage is that columnar databases may compress data more effectively than row-oriented databases. Each column contains just one type of data, making it easier to compress and store that data more efficiently. This compression can significantly lower the amount of data that has to be read from disc, which lowers storage costs and speeds up queries.

Columnar databases are commonly used in data warehousing, business intelligence, and analytics programmes where high performance and scalability are crucial. Some examples of columnar databases are Apache Cassandra, Apache HBase, Google Bigtable, Amazon Redshift, and Apache Kudu.

The tables of a columnar database are arranged in rows. Fast tabular operations like MIN, MAX, SUM, COUNT, and AVG execution as well as effective data write and read operations to and from hard disc storage are all advantages. Columnar databases allow for random read and write in Hadoop and meet the ACID criteria for a database.

ACUTE PIG

Pig analyses data in Hadoop using a language known as Pig Latin. It is a high-level language for processing data and provides a large range of operators and data types for handling data in various ways.

To carry out a given task while using Pig, programmers must write a Pig script in the Pig Latin language and run it using one of the execution methods (Grunt Shell, UDFs, Embedded). These scripts will undergo a variety of modifications after being executed by the Pig Framework in order to provide the desired output.

These scripts are internally converted by Apache Pig into a series of MapReduce jobs, which makes the programmer's job easier.

There are several components that make up the Apache Pig architecture.

Parser

The Pig Scripts are originally handled by the Parser. It conducts a variety of tests, including type checking and syntactic analysis of the script. The output of the parser will be a directed acyclic graph (DAG), which represents the logical operators and sentences in Pig Latin.

The DAG shows the logical operators of the script as nodes and the data processes as edges.

Optimizer The logical optimizer processes logical optimisations like projection and pushdown after receiving the logical plan (DAG).

Compiler The compiler transforms the optimised logical plan into a series of MapReduce tasks.

Execution engine

Prior to being delivered to Hadoop, the MapReduce tasks are sorted. The desired results are acquired when these MapReduce tasks are eventually executed on Hadoop.

CODING

HDFS

\*\*\* must be root for this

su root

\*\*\* mkdir

hdfs dfs -mkdir /myhdfs

\*\*\* copy from local to hdfs

hdfs dfs -put /home/cloudera/myfiles/datafile.txt /myhdfs/datafile.txt #(proper file path to be copied)

hdfs dfs -ls /myhdfs (list all the files)

\*\*\* delete file

hdfs dfs -rm /myhdfs/delfile.txt

\*\*\* delete directory

hdfs dfs -rmdir /myhdfs

PIG

\*\*\* Pig Commands - Interactive Mode\*\*\*

======================================

\*\*\* subscriber - count bytes exercise \*\*\*

\*\*\* =====================================

### start pig interative mode

pig

Grunt>

### quit pig interative mode

quit

### clear screen

clear

### hdfs commands

su root

hdfs dfs -mkdir /mypig

hdfs dfs -mkdir /mypig/subscriber

hdfs dfs -mkdir /mypig/subscriber/input

hdfs dfs -put /home/cloudera/myfiles/pig-subscriber.txt /mypig/subscriber/input

### pig commands for textfile

### sum bytes of Subscriber

A = load '/mypig/subscriber/input' as (line:chararray); (load file in pig)

B = foreach A generate (chararray)SUBSTRING(line,14,26) as id , (double)SUBSTRING(line,87,97) as bytes;

C = group B by id; (group by if mentioned)

D = foreach C generate group, SUM(B.bytes);

dump D; (to display)

store B into '/mypig/subscriber/output' using PigStorage(','); (to store txt file to csv file on browser)

\*\*\* customer - read csv & write csv exercise \*\*\*

\*\*\* ============================================

### hdfs commands

su root

hdfs dfs -mkdir /mypig

hdfs dfs -mkdir /mypig/customer

hdfs dfs -mkdir /mypig/customer/input

hdfs dfs -put /home/cloudera/myfiles/pig-customer.csv /mypig/customer/input

### pig commands

CustFile = load '/mypig/customer/input' using PigStorage(',') as ( CustId:int, FirstName:chararray, LastName:chararray, Phone:chararray, City:chararray );

dump CustFile;

store CustFile into '/mypig/customer/output' using PigStorage(','); (to store in csv we need to PigStorge(',')

Pig Practice

A = load '/mypractice/hr-prac.txt' as (line:chararray);

B = foreach A generate (chararray)SUBSTRING(line,0,2) as id ,(chararray)SUBSTRING(line,3,22) as name, (chararray)SUBSTRING(line,23,24) as gender,SUBSTRING(line,25,26) as dept,(double)SUBSTRING(line,27,31) as price;

dump B;

store B into '/mypractice/output' using PigStorage(',');

D = group B by id;

F = foreach D generate group, SUM(B.price);

E = FILTER F by (B.price> 7214.0);

dump E;

grunt> SPLIT F into students1 if (B.price>7214.0), students2 if (B.price<=7214.0);

grunt> H = foreach B generate (id,name),LOWER(name);

dump H;

HIVE

\*\*\* Hive Commands \*\*\*

=====================

### version

hive --version

beeline --version

### start

hive

beeline -u jdbc:hive2://

### quit

quit

!quit

### databases / tables

show databases;

use default;

show tables;

create database hr;

use hr;

create table employee (emp\_id string, emp\_name string, salary float, status int) row format delimited fields terminated by ',' lines terminated by '\n';

load data local inpath '/home/cloudera/myfiles/hive-employee.csv' overwrite into table employee; (Data loaded from local)

hive> LOAD DATA inpath 'hdfs:///myproject/output' OVERWRITE INTO TABLE insurance; #(data loaded from hdfs)

desc employee;

select \* from employee;

select \* from employee where salary >= 5000;

select \* from employee where salary = 2000;

select \* from employee where emp\_name like 'cyrus';

select \* from employee where emp\_name like 'cyrus%';

select \* from employee where emp\_name like 'Cyrus%';

select \* from employee where emp\_name like '%ta%';

select \* from employee order by emp\_name; #note - run as mr-job

select count(\*) from employee; #note - run as mr-job

select count(\*) from employee where salary = 2000; #note - run as mr-job

select salary, count(\*) from employee group by salary;

select salary, sum(salary) as sumsal from employee group by salary;

select max(salary) as maxsal, min(salary) as minsal from employee;

EXPLAIN select salary, count(\*) from employee group by salary;

SQOOP

to open sql: mysql -u root -p

# export data ... read from hdfs & store to mysql -- check map reduce job

linux>

hdfs dfs -mkdir /myhdfs/employee

hdfs dfs -put /home/cloudera/myfiles/hive-employee.csv /myhdfs/employee

mysql>

create database employee; #(if db does not exists)

use employee;

create table new\_emp (emp\_id VARCHAR(10), emp\_name VARCHAR(50), salary FLOAT, status INT);

linux>

sqoop export --connect jdbc:mysql://localhost/employee --username root -P --table new\_emp --export-dir /myhdfs/employee --input-fields-terminated-by ',' --lines-terminated-by '\n'