8v’s

The eight characteristics known as "the 8Vs of Big Data" define the scope and nature of big data.

1. Volume: the quantity of data being produced and analysed.

2. Variety: describes the various kinds and configurations of data, such as organized, semi-structured, and unstructured data.

3. Velocity: describes how quickly data is produced and analysed.

4. Veracity: This term describes the data's degree of ambiguity, accuracy, and dependability.

Value is a term that describes the possible gain or monetary value of the data.

6. Visualization: relates to the requirement for data to be secure, comprehensible, and accessible.

Viscosity is a technical term that describes the resistance or friction that is experienced when trying to alter how data is gathered, saved, processed, or analysed. High viscosity can restrict the value that can be extracted from the data and make it challenging to make changes to the data management process.

8. "Virality" describes the propensity for data to proliferate quickly and uncontrollably, much like a virus does. This can happen when data is shared across networks and platforms, which can be difficult for businesses to handle in terms of data privacy and security.

HDFS

Designed for Hadoop's large data processing, HDFS (Hadoop Distributed File System) is a scalable and trustworthy storage system. One computer serves as the Namenode and several other machines serve as Datanodes in a master-slave architecture.

The metadata, such as the position of blocks for a specific file, and the file system namespace are maintained by the Namenode. The Datanodes are in charge of keeping the real data blocks and responding to client read/write requests.

Here is a detailed explanation of the HDFS architecture:

1. Namenode: The HDFS cluster's master node, known as the Namenode, is in charge of keeping the mapping of blocks to Datanodes as well as handling file and directory operations in the file system namespace. The metadata for the file system, including the position of blocks, the replication factor, and the access time, is also kept up to date by the Namenode. Two folders are connected to the meta data:

I. Fs Image: It includes all of the information regarding the file system namespace since the name node's inception.

ii. Edit Logs: These records all recent file system changes made in relation to the most recent FSImage.

To confirm that the DataNodes are live, it routinely gets a heartbeat and a block report from every Data node in the cluster. The other node with the same data is triggered if the requested data is not received within 3 seconds.

2. Datanode: The HDFS cluster's slave node, the Datanode is in charge of storing the real data blocks. When a client wishes to read or write a file, it first communicates with the Namenode to find out where the data blocks are located before speaking directly with the relevant Datanode to get the data or store it.

3. Blocks: The blocks can be stored in numerous chunks and are kept on the Datanodes. The replication factor is chosen by the user when the file is created, and the blocks are also replicated for failure tolerance and high availability.

4. Replication: For high availability and reliability, HDFS offers data replication to make sure that multiple copies of the same data block are kept on various Datanodes. For data retrieval, the Namenode can point clients to the replica that is the nearest by keeping track of all the replicas of a block. The Namenode can instantly reroute clients to another replica of the same data block in the event of a Datanode failure.

5. Data Integrity: To guarantee data integrity and identify any errors during data transmission, HDFS offers checksums for data blocks. The Namenode occasionally verifies the data blocks' integrity and replaces damaged blocks with reliable replicas.

In conclusion, the master-slave HDFS design, with its master-slave architecture for effective data management and parallel processing, offers a scalable and dependable storage system for big data processing in Hadoop. Due to its use of blocks and data replication, which guarantees both data integrity and high availability, HDFS is a crucial part of the Hadoop infrastructure.

• Secondary Name Node (Name Node's Aide)

The FSImage is updated with the file's modifications and saved, after which the image is once more kept in the name node.

HDFS Target

1. Organizing and storing large datasets can be a difficult job to manage. Applications that need to handle large datasets are managed by HDFS. HDFS needs hundreds of servers in each cluster to accomplish this.

2. Finding faults: Given that HDFS used a lot of common hardware, it should have technology in place to scan and find faults swiftly and efficiently. The component failing is a frequent problem.

3. Hardware effectiveness: When there are big datasets involved, less network traffic is generated, which speeds up processing.

ADVANTAGES of HDFS

1. Fault tolerance: It was created to automatically identify errors and quickly recover, ensuring dependability and continuity.

2. Quickness: Thanks to its cluster architecture, it can keep 2GB of data every second.

3. Availability of more kinds of data

4. Portability and compatibility: Designed to work with a variety of hardware configurations and be compatible with a number of underlying OS, HDFS gives users the option to use it in their own arrangement.

5. Scalable: Depending on the extent of your file system, we can scale resources. Scaling is both vertical and lateral.

6. Data locality: Instead of moving to the location of the computational unit, data lives in data nodes. By reducing the distance between data and the processing process, network congestion is reduced, and the system becomes more effective and efficient.

7. Economical: Since HDFS data is virtual, the expense of storing file system metadata and namespace data can be significantly reduced.

8. Can store a lot of data, including data of all shapes and sizes.

9. Flexible: Data must be processed before being stored. can hold virtually any amount of info.

HDFS READ

The Hadoop Distributed File System (HDFS) is a distributed file system that is made to operate on common hardware and is scalable and fault-tolerant. Hadoop utilises HDFS to store and handle a lot of data. To guarantee data availability in the event of node failures, HDFS stores data in blocks that are replicated multiple times.

When a client requests to read a file stored in HDFS, the HDFS read procedure begins. A detailed description of the HDFS read procedure is provided below:

1. NameNode: The client makes communication with the NameNode, the HDFS cluster's master node. The metadata of every file in HDFS, including the size, the number of replicas, and the location of the blocks that make up the file, is kept note of by the NameNode.

2. Block addresses: The NameNode sends the client the file's block addresses.

3. DataNode: The client then makes communication with the DataNode, which is in charge of storing the file's blocks. To guarantee data availability in the event of a DataNode failure, each block is stored in a separate DataNode and has multiple replicas.

4. Read Data: The DataNode gives the client the data from the desired block. The DataNode gets the requested block from a replica node if it is not already present in the local DataNode due to a client request.

5. Combine Data: To create the full file, the client joins the data from every block.

Large amounts of data can be read using the extremely effective and scalable HDFS read method. High data availability is ensured, even in the event of node outages, by the use of blocks and replicas. By limiting access to the data to only authorised clients, the HDFS read procedure also protects the confidentiality and security of the data.

In conclusion, the HDFS read process is a crucial part of the Hadoop ecosystem and enables scalable and effective data handling and archiving. Blocks and replicas are used to ensure high data availability, and the NameNode and DataNode are involved to ensure quick and safe data access.

READ

Step 1: The client calls open() on the File System Object to open the file it wants to view.(which for HDFS is an instance of Distributed File System).

Step 2: To locate the first few blocks in the file, the Distributed File System (DFS) makes a remote procedure call (RPC) to the name server. The name node gives the addresses of the data nodes that possess copies of each block for each block. The client receives an FSDataInputStream from the DFS, which it can use to retrieve data. A DFSInputStream, which controls the I/O for the data node and name node, is wrapped by FSDataInputStream.

Step 3: The client then uses the stream's read() method. DFSInputStream connects to the main (closest) data node for the first block in the file after having stored the info node addresses for the first few blocks within the file.

In step 4, data is streamed back to the client from the data node and read() is frequently called on the stream.

Step 5: After determining the best data node for the following block, DFSInputStream cuts off communication with the data node when the block's conclusion is reached. The client is unaware of this because, in its eyes, it is just perusing an endless stream. As the client scans through the stream, blocks are read as, with the DFSInputStream creating new connections to data nodes. When necessary, it will also make a call to the name node to get the locations of the data nodes for the upcoming group of blocks.

Step 6: After the client has completed reading the file, the FSDataInputStream's function close() is called.

HDFS WRITE

The HDFS (Hadoop Distributed File System) provides a scalable and dependable storage system for storing large data sets and is intended for Hadoop's big data processing. The following steps are involved in publishing a file in HDFS:

1. File Creation: To create a new file, the client must submit a request to the Namenode with the file name, replication factor, and block size specified. A list of Datanodes that the client can use to keep the data blocks is returned by the Namenode.

2. Data Split: The client divides the data into equal-sized blocks and sends the first block to the top-most Datanode in the list. Once all blocks have been written, the client writes the subsequent block to the second Datanode.

3. Data duplication: HDFS makes use of data duplication to guarantee the data's high availability and dependability. The same data block is stored in numerous copies on various Datanodes as a result. When a file is created, the user chooses the duplication factor.

4. File Closing: After confirming that all blocks have been properly written, the client requests that the Namenode close the file.

5. Metadata Update: The Namenode updates its metadata to represent the newly created file, along with any associated blocks and replication details. The Namenode manages the file system namespace using this metadata, and it also uses it to guide clients to the proper Datanode for data retrieval.

6. Data movement: A dependable, effective protocol, such as Data Transfer Protocol, is used for data movement between the client and the Datanode. (DTP).

7. Checksums: To guarantee data integrity and identify any errors during data transfer, HDFS offers checksums for data blocks. Each block is checked for integrity by the client before being sent to the Datanode. To confirm data integrity, the Datanode also computes its own checksum for each block and contrasts it with the checksum produced by the client.

8. Correction of Errors: If a write error occurs, HDFS can use the checksum data to instantly identify and fix the problem. If any good replicas are accessible, the Namenode can also find corrupted blocks and replace them with them.

In summary, the HDFS write procedure offers a scalable, dependable, and effective method for storing large amounts of data in Hadoop. HDFS is a key part of the Hadoop ecosystem because the use of data replication, checksums, and error correction mechanisms guarantees high availability and data integrity.

WRITE

Step 1: The client calls create() on DistributedFileSystem to make the file.(DFS).

Step 2: To create a new file in the namespace of the file system, with no blocks associated with it, DFS performs an RPC call to the name node. To make sure the file doesn't already exist and that the client has the proper rights to create it, the name node runs a number of checks. If all of these tests are successful, the name node creates a record of the new file; if not, the client receives an IOException error because the file cannot be made. The client can begin sending data to the FSDataOutputStream that the DFS returns.

Step 3: The DFSOutputStream splits the data the client puts into packets, which it then sends to an internal queue called the info queue. The DataStreamer consumes the data queue and is responsible for selecting a list of appropriate data nodes to store the replicas before invoking the name node to allocate new blocks. A pipeline is formed by the list of data nodes; in this case, we'll presume that there are three nodes in the pipeline because the replication level is 3. The main data node in the pipeline receives the packets from the DataStreamer and stores them before sending them on to the secondary data node.

Step 4: In a similar way, the second data node in the pipeline saves the packet and sends it on to the third (and final) data node.

Step 5: The DFSOutputStream maintains an internal "ack queue" of messages awaiting acknowledgement from data nodes.

Step 6: This procedure connects to the name node to indicate whether the file is full or not after sending all of the remaining packets to the pipeline of the data node and waiting for acknowledgements.

MAP REDUCE

A parallel, distributed algorithm called MapReduce is a programming paradigm and its related implementation for handling large data sets on a cluster. The core of the Hadoop ecosystem, MapReduce, is used to handle massive amounts of data kept in HDFS. (Hadoop Distributed File System).

The Map stage and the Reduce stage are the two primary stages of the MapReduce model.

Map Stage: In the Map stage, a user-defined mapping function is applied to each element of the input data collection to produce intermediate key-value pairs. Each incoming record is processed by the mapping function, which results in zero or more intermediate key-value pairs. To move on to the Reduce step, the intermediate key-value pairs are grouped by key.

Reduce Stage: The Reduce stage uses a user-defined reducing function to each distinct key in the intermediate key-value pairs produced by the Map stage. Each value connected to each distinct key is processed by the reduction function, which produces zero or more output values.

Consider the following example: You want to tally the number of times each word appears in a big dataset of text documents. A MapReduce solution that addresses this issue is as follows:

Stage 2 of the mapping process involves reading each document, tokenizing it into words, and emitting a key-value combination for each word. The mapping function, for instance, might produce the key-value pairs shown below for a document having the phrase "The quick brown fox jumps over the lazy dog":

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

Lower Stage: The reducing function adds the values for each distinct key in the intermediate key-value pairs from the Map step. For the above intermediate key-value combinations, the reducing function, for instance, might result in the output shown below:

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

The aforementioned illustration shows how MapReduce can be used to address a straightforward word count issue. Numerous issues can be solved using the MapReduce programming model, including complicated data processing jobs like data aggregation, filtering, and transformations.

MRv1

The MapReduce programming paradigm was first implemented as MRv1 (MapReduce version 1) for Hadoop's big data processing. The JobTracker and the TaskTracker are its two major parts, and they're made to process large data sets concurrently across a cluster of commodity machines.

Here is a thorough analysis of MRv1:

1. JobTracker: The master node of the MRv1 architecture, the JobTracker is in charge of overseeing the MapReduce tasks provided by clients. It manages the distribution of data, assigns tasks to specific cluster nodes, and keeps track of task completion.

2. TaskTracker: The TaskTracker, a slave node in the MRv1 architecture, is in charge of carrying out the duties that the JobTracker delegated to it. Every TaskTracker communicates with the JobTracker to tell the status of its tasks while running on a separate machine.

3. Map Tasks: The first stage of the MapReduce process, the Map Tasks are in charge of processing the incoming data and producing intermediate key-value pairs. Each block of data is processed by a distinct Map task, which runs concurrently on the TaskTrackers and executes on a per-block basis.

4. Reduce Tasks: The second stage of the MapReduce procedure, Reduce tasks are in charge of combining the intermediate data produced by the Map tasks. The Reduce tasks utilise the intermediate data to produce the final output as they execute concurrently on the TaskTrackers.

5. Shuffling: The transfer of intermediary data produced by the Map tasks to the Reduce tasks is the process of shuffling. In order for the Reduce task to aggregate the numbers for each key, this data is sorted and grouped by key. Multiple Map tasks transmit their intermediate data in parallel to multiple Reduce tasks, making the shuffling process efficient and parallel.

6. Data Flow: MRv1's data flow is highly optimised for big data processing, with intermediate data being shuffled and aggregated by multiple Reduce tasks while data blocks are handled in parallel by multiple Map tasks. This enables the extremely scalable and effective processing of large data sets by MRv1.

7. Scalability: MRv1 is built to be extremely scalable, allowing for the addition of new nodes to the cluster as the size of the data set increases. This gives MRv1 a flexible big data processing solution and enables it to manage large data sets with ease.

In conclusion, the MapReduce programming paradigm for Hadoop's big data processing is implemented in MRv1 in a highly scalable and effective manner. MRv1 is a crucial part of the Hadoop ecosystem due to its use of a master-slave design, parallel data processing, and effective shuffling and aggregation of intermediate data.

A parallel, distributed algorithm for processing big data sets on a cluster is called MapReduce, and it also has an associated implementation.

The components of the MapReduce version 1 design are as follows:

The master node that oversees the cluster's task execution and communicates with Task Trackers is known as the task Tracker.

Task Tracker: This cluster server executes tasks as instructed by the Job Tracker.

NameNode: This master node controls client access to files and oversees the management of the file system domain.

A node in the cluster known as a "DataNode" holds data in the Hadoop Distributed File System. (HDFS).

Here is a detailed explanation of how MapReduce functions:

The MapReduce programme to be used for the job is specified, along with the input and output locations, by the client when they send a job to the Job Tracker.

The Job Tracker divides the input into chunks and gives a job Tracker a map job for each chunk.

Each entry in the data is subjected to the map function by the Task Tracker after it has loaded it. Each record is processed by the map function, which also generates a collection of intermediate key-value pairs.

The intermediate key-value combinations are sorted and grouped by key in the task tracker. The reducer jobs are then given the sorted intermediate key-value pairs as input.

The reduce jobs are given to the task trackers by the job tracker.

In order to create the final output, the Task Tracker aggregates the values for each key after performing the reduce function on the intermediately sorted key-value combinations.

The job is finished when the end product is stored in the HDFS.

If the application calls for it, this procedure can be repeated for additional map and reduce steps.

MRv2

MapReduce version 2 (MRv2) is a fresh take on the MapReduce programming paradigm for Hadoop's big data processing. It offers a more effective and scalable option for big data processing by addressing some of the drawbacks of MRv1, including scalability, fault tolerance, and resource management.

Here is a detailed analysis of MRv2:

1. YARN: The new resource management solution for Hadoop introduced by MRv2 is YARN (Yet Another Resource Negotiator). A scalable and adaptable resource management system called YARN allows for dynamic resource allocation based on the requirements of various apps.

2. NodeManager: A novel element in MRv2, the NodeManager is in charge of overseeing the resources on each cluster node. It keeps track of each node's CPU, memory, and disc utilisation and informs the ResourceManager of it.

3. ResourceManager: The ResourceManager, the master node in the MRv2 architecture, is in charge of distributing resources among the cluster's various apps. It gets data on resource usage from the NodeManager and decides how to distribute resources based on the requirements of the various applications.

4. ApplicationMaster: A novel element in MRv2, the ApplicationMaster is in charge of managing the MapReduce application and negotiating resources with the ResourceManager. To oversee the completion of duties on each node in the cluster, it interacts with the NodeManager.

5. Map Tasks and Reduce Tasks: The Map and Reduce tasks in MRv2 have better resource management and failure tolerance than those in MRv1. While the Reduce tasks assemble the intermediate data to create the end output, the Map tasks process the input data to produce intermediate key-value pairs.

6. Improved Shuffling: MRv2 introduces a more effective and data loss-resistant shuffling method. The shuffling procedure is parallel processing optimized, and intermediate data is transferred more reliably and effectively between Map and Reduce jobs.

7. Scalability: MRv2 is made to be very scalable, allowing for the addition of new nodes to the cluster as the size of the data collection increases. This gives MRv2 a flexible big data processing solution and enables it to manage large data sets with ease.

In summary, MRv2 is a major advancement over MRv1 and offers a more adaptable, effective, and scalable solution for Hadoop's big data processing. A crucial part of the Hadoop ecosystem, MRv2 now includes YARN, NodeManager, ResourceManager, and ApplicationMaster in addition to better resource management, shuffling, and scaling.

The MapReduce architecture has been updated with MapReduce version 2 (MRv2), also referred to as YARN (Yet Another Resource Negotiator), which offers more flexible resource management and supports a wider range of processing methods besides MapReduce.

Following are the elements of the MRv2 architecture:

Resource Manager: The main node controls the cluster's resources and communicates with Node Managers.

A node in the cluster known as the node manager runs the containers, keeps track of their resource consumption, and reports that information to the resource manager.

Application Master: It negotiates for resources from the Resource Manager and collaborates with Node Managers to carry out and keep track of duties.

To perform a task, resources (such as memory and CPU) are allocated in a container.

Here is a step-by-step explanation of how MRv2 functions:

The client sends a job request to the resource manager, detailing the resources needed, the input and output locations, and the programme to run.

The Resource Manager assigns a task to an Application Master, who is in charge of arranging resources and overseeing tasks.

The Resource Manager's resources are negotiated by the Application Master after the input has been divided into jobs.

The Node Managers execute the tasks in containers after receiving the tasks from the Application Master.

The tasks are carried out by the Node Managers, who also keep an eye on resource utilisation and update the Application Master on their progress.

The job output is combined by the application master, who then stores the finished product in the file system.

When all of the tasks have done running and the final output has been saved to the file system, the job is considered to be finished.

Remember that MRv2 can run different kinds of applications, such as batch processing, interactive processing, graph processing, and streaming, in addition to supporting a broader range of processing paradigms than just MapReduce.

Regarding recovery, the primary distinction between MRv1 and MRv2 is how they approach job failures.

process failures in MRv1 were handled by restarting the full process, which meant that any work completed up until that point was lost. It was also challenging to bounce back from task failures because there was no coordination between the JobTracker and the TaskTracker to handle a task's progress.

The ApplicationMaster, on the other hand, is a novel component that MRv2 introduces and is in charge of managing the application and coordinating task execution. When a task fails, the ApplicationMaster can recognise the issue and transfer control of the task to another server, allowing it to resume where it left off. This lessens the quantity of work that is lost and facilitates task failure recovery.

In addition, MRv2 also includes the NodeManager, which is in charge of keeping track of each node's condition within the network. The ResourceManager can allocate resources to another node to carry out the job if a node fails, and the NodeManager can identify the failure and report it to it.

In general, MRv2's enhanced recovery method increases its resilience to task failures and decreases the amount of lost work, making it a more dependable option for big data processing in Hadoop.

Difference between MRv1 and MRV2

Both MapReduce version 1 (MRv1) and version 2 (MRv2), also referred to as YARN (Yet Another Resource Negotiator), are methods for parallel, distributed processing of large data sets. However, there are a number of significant variations between MRv1 and MRv2:

Resource Administration: When using MRv1, the Job Tracker is in charge of managing resources, which includes assigning map and reduce jobs to Task Trackers. In MRv2, the Application Master is in charge of handling tasks and negotiating resources, while the Resource Manager is in charge of managing resources.

Scalability: MRv2 has greater scalability than MRv1 due to its capacity to support a much higher node count and more adaptable resource management system.

freedom: MRv2 offers more freedom as it can run a variety of applications, including batch processing, interactive processing, graph processing, streaming, and other processing paradigms in addition to MapReduce.

Processing Model: While MRv2 can support a broader variety of processing models, including MapReduce and other processing paradigms, MRv1 is designed exclusively for the MapReduce processing model.

Task Management: In MRv1, the Job Tracker is in charge of overseeing the jobs and making sure they are finished. The Application Master in MRv2 is in charge of overseeing the duties and making sure they are finished.

A more flexible resource management system, better scalability, and more flexibility are just a few of the significant advancements that MRv2 provides over MRv1.

++++++

MAP REDUCE RECOVERY FAILURE

With the help of the MapReduce programming model, programmers can handle sizable datasets in a distributed setting. Data processing jobs are split into the Map and Reduce phases in MapReduce.

The input data is broken up into smaller pieces and processed concurrently by several cluster nodes during the Map step. A group of intermediate key-value pairs is the result of the Map step.

The intermediate key-value combinations created by the Map phase are combined, sorted, and processed to create the final result during the Reduce phase.

Failures in a distributed setting are unavoidable. Nodes may fail for a variety of causes, including network issues, hardware or software problems, etc. A strong failure recovery mechanism is necessary to guarantee that the MapReduce job is successfully finished.

Here are some methods for recovering from failures in MapReduce:

1. job retries: The MapReduce framework can attempt a failed Map or Reduce job again on a different node. The job can be repeated a configurable number of times before permanently succeeding or failing.

2. Speculative Execution: The framework can start a duplicate task on a different node when a Map or Reduce job is taking longer than anticipated. The task that is effectively completed first is used, and the other is eliminated. By utilising idle resources, this method can cut down on execution time altogether.

3. Launching backup tasks concurrently with the main tasks is an option. Unless the primary job fails, these tasks are not used. The backup task assumes control in the event of a primary task failure to guarantee work completion.

4. Checkpointing: Interim outcomes may be routinely stored to a distributed file system, like HDFS. To minimise the amount of work that needs to be redone after a task fails, the framework can resume the task from the most recent checkpoint.

5. Task Monitoring: The MapReduce framework can keep track of each task's progress and rapidly identify failures. The framework can record the error message and alert the administrator when a job fails.

These methods allow MapReduce to manage errors in a distributed setting and guarantee the effective completion of massive data processing tasks.

APACHE HIVE ARCHITECTURE

A data warehousing utility called Apache Hive offers a SQL-like query language (HiveQL) for searching and analysing huge datasets kept in Hadoop. Hive's design is made up of the following elements:

1. User Interface: Hive offers a variety of user platforms through which users can communicate with the system. HiveQL searches are run through the command-line interface (CLI), which is also used to manage the system. A web-based graphical user interface (GUI) for working with Hive is offered by the Hive Web Interface.

2. Driver: The Driver is in charge of controlling a HiveQL query's lifetime. It takes a question from the user interface, turns it into an execution plan, and then uses the cluster to carry out the plan.

3. Compiler: The Compiler is in charge of creating a processing plan from the HiveQL query. A physical plan that can be executed on the cluster is created from a logical plan that the compiler produces.

4. Metastore: The Metastore is the main location where information about the data kept in Hive is kept. It keeps track of tables, partitions, columns, and the data kinds, locations, and formats that go along with them.

5. Execution Engine: The Execution Engine is in charge of carrying out the actual plan that the Compiler produced. Numerous processing engines, such as MapReduce, Tez, and Spark, are supported by Hive.

6. SerDe: The SerDe (Serializer/Deserializer) transforms data saved in Hive into serialised and deserialized forms. Between its internal representation and the format needed by the storage device, it converts the data.

7. Storage Handler: The Storage Handler is in charge of communicating with the data storage device. Various storage drivers, such as HDFS, HBase, and Amazon S3, are supported by Hive.

8. Hadoop Distributed File System (HDFS): HDFS serves as Hadoop's main file system. It disperses the data across a number of cluster servers for storage.

9. Apache ZooKeeper: Used in Hadoop to manage configuration data, synchronization, and naming services, ZooKeeper is a distributed coordination tool. For global locking and coordination, Hive makes use of it.

Hive architecture is created to offer a scalable and adaptable data warehousing option that can be used to process sizable datasets kept in Hadoop. Users can adapt the system to their needs by using a variety of execution engines and storage handlers thanks to the design.

APACHE SPARK

An open-source large data processing engine with a focus on speed, scalability, and user-friendliness is called Apache Spark. For processing batch and streaming data, machine learning, graph processing, and interactive queries, Spark offers a unified platform.

The distributed computing paradigm serves as the foundation for the architecture of Spark, which uses a cluster of nodes to handle data in parallel. The following are the principal Spark design elements:

1. Driver Program: The Spark application is controlled by the Driver Program, which is the primary application. It executes the user's code and manages the cluster's distributed processing.

2. Executors: Worker programmes known as executors are active on every cluster node. They are in charge of carrying out the tasks given to them by the driving programme and controlling the memory and resources that the tasks consume.

3. Cluster Manager: The Cluster Manager is in charge of controlling the cluster's network bandwidth, RAM, and CPU usage. In addition to Hadoop YARN, Apache Mesos, and Spark's own standalone cluster manager, Spark also accepts other cluster managers.

4. Resilient Distributed Datasets (RDDs): RDDs are the cornerstone of the Spark data model. They are immutable, partitioned data sets that may be handled concurrently by the cluster.

5. Spark SQL: Spark SQL offers a querying tool for structured data that is similar to SQL. It enables the querying of data held in a number of forms, including Apache HBase, Hadoop Distributed File System, and Apache Cassandra.

6. Spark Streaming: A flexible and fault-tolerant processing engine for streaming data is Spark Streaming. It offers assistance for ingesting and analysing real-time data streams, including web logs, sensor data, and social media feeds.

7. MLlib: Spark's machine learning framework is called MLlib. For the development of scalable machine learning algorithms and models, it offers a collection of high-level APIs.

8. GraphX: Spark's network processing library is called GraphX. It offers a collection of APIs for constructing and working with graphs, including social networks, transportation networks, and biological networks.

Overall, the architecture of Spark is created to offer an adaptable and scalable foundation for handling sizable datasets and carrying out intricate computations on them. Users can pick and choose the components they require for their unique use cases thanks to its modular design, and the development and deployment of big data applications is made easier by its unified programming interface.

Difference between HADOOP AND SPARK

Big data processing frameworks that can manage large amounts of data include Hadoop and Spark. They vary significantly in terms of their architecture, performance, and use cases, though.

1. Architecture: While Spark does not have its own file system and can operate on top of other distributed file systems like HDFS, Amazon S3, or Apache Cassandra, Hadoop uses a distributed file system called Hadoop Distributed File System (HDFS) for storing and managing data. Additionally, Spark uses an in-memory processing engine, whereas Hadoop uses a two-stage MapReduce processing paradigm.

2. Performance: Spark performs better than Hadoop in general, especially when handling big datasets. This is because Hadoop must read and write data to disc between each step of processing,

whereas Spark can perform many computations in memory thanks to its in-memory processing engine.

3. Use Cases: While Spark is better suited for real-time stream processing, machine learning, and interactive data analysis, Hadoop is more commonly used for batch processing and storing vast amounts of data. Furthermore, Hadoop is frequently used in data warehousing and ETL (Extract, Transform, Load) situations, whereas Spark is frequently used in situations that call for quicker processing and more complicated calculations.

4. Ecosystem: Because Hadoop has been around longer and has a more established group, it has a larger ecosystem than Spark. There are many various tools and technologies in this ecosystem that can be used with Hadoop, including Apache Pig, Apache Hive, and Apache Zookeeper. But Spark is quickly gaining acceptance, and its community of tools and libraries—which already includes GraphX, MLlib, and Spark SQL—is expanding.

In conclusion, there are some key distinctions between Hadoop and Spark in terms of design, performance, and use cases. While Spark is better suited for real-time stream processing, machine learning, and interactive data analysis, Hadoop is more commonly used for batch processing and data storage.

SIMILARITIES OF HADOOP AND SPARK

There are several similarities between Hadoop and Spark as large data processing frameworks:

1. Distributed computing: Both Hadoop and Spark are built for distributed computing, allowing for the parallel processing of big datasets across a cluster of nodes.

2. Open Source: Since Hadoop and Spark are both open-source projects, users can freely access and alter them to suit their requirements.

3. Horizontal scaling is possible with both Hadoop and Spark by adding additional nodes to the network as required.

4. Fault Tolerance: Hadoop and Spark are both made to be fault-tolerant, which means they can recover from hardware problems or other kinds of system failures without losing any data.

5. Batch Processing: Spark and Hadoop are both capable of being used for batch processing, which entails handling massive amounts of data in a scheduled or regular way.

6. Ecosystem: To perform a variety of big data processing jobs, both Hadoop and Spark have rich ecosystems of tools, libraries, and technologies.

In conclusion, there are many design, functional, and ecosystem similarities between Hadoop and Spark that make them both ideal for handling big data processing jobs in a distributed computing environment.

COLUMNAR DATABASE

A database management system (DBMS) that keeps data in columns rather than rows is referred to as a columnar database, also referred to as a column-oriented database or a columnar store. Each column of data is stored separately in a columnar database, and each column may have a distinct data type.

For analytical workloads, particularly those involving complex queries or the aggregation of sizable datasets, columnar databases are built to maximise efficiency and speed. The ability of columnar databases to access only the specific columns required by a given query reduces the amount of data that must be processed and can greatly enhance query performance because each column is stored separately.

The fact that columnar databases can condense data more successfully than row-oriented databases is another benefit. Since there is only one sort of data in each column, it is simpler to compress and store that data more effectively. By reducing the amount of data that needs to be read from disk, this compression can result in significant storage cost savings and enhance query speed.

Where high performance and scalability are essential, columnar databases are frequently used in data warehousing, business intelligence, and analytics apps. Apache Cassandra, Apache HBase, Google Bigtable, Amazon Redshift, and Apache Kudu are a few instances of columnar databases.

In a columnar database, the tables are organised by row. Benefits include quick execution of tabular operations like MIN, MAX, SUM, COUNT, and AVG, as well as efficient write and read operations of data to and from hard disc storage. Columnar databases satisfy the ACID requirements for a database and enable random read and write in Hadoop.

APACHE PIG

Pig Latin is the name of the language used to analyse data in Hadoop using Pig. It is a high-level data processing language that offers a wide variety of operators and data types to handle the data in different ways.

Programmers must create a Pig script in the Pig Latin language and execute it using one of the execution mechanisms in order to complete a specific job when using Pig. (Grunt Shell, UDFs, Embedded). Following execution, the Pig Framework will apply a number of transformations to these scripts in order to create the intended output.

Internally, Apache Pig transforms these scripts into a string of MapReduce tasks, simplifying the work of the coder.

The Apache Pig architecture is made up of different parts.

Parser

The Parser initially handles the Pig Scripts. It performs type checking, examines the script's syntax, and performs various other checks. A DAG (directed acyclic graph), which depicts the logical operators and statements in Pig Latin, will be the parser's output.

The script's logical operators are depicted in the DAG as nodes, and the data processes are depicted as edges.

Optimizer The logical optimizer receives the logical plan (DAG) and performs logical optimisations like projection and pushdown.

Compiler The optimised logical plan is converted into a sequence of MapReduce jobs by the compiler.

Engine of execution

The MapReduce tasks are then sorted before being sent to Hadoop. When these MapReduce jobs are finally run on Hadoop, the intended outcomes are obtained.

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'
```