**Stats:**

1. Each day, *Google* processes **8.5 billion** searches.
2. *WhatsApp* users exchange up to **65 billion** messages daily.
3. The world will produce slightly over **180 zettabytes** of data by *2025*
4. **80-90%** of the data we generate today is unstructured.

**Case Study:**

Volume: The amount of data generated by social media platforms can reach several petabytes, requiring large-scale storage solutions with capacity in the range of hundreds of terabytes.

Velocity: Social media data is generated in real-time, with new data being generated constantly at a rate of several gigabytes per second.

Variety: Social media data can come in various forms such as text, images, videos, and numerical data, requiring a data storage solution that can handle multiple data types.

Veracity: The accuracy of social media data is crucial, with a requirement for data cleansing and validation processes to maintain a minimum error rate of less than 0.5%.

Value: Social media sentiment analysis can help companies understand public opinion, improve brand reputation, and increase customer engagement, leading to increased revenue of up to 10%.

Visualization: Presenting social media sentiment analysis results in a clear and intuitive manner is crucial for effective decision-making, requiring advanced visualization techniques.

Viscosity: The complexity of social media sentiment analysis can make it challenging to analyze data in real-time, requiring advanced algorithms and data processing techniques.

Virality: Social media sentiment can spread rapidly, requiring flexible analysis techniques that can adapt to changes in sentiment within hours or minutes.

**HDFS is composed of master-slave architecture, which includes the following elements:**

**Name Node**

* Name Node is controller and manager of HDFS
* It knows the status and the metadata of all the files in HDFS
* Metadata [ file names, permissions, and locations of each block of file ]
* HDFS cluster can be accessed concurrently by multiple clients, even then this metadata information
* is never desynchronized; hence, all this information is handled by a single machine
* Since metadata is typically small, all this info is stored in main memory of Name Node, allowing fast
* access to metadata

**Data Node**

* Actual data is stored on the Data Node
* Knows only about the data stored on it
* Will read data and send to client when retrieval requested
* Will receive data and store locally when storage is requested

**Name Node HA**

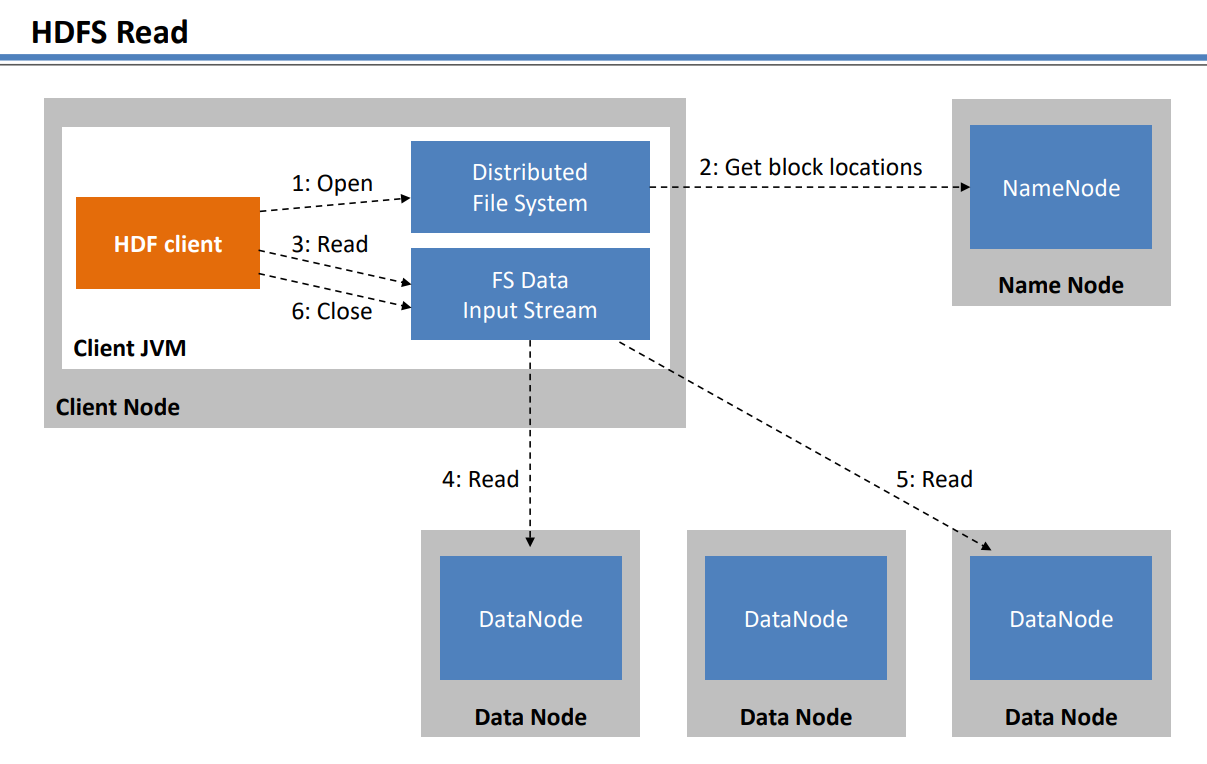
* Function of name node is very critical for overall health of HDFS
* If individual data nodes fail, HDFS can recover and function with a little less capacity
* Crash of name node can lose all the information and the complete file system irrecoverably
* That's why metadata and involvement of name node in data transfer is kept minimal
* Name Node can also be set to work in HA

**Secondary Name Node**

* It is not backup of name node nor data nodes connect to this; it is just a helper of name node.
* It only performs periodic checkpoints
* It communicates with name node and to take snapshots of HDFS metadata
* These snapshots help minimize downtime and loss of data

**Features of HDFS:**

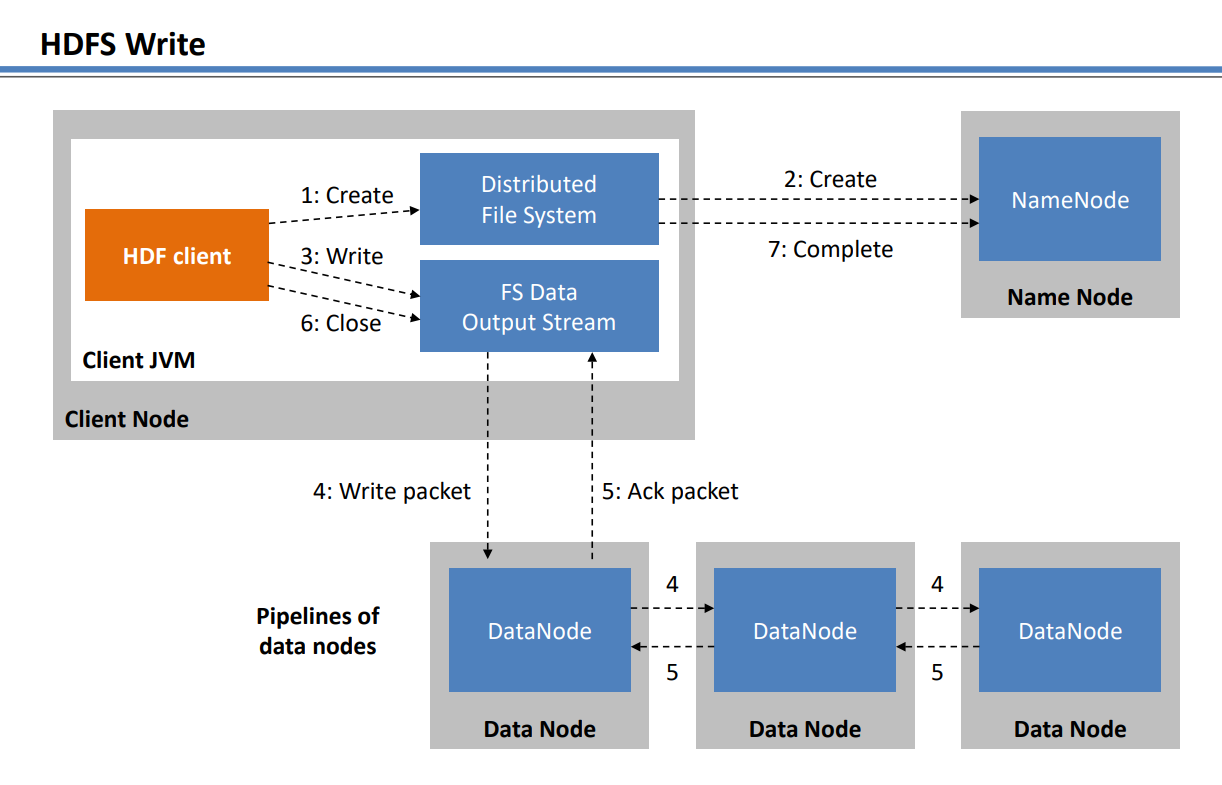
* Data is distributed over several machines
* Replicated to ensure their durability to failure & high availability to parallel applications
* Designed for very large files (in GBs, TBs)
* Block oriented
* Unix like commands interface
* Write once and read many times
* Commodity hardware
* Fault Tolerant when nodes fail
* Scalable by adding new nodes



In Hadoop Distributed File System (HDFS), reading a file works as follows:

* The client requests to read a file from HDFS.
* The NameNode, which is the master node in HDFS, determines which blocks of the file are stored on which DataNodes.
* The client then sends requests to the appropriate DataNodes to retrieve the blocks of the file.
* Each DataNode sends the requested block to the client.
* The client assembles the blocks into the original file and returns the file to the user.

This approach allows HDFS to scale horizontally by distributing data across multiple nodes, while also providing fault tolerance through replication of data blocks.



In HDFS (Hadoop Distributed File System), a write operation works as follows:

* Client application sends a write request to the NameNode (the master node in HDFS cluster that manages the metadata of the file system).
* NameNode checks if the requested block size is within the limit, and if there is enough available storage in the DataNodes (the worker nodes in HDFS cluster that store the actual data).
* NameNode allocates a block for the write request and selects three DataNodes to replicate the block.
* Client writes the data to the first DataNode, which stores the block and forwards it to the next two DataNodes for replication.
* The first DataNode acknowledges the successful write to the Client.
* The second and third DataNodes also acknowledge the successful replication to the first DataNode.
* The NameNode updates its metadata to reflect the new block information and replication factor.
* The Client can then write more data, repeating the above steps until the file is complete.

Note that HDFS replication provides reliability and fault tolerance by storing multiple copies of each block on different DataNodes.

**Some HDFS commands:**

hdfs dfs -mkdir

hdfs dfs -put

▪ hdfs dfs -ls

▪ hdfs dfs -cp

▪ hdfs dfs -mv

▪ hdfs dfs -rm

▪ hdfs dfs -du

▪ hdfs dfs -cat

▪ hdfs dfs -tail

▪ hdfs dfs -chmod

▪ hdfs dfs -help

▪ yarn version

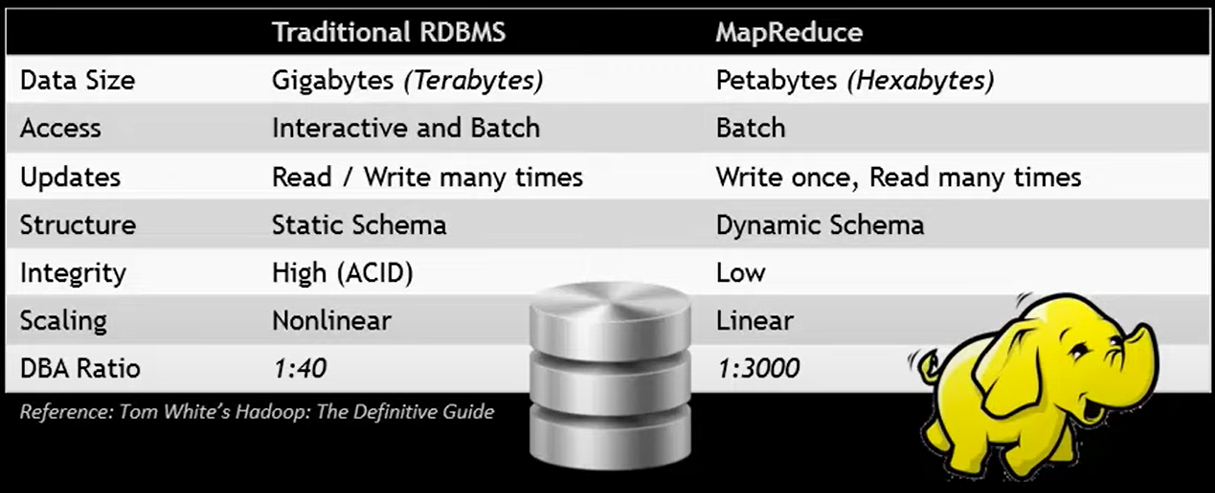
▪ hdfs version

▪ hadoop version (deprecated)

*From Hadoop*

* dfs run a filesystem command on the file systems supported in Hadoop.
* namenode -format format the DFS filesystem
* secondarynamenode run the DFS secondary namenode
* namenode run the DFS namenode
* journalnode run the DFS journalnode
* zkfc run the ZK Failover Controller daemon
* datanode run a DFS datanode
* dfsadmin run a DFS admin client
* diskbalancer Distributes data evenly among disks on a given node
* haadmin run a DFS HA admin client
* fsck run a DFS filesystem checking utility
* balancer run a cluster balancing utility
* jmxget get JMX exported values from NameNode or DataNode.
* mover run a utility to move block replicas across storage types
* oiv apply the offline fsimage viewer to an fsimage
* oiv\_legacy apply the offline fsimage viewer to an legacy fsimage
* oev apply the offline edits viewer to an edits file
* fetchdt fetch a delegation token from the NameNode
* getconf get config values from configuration
* groups get the groups which users belong to
* snapshotDiff diff two snapshots of a directory or diff the current directory contents with a snapshot
* lsSnapshottableDir list all snapshottable dirs owned by the current user Use -help to see options
* portmap run a portmap service
* nfs3 run an NFS version 3 gateway
* cacheadmin configure the HDFS cache
* crypto configure HDFS encryption zones
* storagepolicies list/get/set block storage policies
* version print the version

**Map-Reduce Steps**



▪ Split input data in independent chunks is already available via HDFS

▪ Job needs to be scheduled to carry out required process

▪ Schedule tasks on nodes where data is already present

▪ Map Phase – Transformation Phase

=> input Data | output – list <key, value> pairs

▪ Sort & Shuffle – Group & Order Phase

=> input – list of <key , value> pairs | output – sorted & grouped list of <key, value> pairs

▪ Reduce Phase – Aggregation Phase

=> sorted & grouped list of <key , value> pairs | output – aggregated <key, value>

Observations

▪ Manager required for scheduling jobs, collating results & updating status

▪ Process and storage nodes are same. i.e., MR-Tasks and HDFS-DataNode run on same machine.

MR-V1 PROCESS

• A Client invokes a Map Reduce, from a Calling Node (maybe a Data Node or an Extra Node in the cluster)

• An instance of Job Tracker is created in the memory of the Calling Node

• The Job Tracker queries the Name Node and finds the Data Nodes (location of the data to be used)

• Job Tracker then launches Task Trackers in the memory of all the Data Nodes as above to run the jobs

• Job Tracker gives the code to Task Tracker to run as a Task

• Task Tracker is responsible for creating the tasks & running the tasks

• In effect the Mapper of the Job is found here

• Once the Task is completed, the result from the Tasks is sent back to the Job Tracker

• Job Tracker also keeps a track of progress by each Task Tracker

• The Job Tracker also receives the results from each Task Tracker and aggregates the results

• In effect the Reducer of the Job is found here

MR-V2 PROCESS

• A Client invokes a Map Reduce, from a Calling Node (maybe a Data Node or an Extra Node in the cluster)

• An instance of Resource Manager is created in the memory of the Calling Node

• The Resource Manager then launches containers with appropriate resources (memory) with App Node Manager in memory of the Calling Node

• Along with this Application Master is invoked. Application Master is “pause” mode till all containers

• With Task Node Manager (as below) are created

• The Resource Manager queries the Name Node and finds the Data Nodes (location of the data used)

• The Resource Manager then launches containers with appropriate resources (memory) with Task Node Manager in all the Data Nodes as above to run the jobs

• Application Master gives the code to Task Node Manager to run as a Task

• Task Node Manager is responsible for creating & running tasks. In effect the Mapper of the Job is here

• Once the Task is completed, the result from the Tasks is sent back to the Application Master

• Application Master also keeps a track of progress by each Task Node Manager

• The Application Master also receives the results from each Task Node Manager and aggregates the results

• In effect the Reducer of the Job is found here

• Thus, from previous version, Job Tracker has been replaced by Resource Manager & Application Master

• From previous version, Task Tracker has been replaced by Task Node Managers

MAP REDUCE FAILURE RECOVERY

MRv1

• Task Failure new task is started by the Task Tracker

• Task Tracker Failure new Task Tracker is started by the Job Tracker

• Job Tracker Failure no recovery; single point of failure

MRv2

• Task Failure new task is started by Task Node Manager

• Task Node Manager Failure new container with Task Node Manager is created by Resource Manager this Task Node Manager is given the code and started by Application Master

• Application Master Failure New Application Master is started by App Node Manager

• App Node Manager Failure new container with App Node Manager is created by Resource Manager this App Node Manager invokes the Application Master

• Resource Manager Failure new resource manager with saved state is started

**Group by, union, split**

* grunt> Group\_data = GROUP Relation\_name BY age;
* Given below is the syntax of the UNION operator.
* grunt> SPLIT Relation1\_name INTO Relation2\_name IF (condition1), Relation2\_name (condition2),