

Big Data

Unit 1

Course Objectives



Develop

Develop a comprehensive understanding of big data technologies, tools, and techniques for processing and analysing large-scale datasets.



Acquire

Acquire knowledge and skills in utilizing advanced analytics methods tailored for big data to extract meaningful insights.



Learn

Learn to apply big data analytics techniques to address real-world challenges and opportunities across diverse domains.



Gain

Gain proficiency in utilizing big data platforms and tools such as Hadoop, Spark, and NoSQL databases for efficient data processing and analysis.



Equip

Equip students with the capability to leverage big data analytics to make informed decisions and drive innovation in various professional settings.

Course Outcomes

Upon successful completion of the course, students should be able to

Understand	the concepts, characteristics, and challenges of big data, including volume, velocity, variety, and veracity.
Apply	big data processing frameworks, such as Hadoop and Spark, to manage and analyze large-scale datasets efficiently.
Design and implement	data pre-processing techniques for cleaning, transforming, and preparing big data for analysis.
Apply advanced	analytics methods including machine learning, data mining, and predictive modelling, to extract insights and patterns from big data.
Develop and deploy	scalable big data analytics solutions using cloud-based platforms and distributed computing environments.
Evaluate	the performance, scalability, and reliability of big data analytics systems in real world scenarios.

Syllabus Outline



Unit 1: INTRODUCTION TO BIG DATA ANALYTICS

Overview of big data: characteristics, sources, and challenges, Introduction to big data analytics tools and technologies, Big data storage and processing frameworks: Hadoop, Spark, and Flink, Ethical and legal considerations in big data analytics.



Unit 2: DATA PREPROCESSING AND CLEANING

Techniques for data cleaning, transformation, and integration, Handling missing data and outliers in big data, Data reduction and feature selection methods, Scalable data preprocessing workflows using Apache Pig and Apache Hive.



Unit 3: BIG DATA PROCESSING AND ANALYTICS

Map Reduce programming model for distributed data processing, Introduction to Apache Spark: RDDs, Data Frames, and Spark SQL, Machine learning algorithms for big data analytics: classification, regression, clustering, Real-time analytics and stream processing with Apache Flink.



Unit 4: ADVANCED ANALYTICS ON BIG DATA

Deep learning for big data analytics: neural networks, convolutional neural networks (CNNs), recurrent neural networks (RNNs). Text analytics and natural language processing (NLP) on big data, Graph analytics for social network analysis and recommendation systems, Ensemble learning methods and model evaluation techniques.



Unit 5: SCALABLE BIG DATA ANALYTICS PLATFORMS

Cloud-based big data analytics platforms: Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP), Containerization and orchestration tools: Docker, Kubernetes, Big data deployment and scaling strategies, Case studies and applications of big data analytics in various domains

Unit 1 : INTRODUCTION TO BIG DATA ANALYTICS



What is Big data?



Big Data refers to the large and complex datasets that traditional data processing software and methods are unable to handle efficiently.



These datasets come from various sources and grow rapidly, necessitating advanced technologies and techniques to store, manage, and analyze them.



Big Data encompasses not just the volume of data but also the velocity at which it is generated, the variety of data types, and the veracity or quality of the data.

Characteristics of Big Data

Volume: Refers to the vast amounts of data generated every second. Big data often involves terabytes or even petabytes of information.

Velocity: The speed at which data is generated, collected, and analyzed. This includes real-time data streams and the need for quick processing.

Variety: The different types of data, including structured, semi-structured, and unstructured data from various sources like text, images, video, and more.

Veracity: The accuracy and reliability of the data. Big data comes with a lot of noise and uncertainty, requiring robust methods for data validation.

Value: The potential insights and benefits that can be derived from the data. The true value lies in the ability to turn vast amounts of data into actionable insights.

Sources of Big Data



Social Media: Platforms like Facebook, Twitter, Instagram, and LinkedIn generate vast amounts of user-generated content.



IoT Devices: Sensors, smart devices, and connected machines produce continuous data streams.



Transactional Data: Data generated from business transactions, including online purchases, financial transactions, and retail sales.



Healthcare Data: Electronic health records, medical imaging, and genomic data.



Government and Public Data: Census data, public records, and other government-held datasets.



Multimedia: Images, videos, and audio files generated from various sources such as surveillance systems, entertainment, and personal devices.



Clickstream Data: Data generated from users' interactions with websites, including page views, clicks, and navigation paths.

Challenges of Big Data



Storage and Management: Storing large volumes of data requires scalable solutions like cloud storage and distributed databases.



Data Integration: Combining data from diverse sources and formats into a unified dataset.



Quality Assurance: Ensuring data accuracy, consistency, and completeness while managing issues like duplication and missing values.



Privacy and Security: Protecting sensitive data and ensuring compliance with privacy regulations.



Processing and Analysis: Using advanced algorithms and tools to process and analyze large datasets, often requiring parallel processing and distributed computing.

Challenges of Big Data



Scalability: Ensuring systems can handle growing data volumes and complexity.



Real-Time Processing: Developing systems capable of analyzing data as it is generated to provide immediate insights.



Visualization: Creating effective visual representations to make the data comprehensible and actionable.



Cost Management: Balancing the costs of storage, processing, and analysis with the value derived from the data.

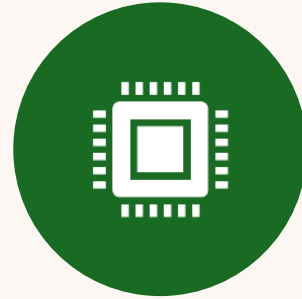


Skill Gaps: Finding skilled professionals who can work with big data technologies and methodologies.

Big Data Analytics



BIG DATA ANALYTICS INVOLVES EXAMINING LARGE AND VARIED DATASETS TO UNCOVER HIDDEN PATTERNS, UNKNOWN CORRELATIONS, MARKET TRENDS, CUSTOMER PREFERENCES, AND OTHER USEFUL BUSINESS INFORMATION.



THIS COMPLEX PROCESS REQUIRES SPECIALIZED TOOLS AND TECHNOLOGIES TO MANAGE AND ANALYSE THE MASSIVE AMOUNT OF DATA EFFICIENTLY.

Key Technologies and Tools for Big Data Analytics

Key Considerations for Choosing Big Data Tools



Scalability: The ability to handle increasing data volume and complexity.



Flexibility: Compatibility with various data types and sources.



Ease of Use: User-friendly interfaces and strong community support.



Integration: Seamless integration with existing systems and tools.



Cost: Balancing between the tool's capabilities and the budget.



Performance: Efficient processing and real-time data analytics capabilities.



Security: Robust security features to protect sensitive data.

Hadoop Ecosystem

Hadoop Distributed File System (HDFS): A scalable and reliable storage system for large datasets.

MapReduce: A programming model for processing large datasets with a distributed algorithm on a Hadoop cluster.

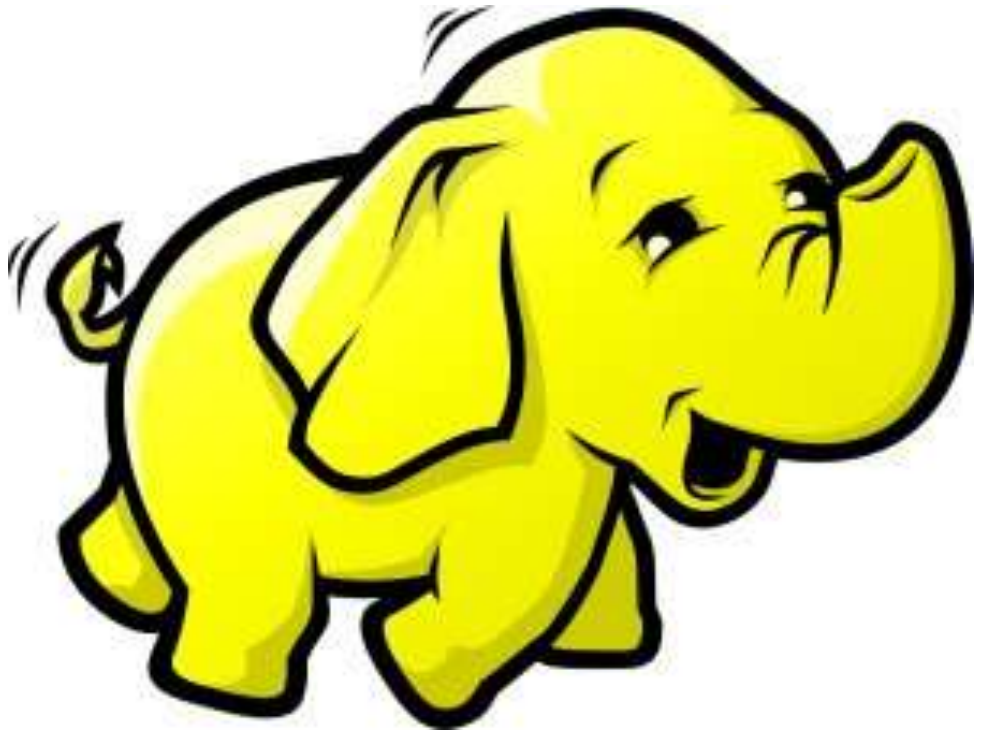
YARN (Yet Another Resource Negotiator): Manages resources in Hadoop and provides a framework for job scheduling and cluster resource management.

HBase: A distributed, scalable, big data store, modeled after Google's Bigtable.

Hive: A data warehouse infrastructure built on top of Hadoop for providing data summarization, query, and analysis.

Pig: A high-level platform for creating MapReduce programs used with Hadoop.

Say
hello
to
Hadoop



HADOOP

- Hadoop is an open-source framework used for storing and processing large datasets in a distributed computing environment. It is designed to scale up from a single server to thousands of machines, each offering local computation and storage.



What's all the excitement about?

Big Data

Analytics

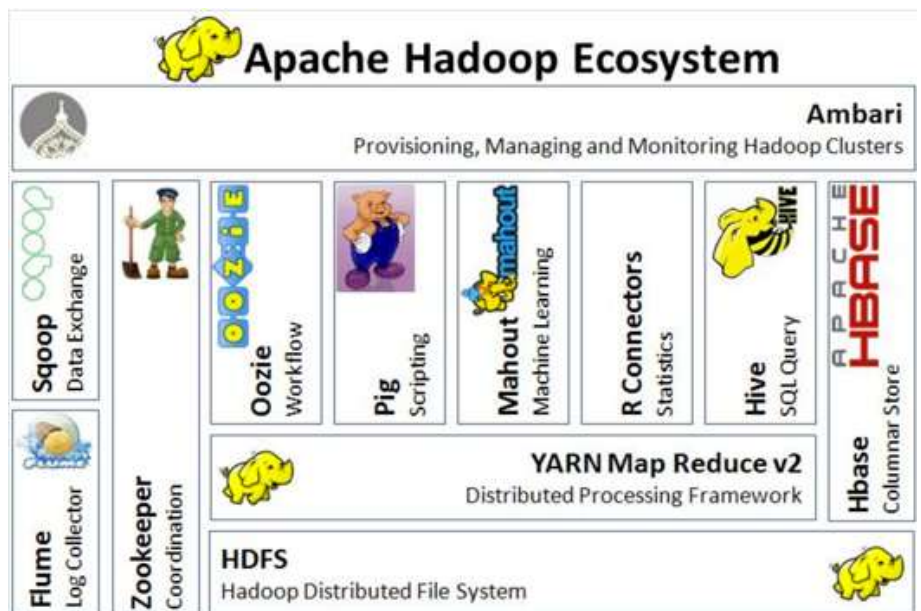
- Data science: Develop insights from Big Data
 - Often statistical, often centered around Clustering and Machine Learning
- Big Brother Is Watching You (BBIWY)
 - Governments? Corporations? Criminals?
 - Somebody else's presentation topic!
 - Bruce Schneier, et al

Cloud Computing

- Public: Amazon, et al, low-cost prototyping for startup companies
 - Elastic Compute Cloud (EC3)
 - Simple Storage Service (S3)
 - EBS
 - Hadoop services
- Private

Grid or Cluster computing

Say hello
to Hadoop's
friends
a.k.a. The
Hadoop
Ecosystem



Overloading Terminology



Hadoop has become synonymous with Big Data management and processing



The name Hadoop is also now a proxy for both Hadoop and the large, growing ecosystem around it



Basically, a very large “system” using Hadoop Distributed File System (HDFS) for storage, and direct or indirect use of the MapReduce programming model and software framework for processing

Hadoop: The High Level

Apache top-level project

- develops open-source software for reliable, scalable, distributed computing.”

Software library

- framework that allows for distributed processing of large data sets across clusters of computers using a simple programming model designed to scale up from single servers to thousands of machines, each offering local computation and storage designed to detect and handle failures at the application layer... delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.”

Hadoop Distributed File System (HDFS)

- primary storage system used by Hadoop applications. HDFS creates multiple replicas of data blocks and distributes them on compute nodes throughout a cluster to enable reliable, extremely rapid computations.

MapReduce

- a programming model and software framework for writing applications that rapidly process vast amounts of data in parallel on large clusters of compute nodes.

What's Hadoop Used For?

Major Use Cases per Cloudera

Cloudera, a company providing enterprise data cloud solutions, identifies several key use cases for Hadoop, including:



Data Warehousing:

Organizations use Hadoop to store and analyze large volumes of structured and unstructured data.



Data Lake:

Hadoop can act as a centralized repository for storing raw data, which can be processed and analyzed later as needed.



Batch Processing:

Hadoop excels in processing large datasets in batches, which is useful for tasks like log processing, ETL (Extract, Transform, Load) operations, and more.



Real-time Processing:

While traditionally used for batch processing, Hadoop components like Apache Storm and Spark enable real-time data processing and analytics.



Machine Learning:

Hadoop's scalable nature makes it a suitable platform for machine learning tasks, including training models on large datasets.



Data Archiving:

Hadoop is also used for long-term data storage, ensuring that historical data is preserved and can be accessed when needed.

HDFS

+



○



Google File System (GFS) as the Original Inspiration:

The Hadoop Distributed File System (HDFS) was inspired by the Google File System (GFS). GFS introduced innovative ideas for handling large-scale data, which laid the groundwork for HDFS's design and functionality.



Common Attributes Between GFS and HDFS:

Both GFS and HDFS share several key characteristics, such as their distributed nature and their ability to handle large-scale data efficiently.



Pattern Emphasis: Write Once, Read Many:

Both systems are designed with the "Write Once, Read Many" (WORM) pattern in mind. This approach is well-suited for large datasets that are written once and then read many times, reducing the need for complex data management.



Historical Reference - WORM Drives and Optical Jukebox Libraries:

The WORM pattern is reminiscent of older technologies like WORM (Write Once, Read Many) drives and optical jukebox libraries, which were used before the advent of CDs (CD-R). These technologies allowed data to be written once and read multiple times without alteration.

HDFS



Evolution of Storage Technology:

During the era when these technologies were prevalent, WORM optical storage offered a better price-to-performance ratio for larger storage volumes compared to disk technology of that time. Even though disk technology was more common, WORM was still a cost-effective solution, especially when compared to tape storage, which was another popular medium.



Handling Very Large Files:

One of the significant capabilities of HDFS is its ability to manage very large files, with a single file potentially spanning the entire distributed system. This allows for the efficient storage and processing of massive datasets.



Use of Commodity Hardware:

HDFS is designed to run on commodity hardware, which typically includes:

- **Two-socket, 64-bit servers:** Basic server configurations that are affordable and widely available.
- **Local disks:** Instead of relying on specialized storage systems like RAID, HDFS uses local disks on each server.
- **No RAID:** HDFS avoids the complexity and cost of RAID configurations by relying on data replication across multiple nodes for fault tolerance.



Co-location of Data and Compute Resources:

HDFS optimizes performance by co-locating data with compute resources. This means that the data is stored on the same machines that perform computations, reducing the need to transfer large amounts of data over the network and improving overall processing speed.



What It's Not Good For

Handling Many Small Files:	The Name Node may face scalability challenges when dealing with a large number of small files. More details to follow.
Low Latency Access:	The focus here is on maximizing throughput ($N=XR$) rather than minimizing the time it takes to start reading the data (latency).
Multiple Writers & File Updates:	The system only allows one writer at a time. Operations are limited to creating, appending, renaming, moving, and deleting files. There are no in-place updates.
Not a Replacement for Relational Databases:	Data is stored in files without indexing. To locate specific information, all data must be read, usually through a MapReduce job or task.
Unsuitable for Traditional Storage Solutions:	Traditional SAN or NAS systems, like those from NetApp and EMC, are not applicable here.
Outdated Technologies:	This is analogous to programming in COBOL or selling mainframes and FICON technologies, which are considered outdated.



HDFS Concepts & Architecture

Core architectural goal: Fault-tolerance in the face of massive parallelism (many devices, high probability of HW failure)

Monitoring, detection, fast automated recovery

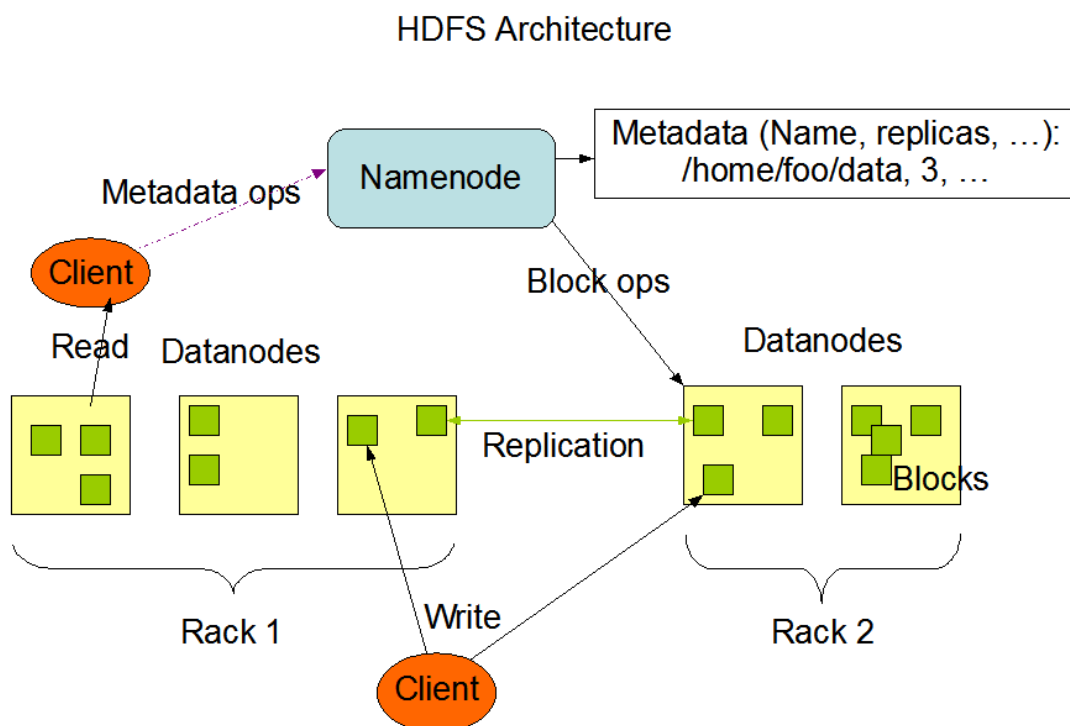
Focus is batch throughput

- Hence support for very large files; large block size; single writer, create or append (no update) and a high read ratio

Master/Slave: Separate filesystem metadata & app data

- One NameNode manages the filesystem and client access
- All metadata - namespace tree & map of file blocks to DataNodes - stored in memory and persisted on disk
- Namespace is a traditional hierarchical file organization (directory tree)
- DataNodes manage locally attached storage (HDD, no RAID)

Not layered on or dependent on any other filesystem



HDFS Architecture

Namenode:

- The **Namenode** is the master server that manages the metadata of the file system. It keeps track of:
 - The filenames
 - The directory structure
 - The locations of the file blocks across the Datanodes
 - The replication factor (how many copies of each block are stored)
- The Namenode does not store the actual data but only the metadata. This includes information such as file paths, permissions, and the mapping of files to blocks.

Datanodes:

- Datanodes** are the worker nodes that store the actual data blocks of the files.
- Data is split into blocks (typically 128 MB or 256 MB in size) and distributed across multiple Datanodes.
- The Datanodes are responsible for serving read and write requests from clients, and they also handle block creation, deletion, and replication as instructed by the Namenode.

Client:

- The **Client** interacts with the HDFS by submitting requests to read or write data.
- When a client wants to read a file, it contacts the Namenode to get the metadata (e.g., the locations of the blocks) and then reads the data directly from the Datanodes.
- When writing data, the client first asks the Namenode where to write the blocks, and then writes the data directly to the Datanodes.

HDFS Architecture

Replication:

- HDFS is designed for high fault tolerance. To achieve this, each block of data is replicated across multiple Datanodes (typically three copies, but this can be configured).
- If a Datanode fails, the system will replicate the blocks on the remaining nodes to maintain the required replication factor.
- The diagram shows that data is written by the client to a Datanode, and then replicated to other Datanodes across different racks to ensure availability even if a rack fails.

Racks:

- In the diagram, the Datanodes are organized into **Racks** (Rack 1 and Rack 2). A rack is a collection of physically co-located Datanodes.
- HDFS is aware of the rack topology and places replicas in such a way that at least one replica is on a different rack, providing greater fault tolerance.

Metadata Operations (Metadata ops):

- When a client interacts with HDFS, it first communicates with the Namenode to get the necessary metadata for the operation (e.g., file location, block locations).
- These operations include checking the location of file blocks and managing the replication of data.

Block Operations (Block ops):

- These are the actual operations performed on the data blocks stored in the Datanodes, such as reading or writing blocks.

HDFS Concepts & Architecture



Block-Oriented Structure:	HDFS (Hadoop Distributed File System) uses a block-oriented storage system where the default block size is 64 MB. The block size can be adjusted on a per-file basis.
Impact of Smaller Block Sizes:	If the block size were significantly smaller, say 512 bytes or 4 KB instead of 64 MB, the metadata required to track these blocks would increase dramatically—by 128,000 or 16,000 times, respectively. This would put immense pressure on the Namednode's memory, possibly limiting the total HDFS storage capacity.
File Size Sensitivity:	The total storage capacity of HDFS is not just affected by block size but also by the total number of files stored in the system.
Block Size Consistency:	All blocks of a file are of the same size, except for the last block, which might be smaller.
Performance Optimization:	The large block size in HDFS reduces the time spent seeking data on the disk. This approach allows the system to achieve disk transfer speeds of up to 100 MB/sec, with disk read-ahead further reducing seek times. The aggregate file read bandwidth can be high, depending on the number of disks per node.
Fault Tolerance through Replication:	HDFS divides files into blocks and stores them across multiple DataNodes with redundancy. The replication factor, which determines the number of copies of each block, is set on a per-file basis. The location of block replicas can change over time to ensure data reliability.



NameNode

Single Point of Failure (SPOF) :

The **NameNode** is crucial in the Hadoop Distributed File System (HDFS) as it manages the metadata for the file system. However, it is a single point of failure, meaning that if it fails, the entire HDFS could become inaccessible.

Metadata Storage:

The **FsImage** file stores the entire file system's metadata, which includes the directory structure, file permissions, and block locations.

Changes to the file system that occur after the last checkpoint (a snapshot of the current state) are recorded in a separate transaction log called the **EditLog**.

Both the **FsImage** and **EditLog** are stored on the NameNode's local filesystem.

Availability and Redundancy :

To ensure high availability, redundant copies of the **FsImage** and **EditLog** are typically stored on Network-Attached Storage (NAS) or other servers. This way, the data can be recovered even if the NameNode's local storage fails.

NameNode



Checkpointing Process:

Checkpointing refers to the process of merging the **EditLog** with the **FsImage** to create an updated version of the file system's metadata.

This process only occurs when the NameNode is restarted. During startup, the NameNode reads the **EditLog** to update the **FsImage** with recent changes. The updated **FsImage** is then loaded into memory, and the **EditLog** is truncated (i.e., cleared) since its changes have been applied.

Once this process is complete, the NameNode is ready to manage the file system again.

Communication with DataNodes:

The NameNode does not directly send instructions to DataNodes. Instead, it embeds instructions in its responses to the heartbeat signals sent by DataNodes. This includes:

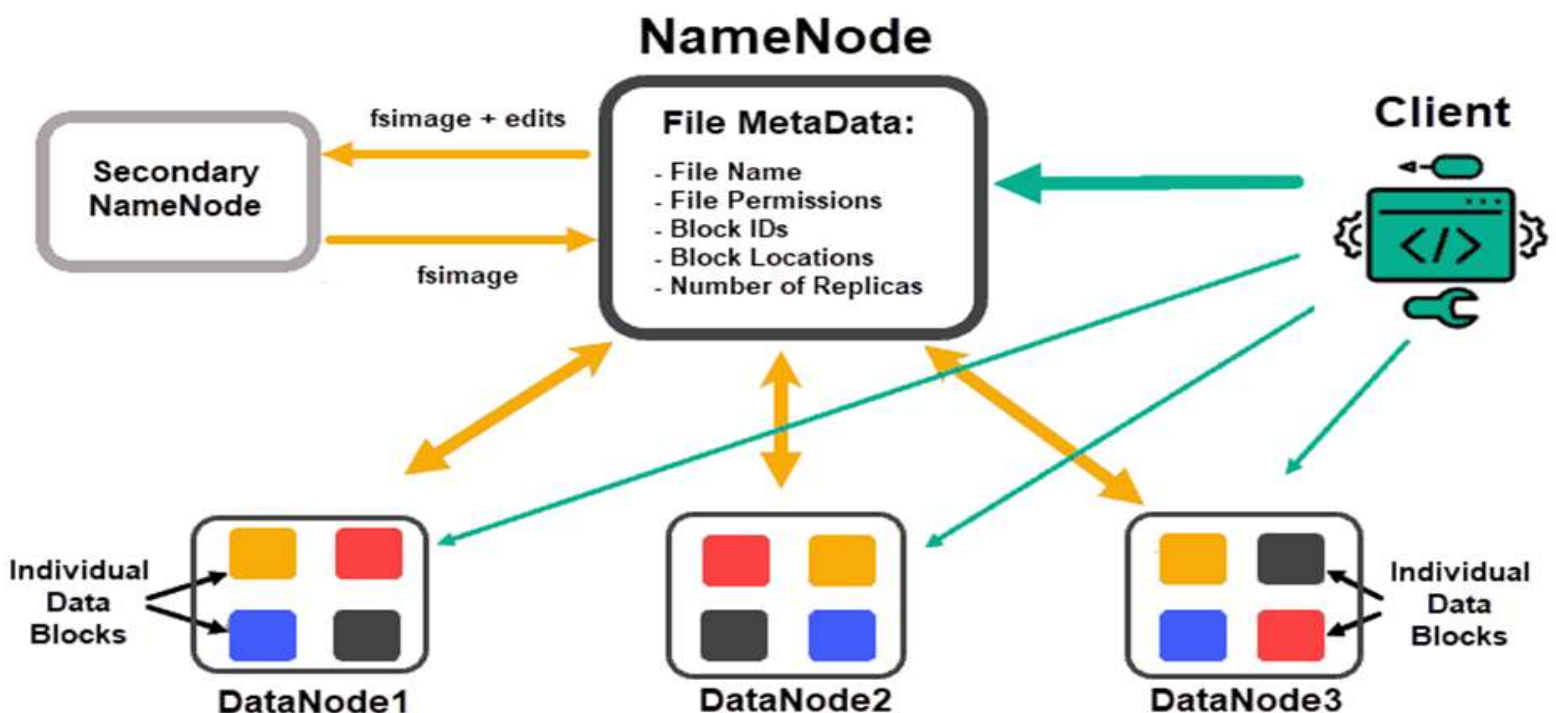
Replication: Directing DataNodes to replicate blocks to other nodes to ensure fault tolerance.

Block Removal: Instructing DataNodes to remove excess block replicas for load balancing.

Node Management: Telling DataNodes to restart (re-register with the NameNode) or to shut down.

Block Reports: Requesting an immediate block report from DataNodes to ensure accurate tracking of block locations.

NameNode



DataNodes



In Hadoop, a **DataNode** is responsible for storing the actual data of the file system. Each block replica on a DataNode consists of two essential files stored in the local file system:

- Block**: This file contains the actual data for the block.

Block Metadata: This file holds important information about the block, including its properties and state.



Additionally, each block replica is associated with a **block checksum** (which ensures data integrity) and a **generation timestamp** (indicating when the block was created).



Connection with the NameNode

When a DataNode starts up, it establishes a connection with the **NameNode** (which manages the file system namespace and metadata).

During this connection, a **handshake** occurs to verify the DataNode's ID and software version. If either of these does not match, the DataNode shuts down to prevent potential data corruption.

Importantly, DataNodes cannot join the cluster or register with the NameNode unless they possess a valid matching ID.

This mechanism helps maintain the integrity and security of the cluster.



DataNodes

Block Storage Efficiency

- Hadoop optimizes the use of storage space by only consuming the amount needed for the data blocks. It particularly focuses on ensuring that full blocks are utilized whenever possible, except for the last block of a file, which may

Reporting and Heartbeats

- DataNodes play a crucial role in maintaining communication with the NameNode:

Block Reports:

- Upon startup and subsequently every hour, DataNodes send a **block report** to the NameNode. This report details all the block replicas that the DataNode currently holds.

Heartbeats:

- DataNodes also send a **heartbeat signal** to the NameNode every 3 seconds (by default).
- If a DataNode fails to send a heartbeat within a 10-minute window, it is considered unavailable.
- In such cases, the NameNode removes the DataNode from its records and schedules the creation of new replicas of the blocks that were stored on the failed DataNode on other DataNodes.



Back to the NameNode

Metadata Storage

- The metadata for the file system is stored in two primary files on the NameNode's local file system:
 - **FsImage:** This file contains the complete and persistent snapshot of the filesystem metadata at a given point in time.
 - **EditLog:** This transaction log records all changes made to the filesystem since the last checkpoint. It captures every modification to the metadata, ensuring that no changes are lost.

• Both the FsImage and EditLog files are stored locally on the NameNode, but to enhance performance, they are also replicated on the Network-Attached Storage (NAS).

Checkpointing Process:

- **Occurs only at the startup of the NameNode**
- **Startup:** When the NameNode starts, it reads the last saved FsImage from its local filesystem.
- **Replay EditLog:** It then replays the EditLog to apply any changes made since the last checkpoint. This process updates the FsImage to reflect the current state of the filesystem.
- **Memory Mapping:** The updated FsImage is mapped into memory for efficient access and operations.
- **Truncation:** After the EditLog has been successfully replayed and the FsImage is updated, the EditLog is truncated to free up space and improve performance.
- This approach ensures that the NameNode is ready to manage the file system efficiently and effectively as soon as it starts up.

Client Operations in Hadoop



Reading Data

File Location:

- When a client initiates a read request, it first contacts the **NameNode** to obtain the locations of the file's data blocks.
- The NameNode provides the client with a list of DataNodes that hold the replicas of these blocks.

Data Retrieval:

- The client then reads the data blocks from the closest available DataNode(s).
- This proximity helps improve the speed and efficiency of the data transfer.

Data Integrity Check:

- To ensure the data's integrity, the client uses a **block checksum** to detect any corruption that may have occurred during storage or transmission.
- This is particularly important in clusters that contain hundreds to thousands of nodes and disks, where the risk of data corruption is higher.

Client Operations in Hadoop Writing Data



Write Request:

- For a client write operation, the client sends the file path to the NameNode and requests it to select a set of DataNodes based on the specified **Replication Factor** (the number of copies of each block that need to be stored).

Write Pipeline:

- The data is written to the DataNodes in a **serial pipeline manner**, which ensures efficient data transfer:
- **Data Bytes:** The client pushes data bytes into the pipeline as a sequence of packets.
- **Packet Buffering:** The client buffers these bytes until a packet reaches the default size of 64KB or until the file is closed. Once the buffer is filled, the packet is pushed into the write pipeline.
- **Checksum Calculation:** Along with the data, the client calculates and sends the corresponding checksum for the block to

Data and Metadata Storage:

- It's important to note that data and metadata (including checksums) are stored separately on the DataNodes, which helps organize and manage the information efficiently.

Client Operation in Hadoop

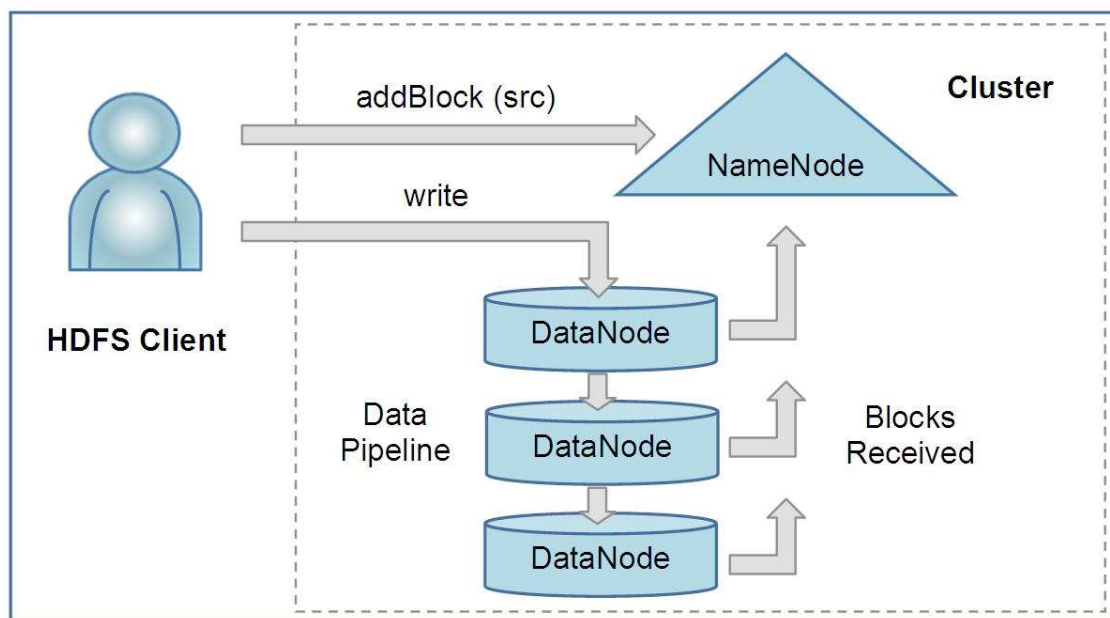


Asynchronous I/O

- The writing process employs **asynchronous I/O**, allowing the client to continue sending packets without waiting for an acknowledgment (ACK) from the DataNodes for each packet.
- The client can send multiple outstanding packets at once, determined by the **packet window size** (the maximum number of packets that can be in transit). This queue size is typically defined as $N = XR$, where X is the packet size and R is the number of outstanding packets.

Data Visibility

- Data becomes visible to new readers only after the file is closed or when the client invokes the **hflush** call, which flushes the buffered data to the DataNodes, making it accessible for reading.



Client Operations in Hadoop



Locality and High Performance in HDFS



The Hadoop Distributed File System (HDFS) differs significantly from traditional non-distributed file systems because its Application Programming Interface (API) reveals the location of a file's data blocks.



This design choice has important implications for performance and data processing.

Data Locality and Computation

Task Scheduling:

- Frameworks like **MapReduce** leverage this feature to schedule tasks that execute close to where the data is stored. This approach allows for:
 - **Computation at the Data:**
 - Instead of moving data to the computation, the system sends the computation to the data. This minimizes data transfer across the network, which is often a bottleneck in data processing.

Network Efficiency:

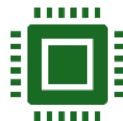
- By executing tasks where the data resides, HDFS reduces the reliance on network infrastructure, leading to:
 - **Lower Network Costs:** Less data movement reduces the overall network load.
 - **Shorter Processing Times:** Tasks complete faster since they operate directly on local data.

High Performance in HDFS



Read Performance

The highest read performance is achieved when tasks are executed with data located on **local disks**. This local access allows for faster data retrieval and processing.



Storage Technology

HDFS employs **point-to-point SAS (Serial Attached SCSI) or SATA (Serial ATA)** disks, which do not share a bus, making them key enablers of high performance. This design results in:

- **Superior Price-to-Performance Ratio:** HDFS provides a significant price advantage compared to traditional **SAN (Storage Area Network)** storage systems, offering better performance at a lower cost.



Replication and Fault Tolerance

HDFS supports an application-specific replication factor for each file, with the default set to **3**. This feature:

- **Enhances Fault Tolerance:** By keeping multiple copies of each block, HDFS can withstand hardware failures and maintain data availability.
- **Increases Read Bandwidth:** Having multiple replicas improves read performance, especially for files that are heavily accessed.

Typical Hadoop Node

A Hadoop node is a fundamental component in a Hadoop cluster, responsible for processing and storing data.

The configuration of a typical Hadoop node is designed to meet the demands of big data applications and is optimized for performance, storage capacity, and cost-effectiveness.

A typical Hadoop node is carefully configured to handle the requirements of big data processing and storage.

With its dual socket architecture, quad-core CPUs, significant memory, and multiple data disks, it is optimized for performance and efficiency in a distributed environment.

This setup not only supports heavy querying but also allows for scalable data storage solutions, making it suitable for a wide range of applications from real-time analytics to long-term data archiving.

Typical Hadoop Node - Hardware Configuration



Dual Socket 1U/2U Rack Mount Server:

The server is designed to fit into standard data center 1U (1.75 inches tall) or 2U (3.5 inches tall) form factor. This design maximizes space utilization in environments with space constraints.

Internal Disk Requirement:

Due to the need for multiple internal disks to store large amounts of data, blade servers are not suitable for typical Hadoop nodes. Blade servers are designed for high density and may lack the necessary storage capacity to support Hadoop's distributed storage model effectively.

Quad-Core CPUs:

Each node is equipped with quad-core CPUs operating at speeds greater than 2 GHz. This configuration allows the node to handle multiple threads simultaneously, improving the performance of data processing tasks, such as running MapReduce jobs.

Memory (16-24 GB) :

With 16 to 24 GB of RAM, the node can efficiently manage data processing tasks and operations. This amount of memory helps in caching data, reducing read times from disk, and improving overall system performance during high-demand scenarios.

Data Disks (4 to 6 @ 1TB) :

The node typically contains 4 to 6 data disks, each with a capacity of 1TB. This configuration allows for substantial local data storage, facilitating high-speed access to data during processing. The distributed nature of Hadoop enables data to be split into blocks, with these blocks stored across the various disks.

Typical Hadoop Node - Performance and Economic Efficiency

Performance Target:

- The node's hardware is specifically targeted towards workloads involving heavily queried data.
- It is designed to efficiently process large volumes of data and execute complex queries in a timely manner, benefiting from the combination of fast CPUs and ample disk space.

Economic Efficiency:

- The economic aspect of this node configuration is crucial, as it balances performance with cost.
- By utilizing commercially available components in a standard server format, organizations can achieve significant price-to-performance benefits compared to more specialized hardware solutions.

Typical Hadoop Node - Use Cases, Networking & OS



Long-Term Storage Considerations:

For use cases that prioritize long-term data storage, organizations might opt for a **DL370 5U form factor** server.

This type of server supports 14 internal 3.5" disks, providing extensive storage capacity for archiving large datasets over extended periods.

This configuration is suitable for organizations that need to store historical data while also ensuring easy access for future analysis.

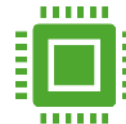


Networking

Each node is equipped with a **1 GbE NIC (Gigabit Ethernet Network Interface Card)**, enabling data transfer rates of up to 1 Gbps.

This speed is adequate for most Hadoop workloads, allowing for efficient data communication between nodes in the cluster.

However, as data demands grow, upgrading to faster networking options, such as 10 GbE, may be necessary to avoid bottlenecks.



Operating System:

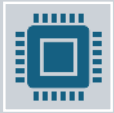
The typical operating system for a Hadoop node is **Linux**, which is preferred for its stability, security, and flexibility.

Linux provides a robust platform for running Hadoop, with extensive support for various distributions that cater to different needs in terms of performance tuning and management.

Hadoop MapReduce



Imagine a massive dataset, such as all Google search queries for a day.



Processing such a vast amount of data using traditional methods is inefficient and time-consuming.



Hadoop MapReduce offers a solution by breaking down the problem into smaller, manageable tasks that can be processed concurrently across multiple computers.

How Hadoop MapReduce Work



Data Division:

The massive dataset is divided into smaller chunks called blocks.

These blocks are distributed across multiple computers in a cluster.



Map Phase:

Each computer independently processes its assigned blocks.

A user-defined map function is applied to each record in the block.

The map function transforms the input data into intermediate key-value pairs.

Example: If processing search queries, the map function might



Shuffle and Sort:

Intermediate key-value pairs from all map tasks are collected.

The framework groups together values with the same key.

This phase ensures that all values associated with a particular key are together.



Reduce Phase:

The grouped data is processed by reducer tasks.

A user-defined reduce function combines the values for each key.

The reduce function produces the final output.

Example: For search queries, the reduce function would sum up the counts for each search term to determine the

Implementation Example: Word Count

Let's consider a simple word count example to illustrate the MapReduce process:

- **Input data:** A text file containing multiple lines of words.
- **Map function:**
 - Input: A key-value pair where the key is the offset in the file and the value is a line of text.
 - Output: A key-value pair where the key is a word and

Reduce function:

- Input: A key (word) and a list of values (counts).
- Output: A key-value pair where the key is the word and the value is the total count of occurrences.

Hadoop Implementation:

- Use the Hadoop API to define the map and reduce functions as Java classes.
- Configure the MapReduce job with input/output formats, number of mappers/reducers, etc.
- Submit the job to the Hadoop cluster for execution.

Real-Time Example: Analyzing Search Queries

Let's say you want to find the most popular search term on Google on a particular day. Here's how Hadoop MapReduce would tackle it:

Divide and Conquer

Divide and Conquer: The massive dataset of search queries is split into smaller chunks, each assigned to a different computer in the cluster.

Map

Map Phase: Each computer processes its chunk and counts the occurrences of each search term. This step is like tallying the votes in an election.

Shuffle and Sort

Shuffle and Sort: The results from all computers are combined and sorted by search term. This is similar to organizing the vote tallies from different polling stations.

Reduce

Reduce Phase: The sorted data is further processed to determine the most frequent search term. This is like counting the final votes to determine the winner.

Advantages of Hadoop MapReduce



Scalability:

Handles massive datasets by distributing processing across multiple machines.



Fault Tolerance:

Automatically recovers from failures by replicating data and tasks.



Efficiency:

Processes data in parallel, improving performance.



Simplicity:

Provides a high-level abstraction for complex data processing tasks.

Practical Applications of Hadoop MapReduce



Log Analysis:

Processing vast amounts of server logs to identify trends, anomalies, and potential security threats.



Web Analytics:

Analyzing website traffic patterns, user behavior, and clickstream data to optimize user experience and marketing campaigns.



Recommendation Systems:

Processing user data to suggest products, movies, or content based on preferences and behavior.



Fraud Detection:

Analyzing financial transactions to identify patterns indicative of fraudulent activities.



Scientific Computing:

Processing large datasets in fields like genomics, astronomy, and climate modeling.



Social Media Analysis:

Analyzing social media data to understand sentiment, trends, and user interactions.

Performance Optimization in Hadoop MapReduce



Data Locality:

Ensuring data is processed on the same node where it's stored to minimize data transfer.



Input Format and Output Format:

Choosing the appropriate input and output formats based on data characteristics.



Combiner:

Performing partial aggregation within the map phase to reduce data transferred to reducers.



Partitioner:

Controlling how data is distributed to reducers based on keys.

Performance Optimization in Hadoop MapReduce



Number of Mappers and Reducers:

Adjusting the number of mappers and reducers to balance workload and resource utilization.



Data Compression:

Compressing input and output data to reduce storage and network overhead.



Hardware Optimization:

Leveraging faster processors, more memory, and SSDs to improve performance.



Language Support



MapReduce is a programming model used to process large data sets with a distributed algorithm on a cluster.



Java natively supports the MapReduce framework, making it a good fit for stream processing and handling large-scale text processing tasks.



The framework allows the user to write Map and Reduce functions that process data in a pipeline, where the Map function's output (key-value pairs) is sorted and then passed to the Reduce function.



Python, Perl, and Ruby are also viable for MapReduce but typically use Unix pipes for processing. They are known for their text processing capabilities.



C++ can also be used but requires a different approach, involving socket programming, which deals with network communication rather than standard input/output streams. This approach is less common than the native Java or scripting language implementations.



Master/Slave Architecture in Hadoop's MapReduce:

Master/Slave Architecture:

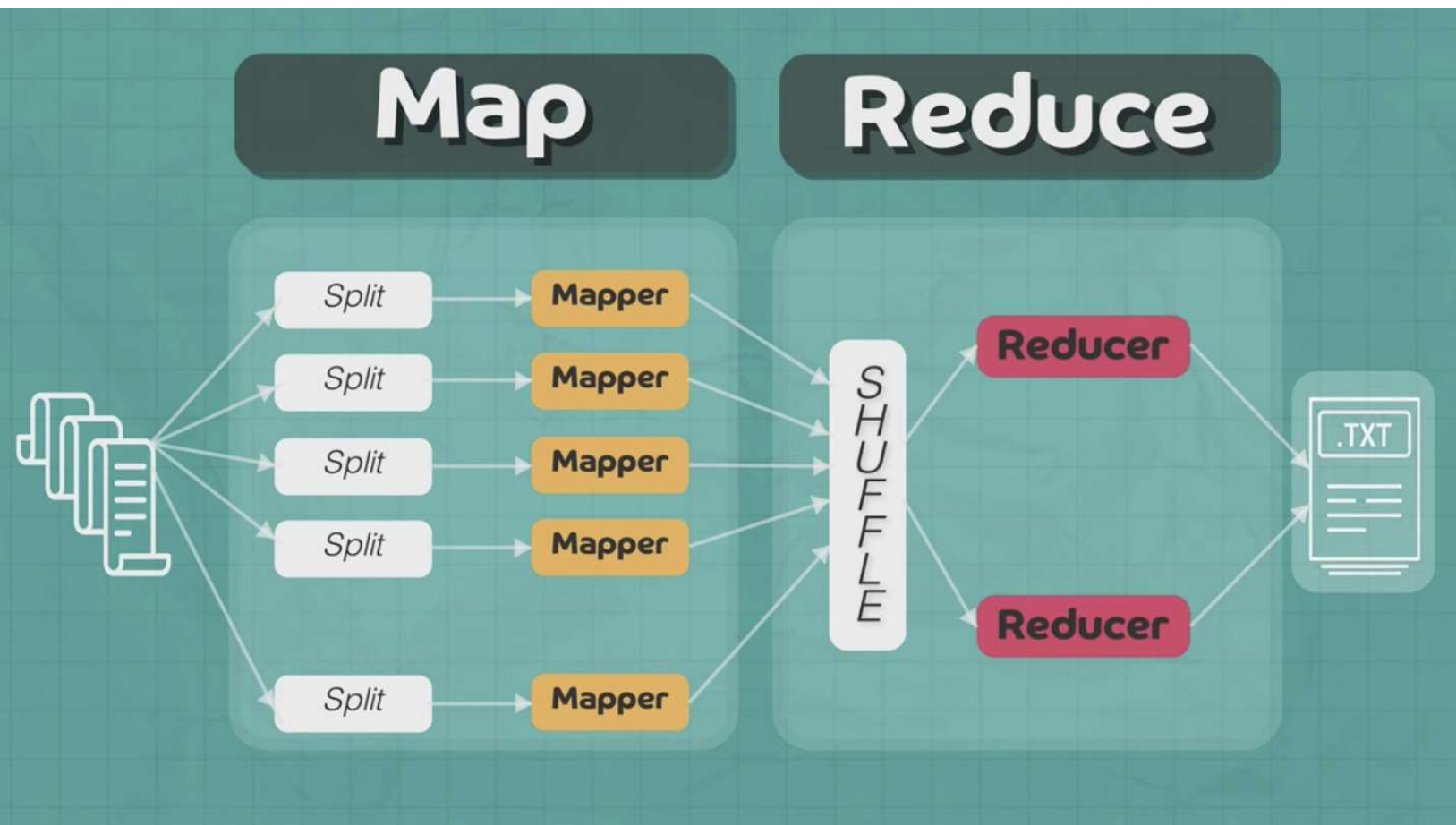
- Hadoop's MapReduce framework uses a **Master/Slave architecture** to manage and execute distributed data processing tasks.

Job Tracker (Master):

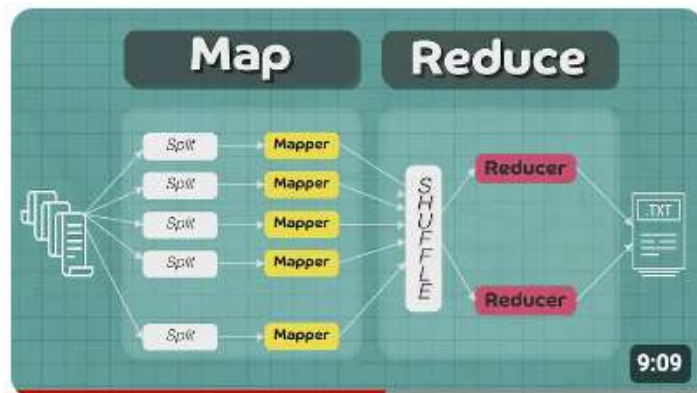
- The **JobTracker** acts as the **master** in the cluster.
- There is only **one JobTracker per cluster**.
- The JobTracker's main responsibilities are:
 - **Scheduling:** It schedules the tasks of a job across the cluster. Each job is divided into smaller tasks, which are then assigned to different nodes for execution.
 - **Monitoring:** It continuously monitors the tasks assigned to various nodes. If any task fails, the JobTracker reassigns that task to another node to ensure successful completion.

TaskTracker (Slave):

- Each DataNode in the cluster has a **TaskTracker**, which acts as the **slave**.
- The TaskTracker's role is to:
 - **Execute Tasks:** It performs the actual execution of tasks as directed by the JobTracker.
 - It ensures that the tasks are completed as per the instructions and reports back to the JobTracker with the status of task execution.



Map Reduce



Map Reduce explained with example | System Design

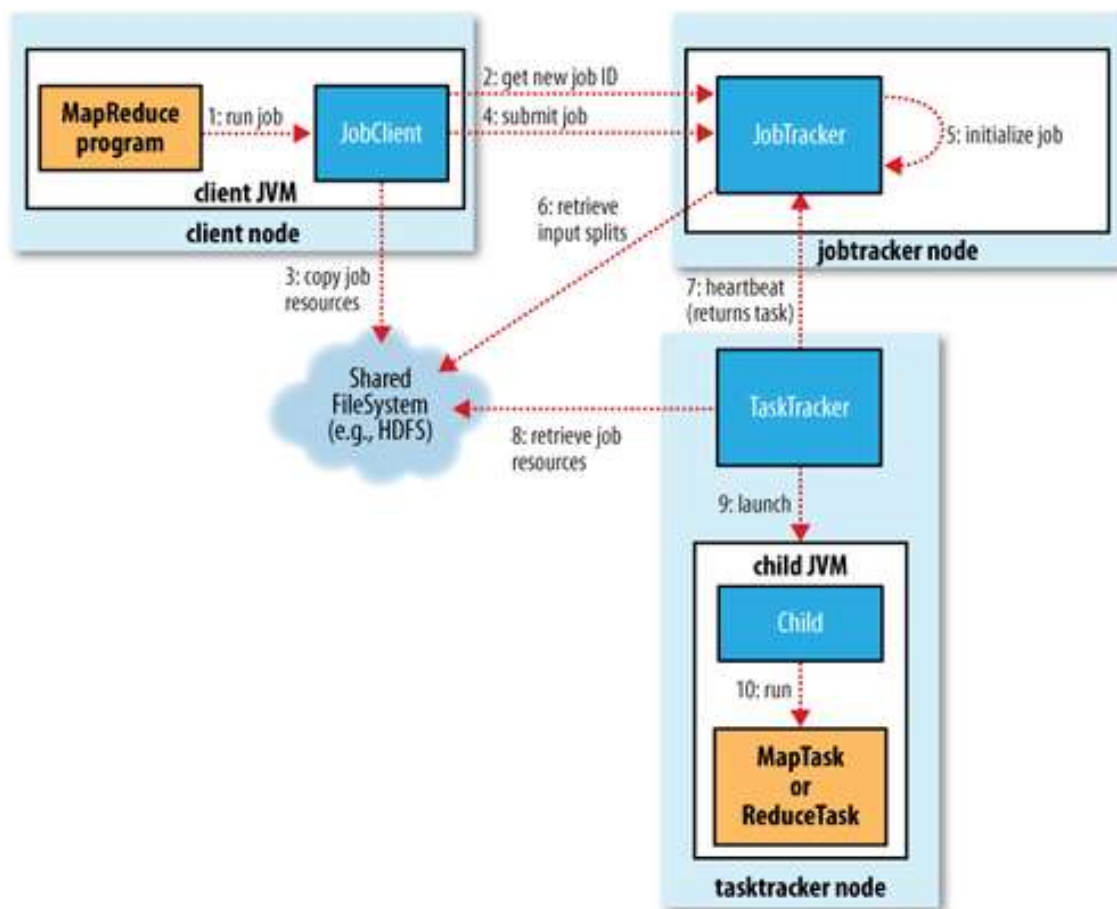
142K views • 1 year ago



ByteMonk

In this video I explain the basics of Map Reduce model, an important concept for any software engineer to be aware of. This...

4K



The MapReduce Job Submission Process:



Client Submits Job:

A **MapReduce program** written by the client submits a job to the cluster through the `JobClient.runJob` API.

The **JobClient** is an instance running in the client's Java Virtual Machine (JVM) on the client node.

Job Request Sent to JobTracker:

The **JobClient** sends a request to the **JobTracker**, which resides on a dedicated JobTracker node within the cluster.

The **JobTracker** is responsible for coordinating the execution of the job across the cluster.

Job ID Assignment:

Upon receiving the job request, the **JobTracker** assigns a unique **Job ID** to the job. This ID is used to track and manage the job throughout its lifecycle.

Data Splitting by JobClient:

The **JobClient** calculates the **splits** of the input data. These splits represent chunks of the data that will be processed by individual Map tasks.

The **JobClient** then **fans the job out** to multiple systems by preparing it for distributed execution across the cluster.

Program and Data Distribution:

The **JobClient** copies the necessary program components, such as the Java JAR file containing the MapReduce code and the configuration file, along with the calculated data splits to the **JobTracker's** filesystem.

By default, the **JobClient** writes 10 copies of this data to 10 different **JobTrackers** for redundancy and reliability. It's important to note

The MapReduce Job Submission Process:



Job Queueing and Initialization:

The JobTracker inserts the job into its internal **job queue**.

This queue is managed by the JobTracker's scheduler, which dequeues jobs and initializes them when resources are available.

During initialization, the JobTracker creates a **job object**.

This object represents the various tasks that need to be executed and keeps track of



Job Execution Notification:

The JobClient informs the JobTracker that the job is now ready to execute.

This triggers the JobTracker to move forward with task assignment.



Task Creation and Data Retrieval:

The JobTracker creates a list of **tasks** that need to be run as part of the job.

It retrieves the input data splits computed by the JobClient from the Hadoop Distributed File System (**HDFS**).

For each data split, the JobTracker creates a corresponding **Map task** that will process that particular piece of data.



Map Task Execution:

The Map tasks are distributed across the cluster and executed in parallel on the available TaskTrackers (slaves).

The output of these Map tasks is then processed by the Reduce tasks, leading to the final output of the MapReduce job.

TaskTrackers Communication with JobTracker:

Periodic Heartbeat Messages:

- TaskTrackers, which are slave nodes in the Hadoop cluster, communicate with the JobTracker via periodic heartbeat messages.
- These messages serve two main purposes:
 - **Indicate that the TaskTracker is Alive:**
 - The heartbeat confirms that the TaskTracker is operational and part of the cluster.
 - **Availability Status:**
 - The heartbeat message also informs the JobTracker whether the TaskTracker is available to run new tasks.

JobTracker Response:

- **Task Assignment:**
 - If a TaskTracker is available to run a task, the JobTracker includes a task assignment in its heartbeat response message.
 - This ensures efficient utilization of cluster resources by distributing tasks across available TaskTrackers.

TaskTrackers' Task Management:

Limited Queue Size:

- Each TaskTracker has a limited queue size for both Map and Reduce tasks.
- This queue size is determined by the number of CPU cores and the amount of physical memory available on the TaskTracker node.

Task Scheduling:

- The TaskTracker's internal scheduler prioritizes the execution of Map tasks over Reduce tasks.
- This approach helps ensure that data is processed as early as possible in the pipeline.

Task Preparation and Execution:

Copying Jar File:

- Once a task is assigned, the TaskTracker retrieves the necessary jar file containing the MapReduce code from the Hadoop Distributed File System (HDFS).
- This jar file is copied to the TaskTracker's local filesystem.

Unpacking the Jar File:

- The jar file is then unpacked into a local working directory specifically created for the task.
- This directory contains all the resources needed to execute the task.

TaskRunner Creation:

- A TaskRunner instance is created to manage the execution of the task.
- This component is responsible for the actual running of the Map or Reduce task on the node.

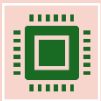
Starting a JVM for Each Task:

- The TaskRunner starts a new Java Virtual Machine (JVM) for each task.
- By running each task in a separate JVM, the TaskTracker ensures that the main TaskTracker process is isolated from any potential bugs or errors in the user's MapReduce program.

Isolation and Reliability



Separate JVMs: Running each task in its own JVM enhances the reliability and stability of the Hadoop cluster.



If a user's MapReduce program contains bugs or causes a crash, it will only affect that specific JVM instance and not the entire TaskTracker node or other running tasks.

Task Progress Reporting in Hadoop MapReduce

TaskTracker Per-Task JVM (Children) Communication:

- **Umbilical Interface:** Each task within a MapReduce job runs in a separate Java Virtual Machine (JVM), referred to as a "child" process.
- These child JVMs communicate with the parent TaskTracker via an "umbilical" interface. This interface is responsible for sending regular updates on the status and progress of the task every few seconds.

Streaming Interface:

- For tasks using the Hadoop Streaming API, the streaming interface handles communication between the TaskTracker and the task process using standard input (stdin) and output (stdout) streams.
- This is typically used when MapReduce jobs are written in languages other than

Importance of Status and Progress Communication:

- **Long-Running Jobs:** MapReduce jobs can run for extended periods, ranging from minutes to several hours. Because of this, it is crucial for the TaskTracker and JobTracker to monitor the status and progress of each task to ensure smooth execution and timely intervention if needed.
- **State Reporting:** Tasks report their state to the TaskTracker, which could be one of the following:
 - **Running:** The task is currently being executed.
 - **Successful Completion:** The task has completed its execution successfully.
 - **Failed:** The task has encountered an issue and failed.
- **Progress Reporting:** The progress of the task is communicated, indicating how much of the work has been completed. This includes the progress of both the Map and Reduce phases of the job.

Task Progress Reporting in Hadoop MapReduce

Job Counters and User-Defined Status Messages:

- **Job Counters:** Hadoop uses counters to measure various metrics during the execution of a job, such as the number of processed records, bytes read/written, etc. These counters are essential for monitoring the job's progress and performance.
- **User-Defined Status Messages:** Developers can define custom status messages that are reported back to the TaskTracker. These messages can provide more granular insights into the specific operations being

Defining Task Progress:

- **Input Record Read:** For Map tasks, progress can be measured by the number of input records read from the data split.
- **Output Record Written:** For both Map and Reduce tasks, the progress can also be measured by the number of output records written.
- **Reporter Interface:** Hadoop provides a Reporter interface that tasks can use to report progress and update status messages. The task can:
 - **Set Status Description:** Update the status description that is visible in the job tracker UI.
 - **Increment Counters:** Use the Reporter to increment custom or built-in counters.
 - **Call progress() Method:** Explicitly call the progress() method on the Reporter to indicate that the task is making progress and prevent the TaskTracker from assuming that the task has hung or failed.

Job Completion and Cleanup in Hadoop MapReduce

JobTracker Receives Last Task's Completion Notification:

- **Task Completion:** As tasks in a MapReduce job finish their execution, they notify the JobTracker of their completion. When the final task in the job finishes and sends this notification, the JobTracker knows that the entire job is done.

Job Status Changed to "Successful":

- **JobTracker's Role:** Upon receiving the last task's completion notification, the JobTracker updates the job's status to "successful." This indicates that all tasks have been executed without errors and that the job has successfully processed the input data.

HTTP Job Notification to JobClient:

- **Notification Mechanism:** The JobTracker can send an HTTP notification to the JobClient to inform it that the job has been successfully completed. This is an optional mechanism that can be configured depending on the setup.

Job Completion and Cleanup in Hadoop MapReduce

JobClient Polling and Success Notification:

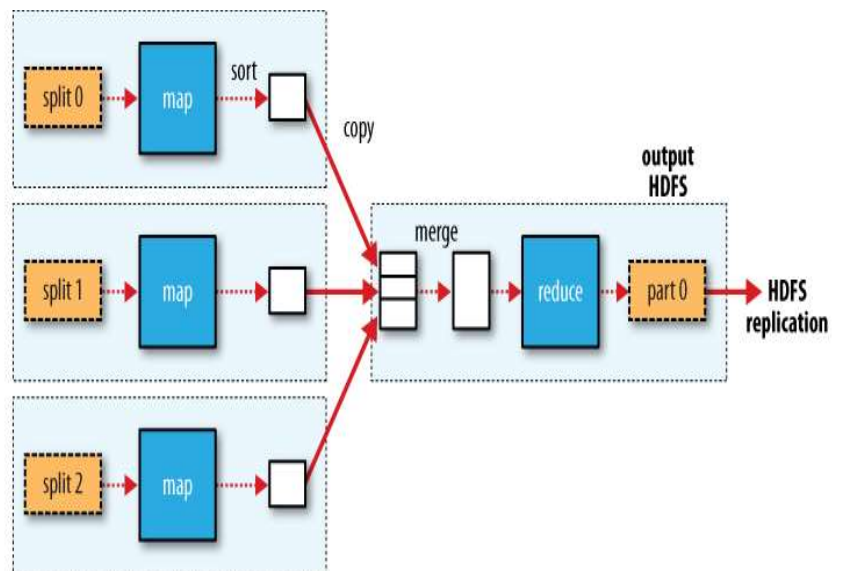
- **Polling for Status:** Even if the JobClient does not receive an HTTP notification, it regularly polls the JobTracker to check on the status of the submitted job. During its next polling interval, the JobClient will learn that the job has completed successfully.
- **User Notification:** After detecting the job's success, the JobClient notifies the user, typically via a console message or an email notification.

JobTracker Cleanup and TaskTracker Cleanup Commands:

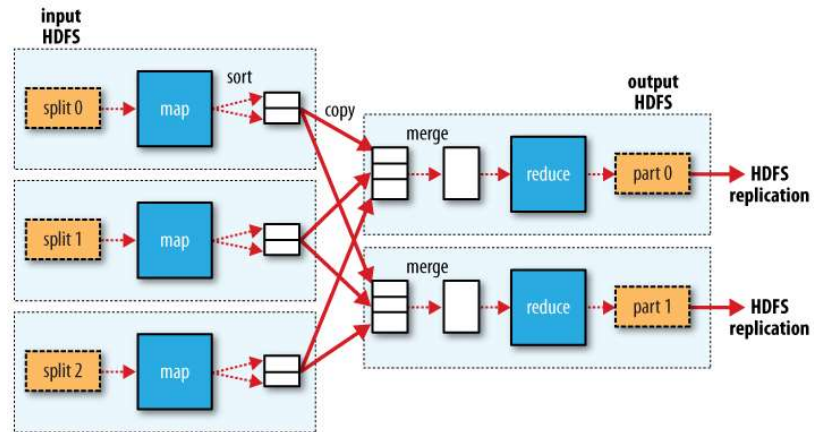
- **Post-Completion Cleanup:** Once the job has been marked as successful, the JobTracker initiates a cleanup process. This involves:
 - **Removing Temporary Files:** The JobTracker commands the TaskTrackers to clean up any temporary files or data that were created during the execution of the job. These might include intermediate Map outputs, temporary working directories, and other files stored on the TaskTracker nodes.
 - **Releasing Resources:** The JobTracker releases the resources allocated to the job, making them available for other jobs in the queue.
- **Commands to TaskTrackers:** The JobTracker sends specific cleanup commands to the TaskTrackers to ensure that all associated resources are freed, and the system is ready for new jobs.

Example Job & Filesystem Interaction

- Multiple Map tasks, one Reduce task
- Map reads from HDFS, writes to local FS
- Reduce reads from local FS, writes to HDFS



Example # 003 &
Filesystem
Interaction
Multiple Map
tasks, Two
Reduce tasks
Each Reduce
writes one
partition per
Reduce task to
HDFS



Apache Spark



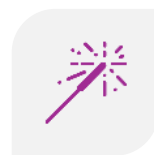
SPARK CORE: THE FOUNDATION OF THE APACHE SPARK FRAMEWORK PROVIDING IN-MEMORY COMPUTING CAPABILITIES.



SPARK SQL: MODULE FOR WORKING WITH STRUCTURED DATA, PROVIDING AN INTERFACE SIMILAR TO SQL.



SPARK STREAMING: ENABLES SCALABLE AND FAULT-TOLERANT STREAM PROCESSING OF LIVE DATA STREAMS.



MLLIB: MACHINE LEARNING LIBRARY FOR SPARK OFFERING VARIOUS ALGORITHMS AND UTILITIES.



GRAPHX: API FOR GRAPHS AND GRAPH-PARALLEL COMPUTATION.

NoSQL Databases



MongoDB: A document-oriented NoSQL database used for high-volume data storage.



Cassandra: A highly scalable NoSQL database designed for handling large amounts of data across many commodity servers.



Couchbase: A NoSQL database that provides distributed data management with built-in caching, indexing, and querying.

Data Warehousing Solutions



Amazon Redshift: A fully managed data warehouse service in the cloud.



Google BigQuery: A serverless, highly scalable, and cost-effective multi-cloud data warehouse.



Snowflake: A cloud-based data warehousing service known for its scalability and ease of use.

Data Integration Tools



Apache Nifi: An open-source data integration tool that supports powerful and scalable directed graphs of data routing, transformation, and system mediation logic.



Talend: Provides tools for data integration, data management, enterprise application integration, and big data.

Data Visualiz ation Tools



Tableau: Offers interactive data visualization products focused on business intelligence.



Power BI: Microsoft's suite of business analytics tools for data analysis and sharing insights.



D3.js: A JavaScript library for producing dynamic, interactive data visualizations in web browsers.

Machine Learning Platforms



TensorFlow: An open-source library for dataflow and differentiable programming across a range of tasks.



Scikit-learn: A free software machine learning library for the Python programming language.



PyTorch: An open-source machine learning library based on the Torch library, used for applications such as computer vision and natural language processing.

Cloud-Based Solutions

Amazon Web Services (AWS): Provides a comprehensive suite of cloud computing services including storage, databases, analytics, and machine learning.

Google Cloud Platform (GCP): Offers a range of cloud services for computing, storage, data analytics, and machine learning.

Microsoft Azure: A cloud computing service created by Microsoft for building, testing, deploying, and managing applications and services through Microsoft-managed data centers.

Ethical and legal considerations in big data analytics

Consideration	Ethical Concerns	Legal Concerns
Data Privacy	Respecting individuals' privacy, preventing unauthorized access or misuse of personal data.	Compliance with privacy laws (e.g., GDPR, CCPA), ensuring individuals' rights to access, correct, and delete their data.
Data Security	Protecting data from unauthorized access, breaches, and theft. Implementing	Legal consequences for data breaches, including fines and lawsuits. Compliance with

Ethical and legal considerations in big data analytics

Consideration	Ethical Concerns	Legal Concerns
Informed Consent	Obtaining informed consent from individuals before data collection. Ensuring users are aware of data use and sharing.	Laws require explicit consent before processing personal data. Non-compliance can lead to legal action.
Bias and Fairness	Avoiding the perpetuation or amplification of biases. Ensuring algorithms do not	Legal prohibitions against discrimination based on protected characteristics. Ensuring analytics

Ethical and legal considerations in big data analytics

Consideration	Ethical Concerns	Legal Concerns
Transparency and Accountability	Being transparent about data use, methodologies, and algorithms. Building trust and allowing for scrutiny.	Regulations may require explanations for automated decisions that significantly affect individuals. Lack of transparency can lead to legal challenges.
Data Ownership and Intellectual Property	Ensuring proper authorization for data use and crediting data originators. Addressing ethical issues of data sharing and ownership.	Intellectual property laws protect data and algorithms. Violations can result in disputes over ownership and copyright infringement.

Ethical and legal considerations in big data analytics

Consideration	Ethical Concerns	Legal Concerns
Social Impact	Considering the societal consequences of big data analytics, such as influencing public opinion or behavior.	Regulation of data use in areas like political campaigns and social media to prevent manipulation and ensure fairness.
Environmental Impact	Considering the environmental impact of data processing, including energy	Emerging regulations may address the environmental impact of large-