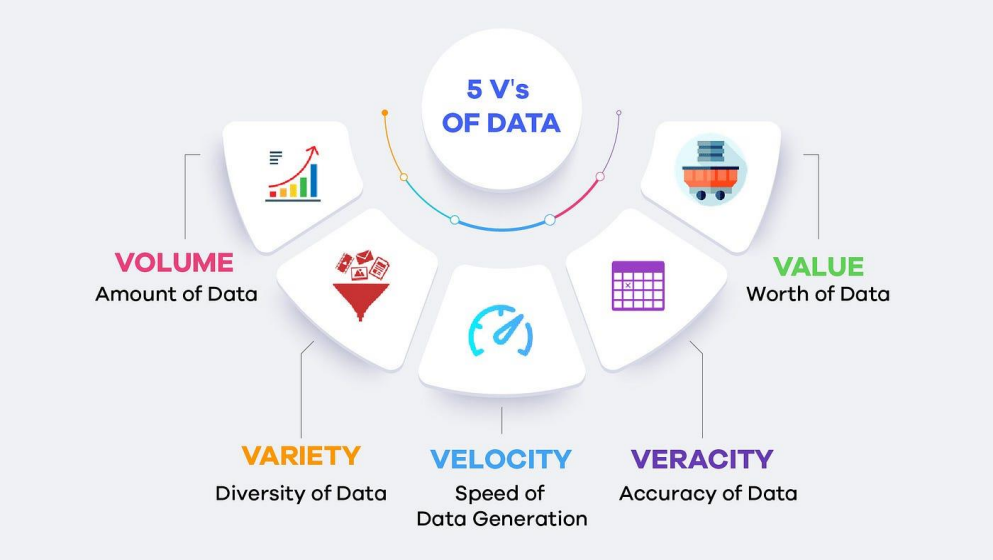
Master Spark Concepts Zero to Hero:

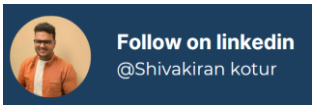
Big Data - Definition

Big Data refers to datasets that are so large and complex that traditional data processing systems are unable to store, manage, or analyse them efficiently. These datasets are characterized by the 5V's, which include:



5V's of Big Data

1. **Volume**
    - Massive amounts of data generated from various sources that exceed the storage capacity of conventional systems.
    - Example: Social media data, sensor data in petabytes.
2. **Variety**
    - Different formats and types of data, requiring different approaches for storage and processing:
        - Structured: Predefined schema (e.g., databases).
        - Semi-Structured: Partial schema (e.g., JSON, CSV).
        - Unstructured: No defined schema (e.g., images, videos).



3. Velocity

- The speed at which new data is generated and needs to be processed in real-time or near real-time.
- Example: Streaming data during online sales events.

4. Veracity

- The reliability and quality of data. Systems must handle inaccuracies, inconsistencies, and incomplete data. i.e

Is the quality or correctness of Data. Data cannot always be in the right format and we should be able to handle this not so high quality data.

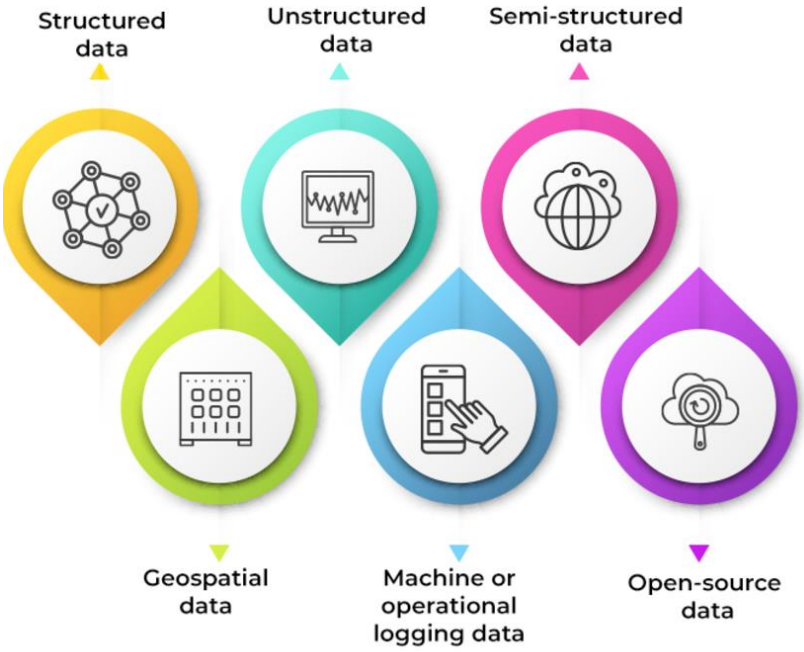
- **Example**: Dealing with errors in sensor readings.

5. Value

- The potential to extract meaningful insights from the data to make informed business decisions.
- Example: Analysing customer purchase patterns to improve marketing strategies.

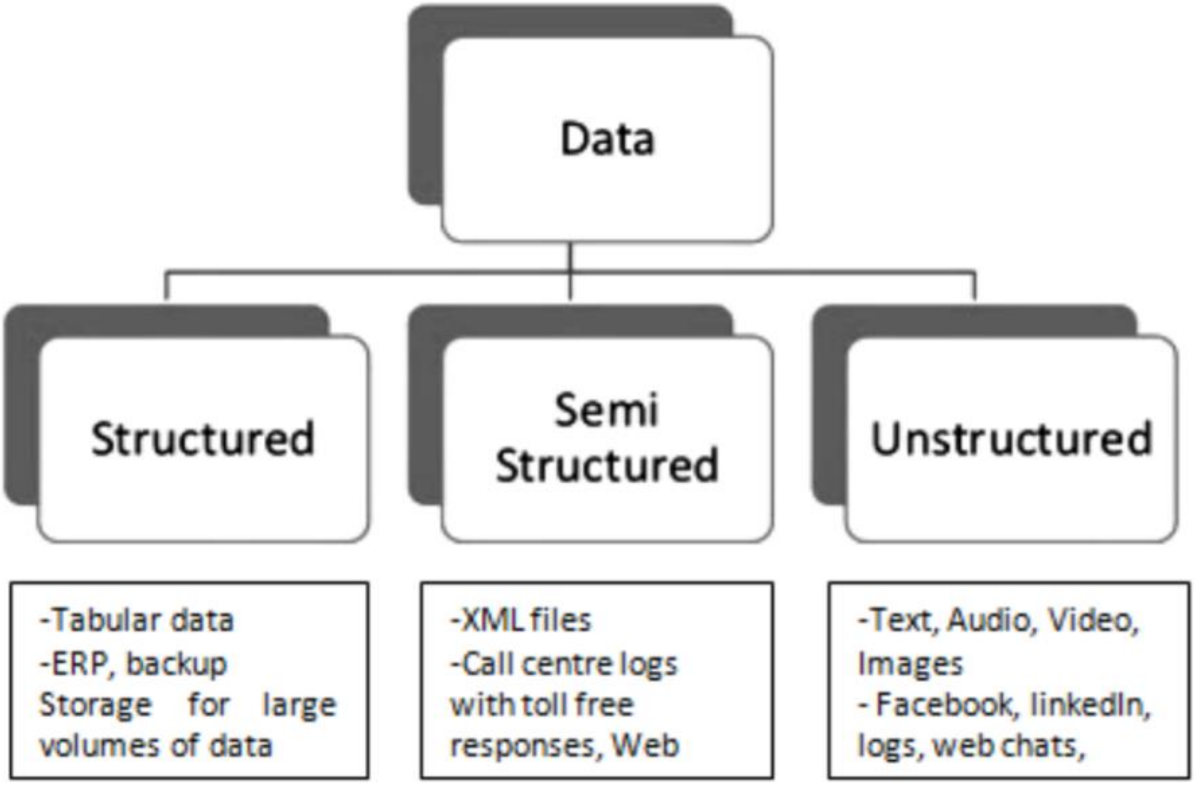
These characteristics distinguish Big Data from traditional data and drive the need for advanced processing technologies.

Types Of Big Data



Master Spark Concepts Zero to Hero:

Various Forms of data collected across



Unstructured data

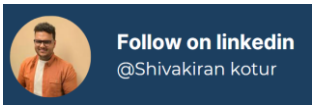
The university has 5600 students.
John's ID is number 1, he is 18 years old and already holds a B.Sc. degree.
David's ID is number 2, he is 31 years old and holds a Ph.D. degree. Robert's ID is number 3, he is 51 years old and also holds the same degree as David, a Ph.D. degree.

Semi-structured data

```
<University>
  <Student ID="1">
    <Name>John</Name>
    <Age>18</Age>
    <Degree>B.Sc.</Degree>
  </Student>
  <Student ID="2">
    <Name>David</Name>
    <Age>31</Age>
    <Degree>Ph.D. </Degree>
  </Student>
  ....
</University>
```

Structured data

ID	Name	Age	Degree
1	John	18	B.Sc.
2	David	31	Ph.D.
3	Robert	51	Ph.D.
4	Rick	26	M.Sc.
5	Michael	19	B.Sc.



1. Structured Data

- **Definition**: Data that is organized in a predefined format (rows and columns) and stored in relational databases. It has a fixed schema.
- **Characteristics**:
  - Easy to store, access, and analyze.
  - Follows strict data models (e.g., tables with rows and columns).
  - Queryable using SQL.
- **Examples**:
  - Customer database:
  - Financial transactions: bank records, sales data.
  - Sensor readings (if stored in structured formats).

---

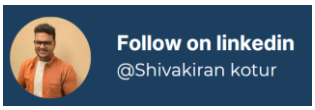
2. Unstructured Data

- **Definition**: Data that does not have a fixed format or schema and cannot be easily stored in traditional databases.
- **Characteristics**:
  - Often large in volume and stored in files or object storage.
  - Requires special tools for processing (e.g., Natural Language Processing, computer vision).
  - Harder to query and analyze directly.
- **Examples**:
  - Media files: Images, videos, audio recordings.
  - Text data: Social media posts, emails, PDF documents.
  - IoT data in raw formats: Logs, binary sensor outputs.

---

3. Semi-Structured Data

- **Definition**: Data that does not follow a strict tabular format but contains organizational markers (tags, keys) to separate elements. It is flexible yet partially organized.



- **Characteristics**:
    - Does not conform to relational models but can be parsed with tools.
    - Common in data exchange formats.
    - Easier to work with compared to unstructured data.
- **Examples**:
    - NoSQL databases (e.g., MongoDB, Cassandra).
    - CSV files with inconsistent row structures.

4. Geospatial Data

- **Format**: Represents geographical information (coordinates, polygons).
- **Examples**:
    - GPS data, satellite imagery, map boundaries (GeoJSON, Shapefiles).

---

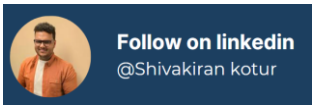
5. Machine or Operational Log Data

- **Format**: Logs from systems, often semi-structured or unstructured.
- **Examples**:
    - Server logs, IoT device logs, application error logs.

---

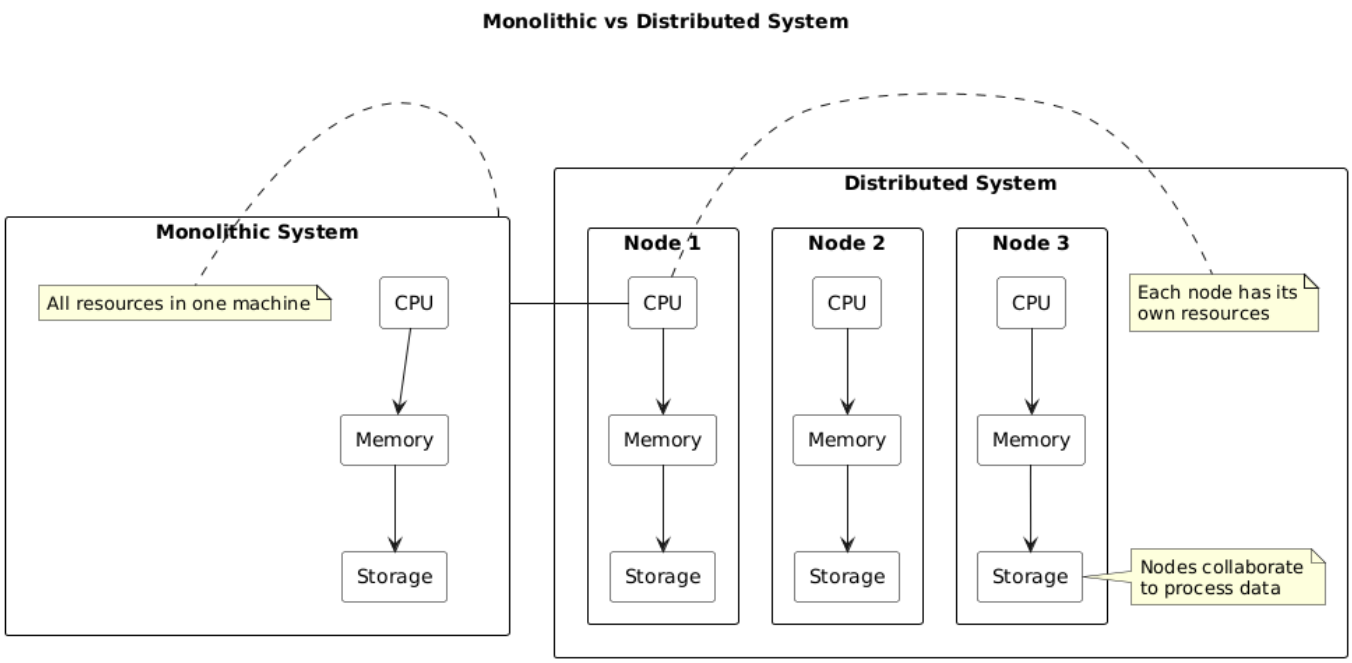
6. Open Source Data

- **Definition**: Freely available data for public use, often structured or semi-structured.
- **Examples**:
    - Census data, government APIs, Kaggle datasets, GitHub repositories.



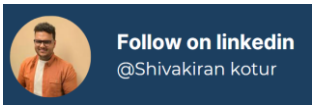
Master Spark Concept Zero to Hero:

Monolith vs. Distributed Systems in Big Data



Monolithic System

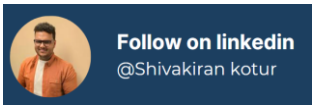
- **Architecture**:
  - Consists of a single, integrated system that contains all resources (CPU, RAM, Storage).
- **Resources**:
  - **CPU Cores**: 4
  - **RAM**: 8 GB
  - **Storage**: 1 TB
- **Scaling**:
  - **Vertical Scaling**: Increases performance by adding more resources (e.g., upgrading CPU, RAM, or storage) to the same machine.
  - **Limitations**:
    - Performance gains diminish after a certain point due to hardware constraints.



- Eventually, hardware limitations restrict the ability to effectively scale performance in proportion to the resources added.

Distributed System

- **Architecture**:
  - Composed of multiple interconnected nodes, each operating independently but contributing to a common system.
- **Node Resources** (Example of three nodes):
  - **Node 1**:
    - CPU Cores: 4
    - RAM: 8 GB
    - Storage: 1 TB
  - **Node 2**:
    - CPU Cores: 4
    - RAM: 8 GB
    - Storage: 1 TB
  - **Node 3**:
    - CPU Cores: 4
    - RAM: 8 GB
    - Storage: 1 TB
- **Scaling**:
  - **Horizontal Scaling**: Increases performance by adding more nodes to the system (e.g., adding more machines).
  - **Advantages**:
    - Performance increases in direct proportion to the number of nodes added, achieving **true scaling**.
    - Each node can independently handle its own workload, improving overall system performance and fault tolerance.

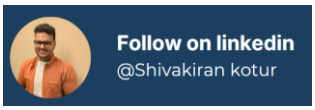


1. Monolithic Systems

- **Definition**: In monolithic architecture, all components of a system (data storage, processing, and user interface) are tightly integrated into a single framework or application.
- **Characteristics**:
    - Centralized design with all operations performed on a single machine or tightly coupled system.
    - Easier to manage in small-scale applications.
    - Requires more resources as data grows, leading to performance bottlenecks.
    - Limited scalability, as the system can only handle the data and processing power of one machine.
- **Challenges in Big Data**:
    - Difficulty in handling large volumes of data.
    - Single point of failure: If the system goes down, the entire operation stops.
    - Harder to scale horizontally (i.e., adding more machines).
- **Example**: Traditional relational databases (like MySQL on a single server) where both storage and processing occur on the same machine.

2. Distributed Systems

- **Definition**: In distributed architecture, data storage and processing are split across multiple machines or nodes, working together as a unified system.
- **Characteristics**:
    - Data and tasks are distributed across multiple machines (nodes) that collaborate to process data efficiently.
    - Highly scalable by adding more nodes to handle increasing data volumes and processing demands.
    - Fault-tolerant: Even if one node fails, the system continues to operate using the remaining nodes.
    - Offers better performance and resilience for handling massive datasets.



- Enables parallel processing, which significantly speeds up data analysis in Big Data environments.

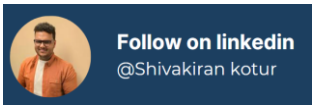
- **Advantages for Big Data**:
  - Scales horizontally, making it ideal for processing large datasets.
  - Handles a variety of data types and sources efficiently.
  - Can process data in real-time, distributing workloads across multiple nodes.
- **Example**: Apache Hadoop, Apache Spark, and other distributed systems designed for Big Data processing, where tasks are spread across multiple servers.

Key Differences:

Aspect	Monolithic System	Distributed System
Architecture	Single, tightly integrated system	Multiple nodes working together
Scalability	Limited (vertical scaling)	High (horizontal scaling by adding nodes)
Fault Tolerance	Low (single point of failure)	High (nodes can fail without affecting the system)
Processing Power	Limited to one machine	Parallel processing on multiple machines
Suitability for Big Data	Not well-suited for large datasets	Ideal for handling Big Data
Example	Traditional databases (e.g., MySQL)	Apache Hadoop, Apache Spark

Conclusion:

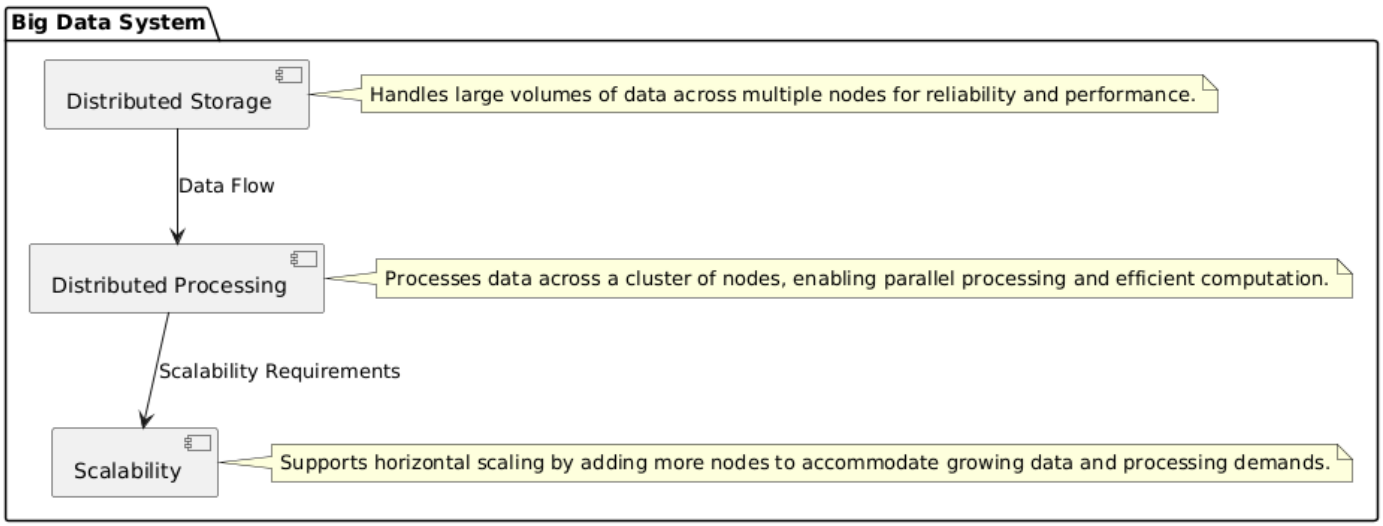
- **Monolithic systems** may work for smaller-scale applications but struggle with the volume, variety, and velocity of Big Data.
- **Distributed systems** are essential for efficiently processing and analyzing Big Data, offering scalability, fault tolerance, and high performance.



Master Spark Concept Zero to Hero:

Design the big data system

Here are the notes on the three essential factors to consider when designing a good Big Data System:



1. Storage

- **Requirement**: Big Data systems need to store massive volumes of data that traditional systems cannot handle effectively.
- **Solution**: Implement **Distributed Storage**.
    - **Definition**: A storage system that spreads data across multiple locations or nodes, allowing for better management of large datasets.
    - **Benefits**:
        - Scalability: Easily accommodates increasing data sizes by adding more storage nodes.
        - Reliability: Redundant storage across nodes enhances data durability and availability.
        - Performance: Enables faster data retrieval and processing through parallel access.

2. Processing / Computation

- **Challenge**: Traditional processing systems are designed for data residing on a single machine, which limits their capability to handle distributed data.

- **Solution**: Utilize **Distributed Processing**.
    - **Definition**: A computation model where data processing tasks are distributed across multiple nodes in a cluster.
    - **Benefits**:
        - Efficiency: Processes large volumes of data in parallel, significantly reducing processing time.
        - Flexibility: Adapts to various types of data and processing tasks without requiring significant changes to the underlying architecture.
        - Fault Tolerance: If one node fails, other nodes can continue processing, ensuring system reliability.

3. Scalability

- **Requirement**: The system must be able to adapt to increasing data volumes and processing demands.
- **Solution**: Design for **Scalability**.
    - **Definition**: The capability of a system to grow and manage increased demands efficiently.
    - **Benefits**:
        - Horizontal Scaling: Adding more nodes to the system allows for increased capacity and performance.
        - Cost-Effectiveness: Scaling out (adding more machines) is often more economical than scaling up (upgrading a single machine).
        - Future-Proofing: A scalable architecture can accommodate future growth without requiring a complete redesign.

Summary

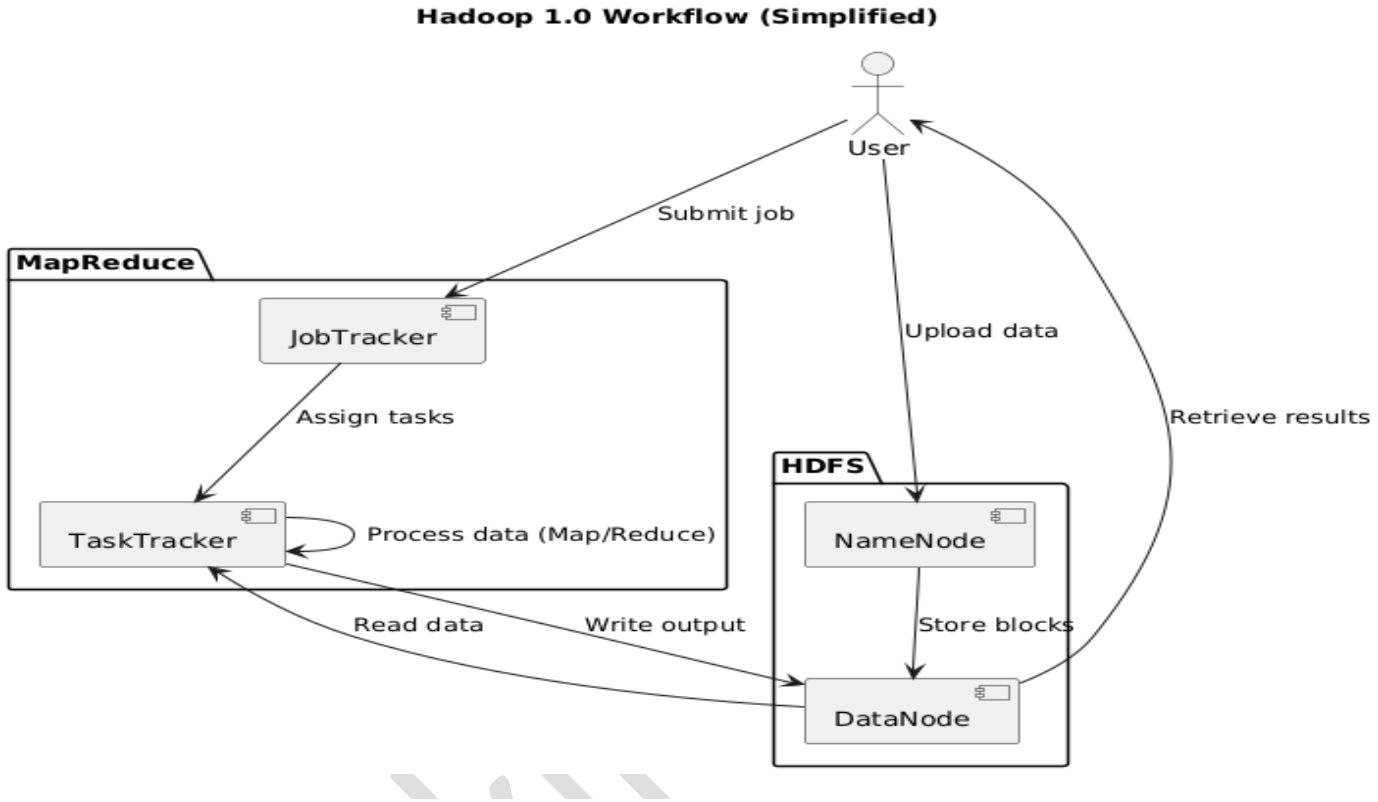
When designing a Big Data system, it's crucial to focus on:

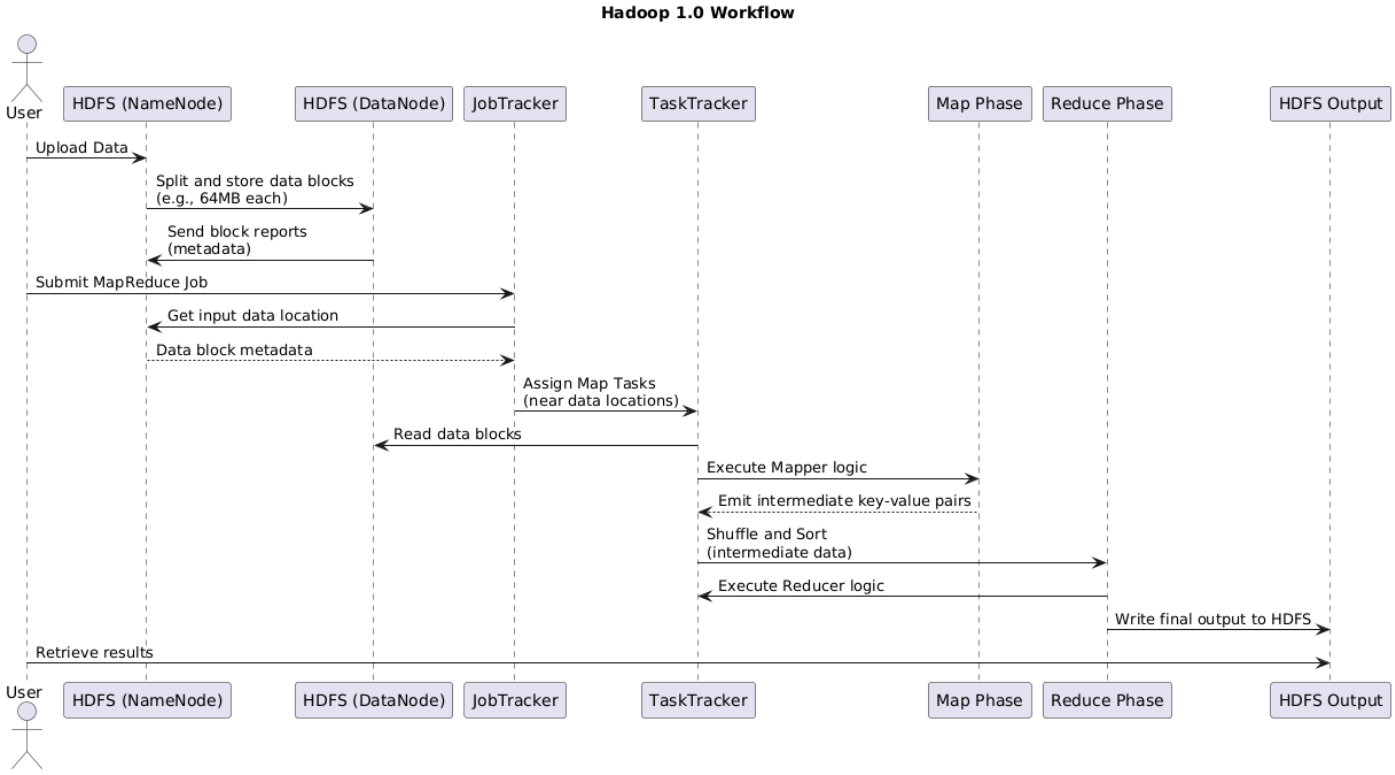
- **Storage**: Implement distributed storage solutions to handle massive datasets.
- **Processing**: Use distributed processing methods to efficiently compute across multiple nodes.
- **Scalability**: Ensure the system can grow to meet increasing data and processing demands, leveraging both horizontal scaling and cost-effective strategies.

Master Spark Concepts Zero to Big Data Hero:

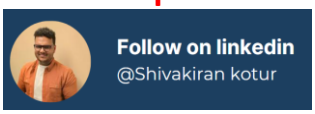
Hadoop Architecture Evolution

Hadoop Architecture 1.0





Core Components:



1. **HDFS (Hadoop Distributed File System):**
    - A distributed storage system that stores large data files across multiple nodes in the cluster.
    - Data is split into blocks (default 64MB or 128MB) and stored on DataNodes.
    - Key Components:
        - **NameNode (Master):** Stores metadata like file structure, block locations, and replication details.
        - **DataNode (Slave):** Stores the actual data blocks and sends periodic updates to the NameNode.
2. **MapReduce:**
    - A distributed data processing framework that processes data in parallel.
    - It involves two phases:
        - **Map Phase:** Processes and filters data, generating key-value pairs.
        - **Reduce Phase:** Aggregates the data produced by the Map phase.
    - Key Components:
        - **JobTracker (Master):** Manages job scheduling and resource allocation.
        - **TaskTracker (Slave):** Executes individual tasks assigned by the JobTracker.

How Hadoop 1.0 Worked (Step-by-Step)

1. **Data Ingestion and Storage in HDFS**:
    - Input data is uploaded to HDFS using commands or APIs.
    - The file is split into fixed-size blocks (e.g., 64MB) and replicated (default replication factor: 3).
    - NameNode stores the metadata about block locations, while DataNodes store the actual data blocks.
2. **Job Submission**:
    - A user submits a MapReduce job to the JobTracker.
    - The job contains:
        - Input data location in HDFS.

- Mapper and Reducer logic.
- Output data location.

3. **JobTracker Assigns Tasks**:
    - JobTracker splits the job into smaller tasks (Map and Reduce tasks).
    - It assigns these tasks to TaskTrackers on different DataNodes.
    - TaskTrackers fetch input data blocks locally (Data Locality optimization).
4. **Map Phase Execution**:
    - TaskTrackers execute the Mapper code on their assigned data blocks.
    - Each Mapper processes its block and generates intermediate key-value pairs.
    - The intermediate data is temporarily stored locally.
5. **Shuffling and Sorting**:
    - Intermediate key-value pairs are shuffled (grouped by keys) and sorted.
    - Data is then sent to the appropriate Reducers.
6. **Reduce Phase Execution**:
    - TaskTrackers run the Reducer code to process grouped key-value pairs.
    - The Reducer aggregates or performs computations (e.g., summing, counting) and generates final results.
7. **Output Storage in HDFS**:
    - Reducers write the final output data back to HDFS.
    - Users can access this data through HDFS commands or APIs.
8. **Monitoring and Reporting**:
    - JobTracker continuously monitors task execution and reassigns failed tasks to another TaskTracker (fault tolerance).
    - After all tasks complete, JobTracker provides a final report to the user.

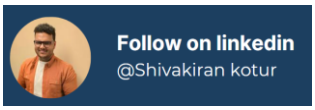
Key Highlights of Hadoop 1.0 Workflow:

- **Data Locality**: Processing happens where the data resides to minimize network overhead.
- **Fault Tolerance**: Failed tasks are retried or reassigned using replication.
- **Batch Processing**: Suitable for large-scale, offline data processing tasks.

This process enabled reliable, parallel processing of massive datasets, although it suffered from limitations like single points of failure and scalability issues in large clusters.

Why Hadoop 1.0 Failed: Limitations

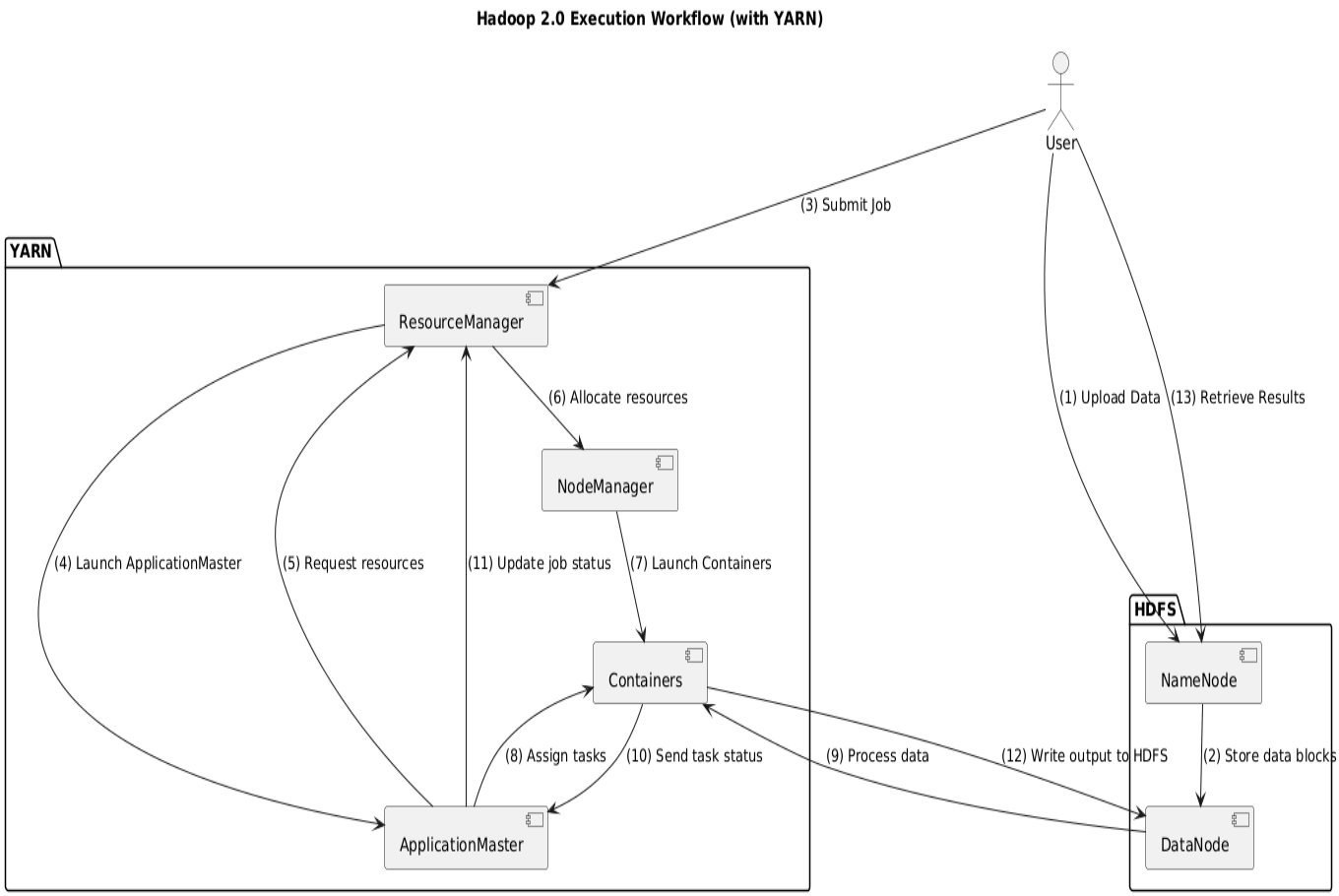
1. **Single Point of Failure**:
    - **NameNode**: If the NameNode crashed, the entire system became unavailable since it stored all metadata.
2. **Scalability Issues**:
    - The **JobTracker** managed job scheduling, task monitoring, and resource management, creating a bottleneck in large clusters.
3. **Resource Utilization**:
    - **Fixed Slot Model**: Each node had fixed slots for Map and Reduce tasks, leading to inefficient resource usage. For example, idle Map slots couldn't be used for Reduce tasks and vice versa.
4. **Lack of General-Purpose Computing**:
    - Hadoop 1.0 was designed only for **MapReduce** workloads, limiting its usability for other types of data processing like streaming or interactive queries.
5. **Cluster Utilization**:
    - Tasks couldn’t share resources dynamically, leading to underutilization of CPU, memory, and disk.
6. **Latency**:
    - High overhead in job initialization and monitoring due to reliance on **JobTracker**.



Hadoop 2.0: Overview and Steps

Hadoop 2.0 introduced **YARN (Yet Another Resource Negotiator)** to address the limitations of Hadoop 1.0 and decouple resource management from the MapReduce framework.

Hadoop 2.0 Workflow in Steps



1. **Introduction of YARN**:
   - YARN separates **resource management** from **job scheduling** and **monitoring**.
   - Key components:
     - **ResourceManager (Master):** Manages resources across the cluster.
     - **NodeManager (Slave):** Runs on each node and reports resource usage.
     - **ApplicationMaster:** Handles application-specific task scheduling.
2. **Data Storage in HDFS**:
   - HDFS remains the storage layer, but now it supports fault tolerance and NameNode High Availability (HA) using standby NameNodes.

3. **Resource Allocation**:
    - **ResourceManager** assigns resources dynamically to various applications (not just MapReduce) via containers.
4. **Application Submission**:
    - User submits a job to the **ResourceManager**.
    - The **ApplicationMaster** is launched to coordinate tasks for that specific job.
5. **Task Execution**:
    - **NodeManagers** on individual nodes launch containers to execute tasks.
    - Tasks can belong to any framework, such as MapReduce, Spark, or Tez.
6. **Dynamic Resource Utilization**:
    - Containers dynamically allocate CPU, memory, and disk resources based on the workload, improving utilization.
7. **Improved Scalability and Fault Tolerance**:
    - YARN allows scaling to thousands of nodes by delegating specific responsibilities to the ApplicationMaster and NodeManagers.
    - NameNode HA minimizes downtime.
8. **Support for Multiple Workloads**:
    - Beyond MapReduce, Hadoop 2.0 supports frameworks like Spark, Flink, and HBase for a variety of workloads.

Hadoop 2.0 Advantages:

- Eliminated **single point of failure** with NameNode HA.
- Improved resource utilization with dynamic allocation.
- Enabled **multi-tenancy** with support for multiple frameworks.
- Scaled efficiently for large clusters.

Hadoop 2.0 marked a major milestone in making Hadoop versatile, robust, and suitable for modern data processing needs.

Master Spark Concept Zero to Hero:

What is Hadoop?

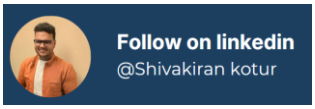
- **Hadoop** is an open-source framework designed for processing and storing large datasets in a distributed computing environment. It comprises a suite of tools that work together to address the challenges posed by Big Data.

Evolution of Hadoop

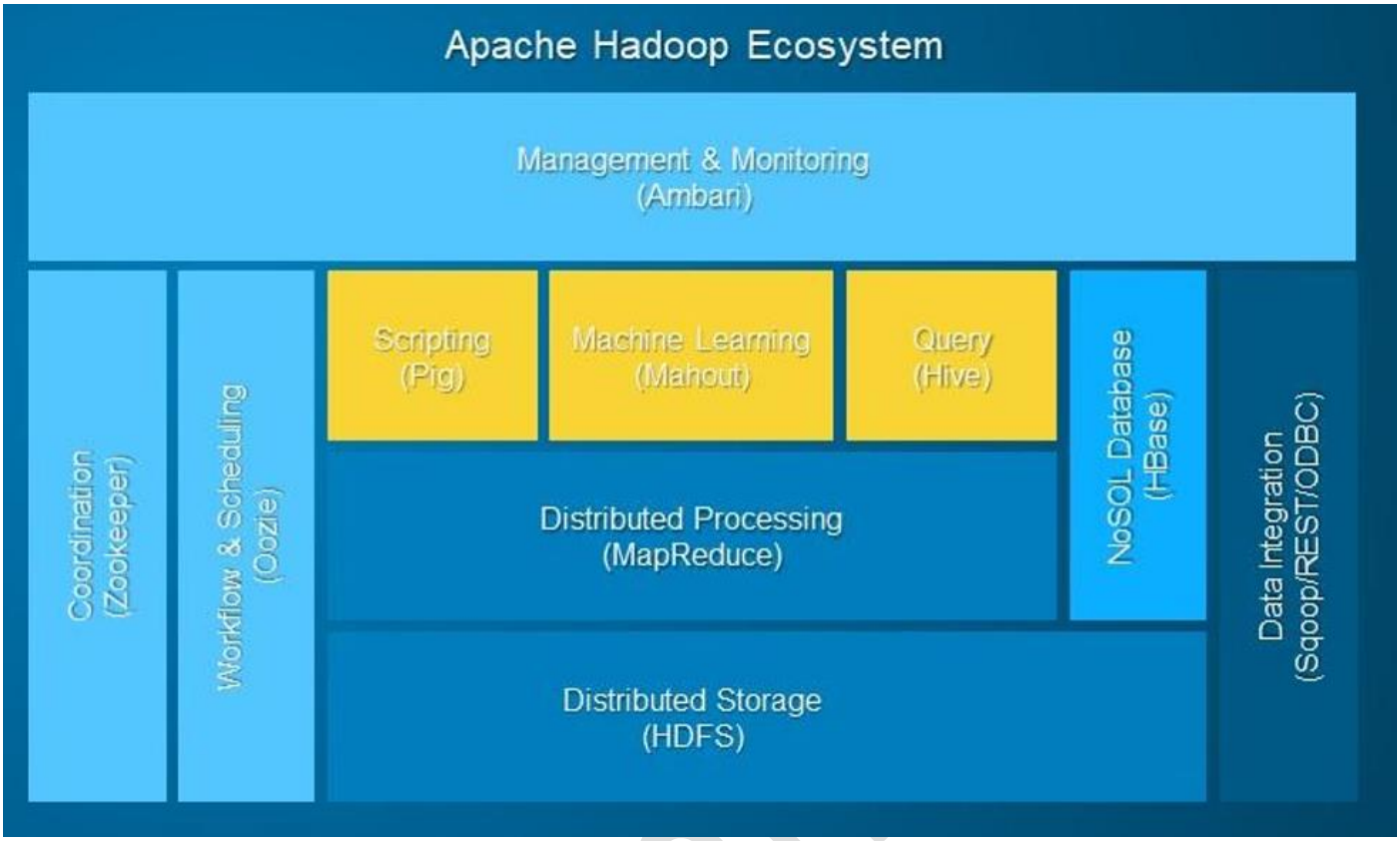
- **2007**: Introduction of **Hadoop 1.0**, laying the foundation for Big Data processing.
- **2012**: Release of **Hadoop 2.0**, introducing YARN for improved resource management.
- **Current Version**: **Hadoop 3.0**, featuring enhancements for scalability and performance.

Core Components of Hadoop

1. **HDFS (Hadoop Distributed File System)**:
    - A distributed storage system that enables the storage of large files across multiple machines, ensuring data redundancy and fault tolerance.
2. **MapReduce**:
    - A programming model for processing vast amounts of data in parallel. It simplifies the processing of data across the HDFS, though it can be complex and requires substantial code for execution.
3. **YARN (Yet Another Resource Negotiator)**:
    - A resource management layer that allocates and manages system resources across the Hadoop cluster, allowing multiple applications to run simultaneously.



Hadoop Ecosystem Technologies



- **Sqoop**:
  - Facilitates data transfer between HDFS and relational databases. It automates the import/export processes, making data movement efficient. *Cloud Alternative*: Azure Data Factory (ADF).
- **Pig**:
  - A high-level scripting language used for data cleaning and transformation, which simplifies complex data manipulation tasks. *Underlying Technology*: Uses MapReduce.
- **Hive**:
  - Provides a SQL-like interface for querying data stored in HDFS, translating queries into MapReduce jobs for execution.
- **Oozie**:
  - A workflow scheduler for managing complex data processing workflows, allowing for dependencies and scheduling of multiple tasks. *Cloud Alternative*: Azure Data Factory.

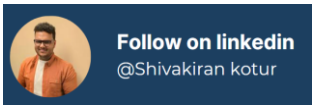
- **HBase**:
    - A NoSQL database for quick, random access to large datasets, facilitating real-time data processing. *Cloud Alternative*: CosmosDB.

Challenges with Hadoop

1. **MapReduce Complexity**:
    - Developing MapReduce jobs can be intricate, often requiring extensive Java coding, making simple tasks time-consuming.
2. **Learning Curve**:
    - Users must master various components for different tasks, resulting in a steep learning curve and increased complexity in the workflow.

Conclusion

Hadoop revolutionized the way we process and store Big Data. While its core components like HDFS and YARN remain vital, the complexity of MapReduce and the necessity to learn multiple ecosystem tools present significant challenges. Despite its evolution, alternatives are emerging to simplify Big Data processing, but Hadoop's foundational role in the Big Data era is undeniable.

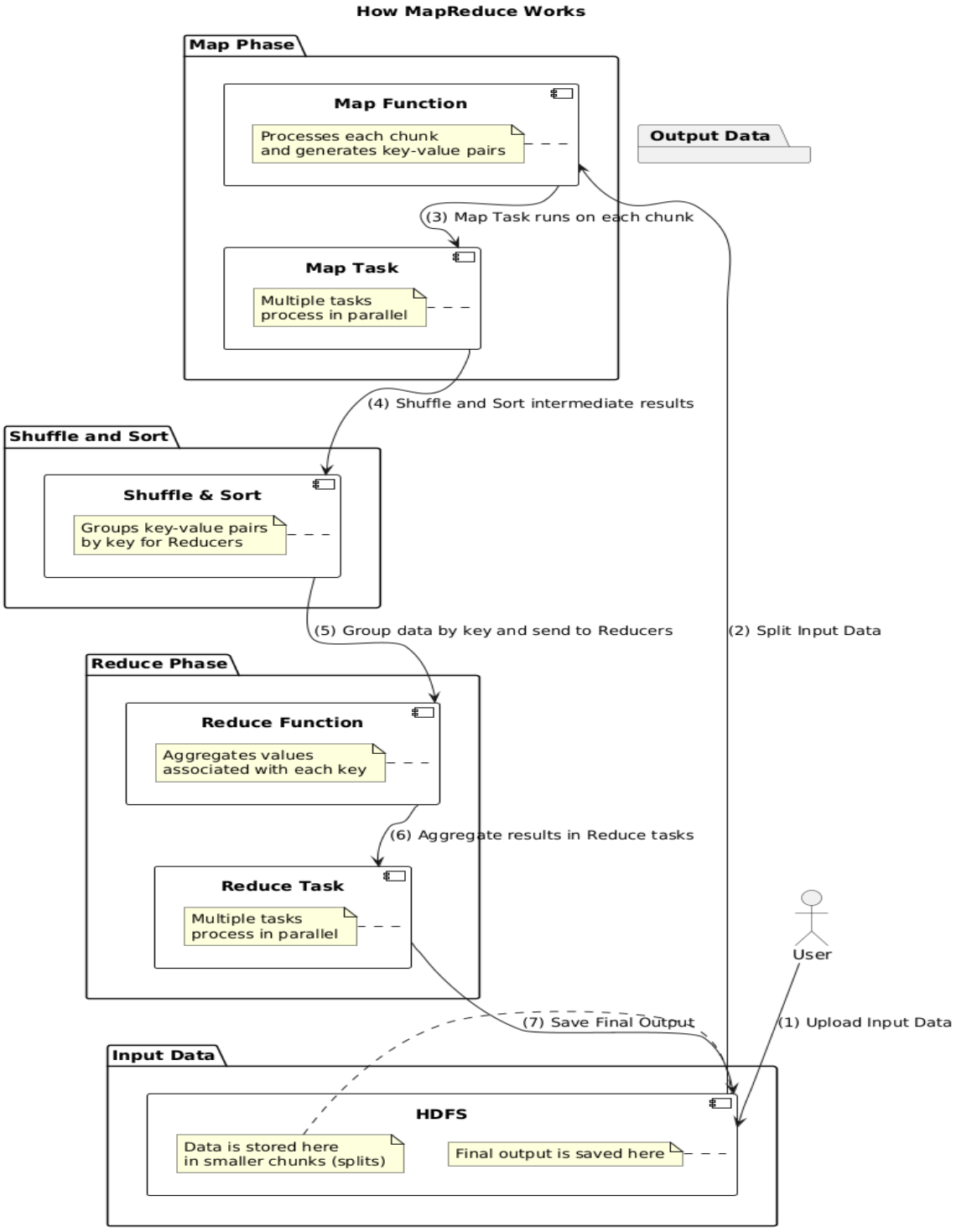


Master Spark Concepts Zero to Big Data Hero:

How does Map Reduce work in Hadoop (Simplified)

MapReduce is a programming model used for processing large datasets across distributed systems like Hadoop. It divides tasks into smaller jobs, processes them in parallel, and then combines the results.

Here's a simple breakdown of how **MapReduce** works:



1. The Basic Process:

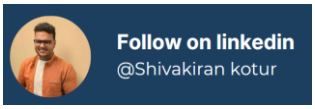
MapReduce involves **two main steps**:

- **Map**: Process input data and generate intermediate results.
- **Reduce**: Aggregate the results to produce the final output.

---

2. The MapReduce Workflow in Steps:

1. **Input Data**:
    - Data is stored in a distributed file system like **HDFS** (Hadoop Distributed File System).
    - The data is divided into smaller chunks called **splits**.
2. **Map Step**:
    - Each chunk of data is processed by a **Map function**.
    - The **Map function** processes each record (e.g., a line of text) and outputs **key-value pairs**.
        - Example: For word counting, the map function will output (word, 1) for each word in a document.
3. **Shuffle and Sort**:
    - After mapping, the system **shuffles and sorts** the data by the key (e.g., all occurrences of the word "apple" will be grouped together).
    - This step ensures that all records with the same key go to the same **Reduce function**.
4. **Reduce Step**:
    - The **Reduce function** takes each key and its list of values (e.g., for the word "apple", the values could be [1, 1, 1]).
    - The Reduce function processes these values (e.g., sums them up) and produces a final result (e.g., the total count of the word "apple").
5. **Output Data**:



- The final results are saved in **HDFS** for further use or analysis.

---

3. Example: Word Count

Let's say we want to count the number of occurrences of each word in a document.

Map Function:

- Input: A line of text.
- Output: Key-value pairs for each word in the line.

Input: "apple orange apple"

Output:

```
("apple", 1)
("orange", 1)
("apple", 1)
```

Reduce Function:

- Input: Key ("apple") and list of values ([1, 1]).
- Output: Total count for the word ("apple", 2).

Input:

```
("apple", [1, 1])
("orange", [1])
```

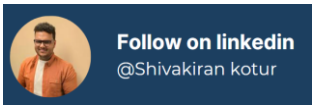
Output:

```
("apple", 2)
("orange", 1)
```

---

4. Summary of Steps:

1. **Map**: Process each record and generate key-value pairs.
2. **Shuffle & Sort**: Group the pairs by key.
3. **Reduce**: Aggregate the values for each key.
4. **Output**: Save the final results.



Master Spark Concepts Zero to Big Data Hero:

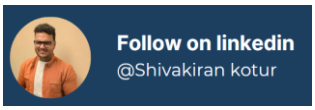
Disadvantages of MapReduce and Why Hadoop Became Obsolete

Hadoop's **MapReduce** framework revolutionized big data processing in its early years but eventually became less favorable due to the following **disadvantages**:

---

1. Limitations of MapReduce

1. **Complex Programming Model**:
    - Writing MapReduce jobs requires significant boilerplate code for even simple operations.
    - Developers need to write multiple jobs for iterative tasks.
2. **Batch Processing Only**:
    - MapReduce is designed for batch processing, making it unsuitable for real-time or streaming data processing.
3. **High Latency**:
    - The system writes intermediate data to disk between the Map and Reduce phases, resulting in high input/output overhead and slower performance.
4. **Iterative Computations Are Inefficient**:
    - Iterative tasks like machine learning or graph processing require multiple MapReduce jobs, with each job reading and writing to disk, causing inefficiency.
5. **Lack of In-Memory Processing**:
    - MapReduce does not leverage in-memory computation, which is faster compared to disk-based processing.
6. **Resource Utilization**:
    - MapReduce uses a static allocation of resources, leading to underutilization of cluster resources.
7. **Not Fault-Tolerant for Iterative Tasks**:
    - While MapReduce can recover from node failures, the re-execution of failed tasks for iterative workloads is time-consuming.



8. **Dependency on Hadoop 1.0's Architecture**:
    - The reliance on the **JobTracker/ TaskTracker** model caused scalability issues and made resource management inefficient.

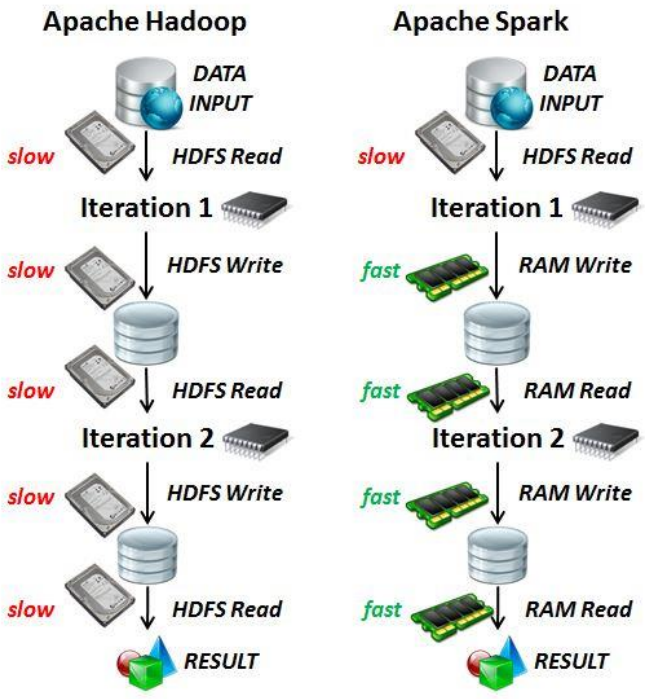
---

2. Why Hadoop MapReduce Became Obsolete

- **Emergence of Alternatives**: Advanced tools like Apache Spark provided a more efficient way of processing data with better APIs and support for real-time data processing.
- **Slow Evolution**: MapReduce couldn't keep up with the growing demand for interactive queries, in-memory computation, and real-time analytics.
- **Disk I/O Bottleneck**: Repeated disk read/write operations made it slower compared to in-memory processing engines.
- **Streaming Limitations**: Hadoop MapReduce lacked native support for stream processing, a critical need for modern applications.

Advantages of Apache Spark

Apache Spark, introduced as a faster and more versatile data processing engine, addressed the limitations of MapReduce and offered several advantages:



1. In-Memory Processing

- Spark processes data in memory, significantly reducing the time spent on disk I/O.
- This makes it much faster for iterative computations like machine learning and graph processing.

2. Unified Framework

- Supports multiple workloads:
  - **Batch Processing** (like MapReduce)
  - **Stream Processing** (real-time data)
  - **Interactive Queries** (e.g., Spark SQL)
  - **Machine Learning** (via MLlib)
  - **Graph Processing** (via GraphX)

3. Easy-to-Use APIs

- Provides high-level APIs in multiple languages like **Python**, **Scala**, **Java**, and **R**.
- Rich libraries for SQL, streaming, and machine learning simplify development.

4. Fast Performance

- Spark is **10-100x faster** than Hadoop for certain tasks, thanks to in-memory computation and optimized DAG execution.

5. Support for Real-Time Processing

- Spark Streaming enables the processing of real-time data streams, unlike Hadoop's batch-only capabilities.

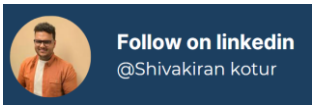
6. Dynamic Resource Management

- Spark utilizes cluster managers like **YARN**, **Kubernetes**, or **Standalone**, dynamically allocating resources to improve cluster utilization.

7. Fault Tolerance

- Spark achieves fault tolerance using **RDDs (Resilient Distributed Datasets)**, which track transformations and can recompute data automatically in case of failures.

8. Scalability



- Spark can scale efficiently from small clusters to thousands of nodes, handling petabytes of data.

9. Wide Ecosystem

- The Spark ecosystem includes libraries like:
  - **Spark SQL**: For structured data processing.
  - **Spark Streaming**: For real-time data processing.
  - **MLlib**: For machine learning.
  - **GraphX**: For graph-based computations.

10. Improved Iterative Processing

- Spark retains data in memory across iterations, making it highly efficient for algorithms like PageRank or k-means clustering.

Comparison of Hadoop MapReduce and Apache Spark

Feature	Hadoop MapReduce	Apache Spark
Processing Speed	Slower (Disk-based)	Faster (In-memory processing)
Ease of Use	Complex to code	Easy APIs for various languages
Real-Time Processing	Not Supported	Supported via Spark Streaming
Fault Tolerance	Yes, but slower recovery	Yes, with RDD-based fast recovery
Resource Utilization	Inefficient (Static)	Efficient (Dynamic allocation)
Supported Workloads	Batch processing only	Batch, Streaming, ML, Graph

Conclusion

Hadoop MapReduce played a pivotal role in big data processing during its time but became obsolete due to its inefficiency and inability to adapt to modern requirements. Apache Spark, with its fast, versatile, and easy-to-use framework, has emerged as the go-to solution for distributed data processing.