Big Data Frameworks

Hadoop and Spark

Hadoop is an open-source framework designed for the **distributed storage and processing of very large datasets** across clusters of commodity hardware. It achieves this through:

- **Distributed Storage:** Hadoop Distributed File System (HDFS) provides a reliable and scalable way to store massive amounts of data across multiple machines.
- **Parallel Processing:** Hadoop's MapReduce programming model enables you to process data in parallel across the cluster, significantly speeding up computations on large datasets.

Something interesting:

- HDFS is designed to handle files that range in size from gigabytes to terabytes, and even petabytes.
- HDFS divides these large files into smaller "blocks" (typically 128MB or 256MB each). These blocks are then distributed and stored across multiple machines in the cluster. This allows HDFS to store files that are larger than the capacity of any single machine.
- To ensure data reliability, HDFS replicates each block multiple times (usually 3) and stores these copies on different machines.

Spark is an open-source, **general-purpose cluster computing framework** that builds upon the foundation laid by Hadoop. It provides:

- **Faster Processing:** Spark leverages in-memory processing and optimized execution strategies to perform computations much faster than traditional Hadoop MapReduce, especially for iterative tasks.
- **Ease of Use:** Spark offers higher-level APIs (like PySpark) and a more flexible programming model, making it easier to develop and maintain big data applications.
- **General-Purpose Computation:** While Hadoop focuses primarily on batch processing, Spark supports a wider range of workloads, including batch processing, real-time streaming, machine learning, and graph processing.

Something interesting:

Spark can be seen as a **next-generation processing engine** that leverages Hadoop's distributed storage (HDFS) but improves upon its processing capabilities with in-memory computation.

Deep Dive

Hadoop

Key Components

- **HDFS (Hadoop Distributed File System):** A distributed file system that stores data across multiple machines. It breaks data into blocks and replicates them for fault tolerance.
- **MapReduce:** A *programming model* for processing data in parallel across a cluster. It involves two main steps:
  - **Map:** Process data in parallel, like dividing a task among many workers.
  - **Reduce:** Aggregate the results from the "map" step, like combining the work of all the workers.

Hadoop, with its MapReduce paradigm, is inherently designed for **batch processing**. This means it excels at processing large volumes of data that are already stored, typically in HDFS.

- **Disk-Oriented:** MapReduce reads data from disk, processes it, and writes intermediate results back to disk. This process is repeated for each stage in the workflow. While reliable, disk I/O introduces latency, making it less suitable for real-time applications.
- **High Latency:** The time it takes for Hadoop to process data is relatively high due to disk I/O and the inherent overhead of the MapReduce framework. This latency is acceptable for batch jobs where immediate results aren't critical.
- **Focus on Throughput:** Hadoop prioritizes **throughput** – processing massive amounts of data over time.

Examples:

- Analyzing website logs to understand user behavior.
- Processing financial transactions for monthly reporting.
- Generating recommendations based on historical purchase data.

Spark

- **Key Features:**
  - **In-Memory Processing:** Spark stores **intermediate** data in memory (RAM), making it much faster than Hadoop for many tasks, especially iterative ones.
  - **RDDs (Resilient Distributed Datasets):** These are the fundamental data structures in Spark. Think of them as distributed collections of data that can be processed in parallel.
  - **PySpark:** The Python API for Spark, which provides a pandas-like DataFrame API.

Spark is capable of **near real-time processing**, making it suitable for applications that demand quicker responses. Here's how it achieves this:

- **In-Memory Computation:** Spark's ability to store and process data in memory drastically reduces latency.
- **Micro-Batching:** Spark Streaming, a component of Spark, divides incoming data streams into small micro-batches and processes them in near real-time. This allows for continuous processing with very low latency.
- **Optimized Execution:** Spark's execution engine optimizes data flow, minimizing data shuffles and maximizing parallelism, further improving speed.

Examples:

- Fraud detection in credit card transactions.
- Monitoring social media feeds for trending topics.
- Analyzing sensor data from IoT devices for immediate insights.

Sparks's Execution Plan

DAG for spark application execution.

Key Components Spark's DAG:

- **Vertices:** Each vertex in a DAG represents an RDD or a stage of transformations.
- **Edges:** The edges connecting the vertices represent the operations or transformations applied to the RDDs. The direction of the arrows indicates the flow of data.
- **Directed:** It means that the data flows in a specific direction, from one RDD to another.
- **Acyclic:** No loops in the graph since the data flows in a linear fashion without circling back to previous stages.

DAGs provided better:

- **Optimization**: Since DAG allows spark to optimize the execution plan, leading to faster processing and improved performance
    - **Analyzing Dependencies:** DAGs clearly show how RDDs are related, allowing Spark to identify dependencies between operations.
    - **Reordering Operations:** Spark can reorder transformations in the DAG to reduce data shuffling and improve efficiency.
    - **Pipelines:** Spark can combine multiple operations into a single stage, minimizing the overhead of reading and writing intermediate data.
    - **Pruning:** Spark can eliminate unnecessary operations or entire branches in the DAG if they don't contribute to the final result.
- **Efficiency:** By breaking down the execution plan into stages and tasks, Spark can efficiently distribute the workload across the cluster.
    - **Stages:** DAGs are divided into stages, where each stage contains a set of tasks that can be executed in parallel without shuffling data between partitions.
    - **Tasks:** Each stage is further broken down into smaller tasks, each operating on a subset of the data.
    - **Distribution:** These tasks are then distributed across the cluster's worker nodes, allowing parallel execution and efficient use of resources.
