Ministry of Higher Education and Scientific Research
University of Mohamed Boudiaf, Msila

Faculty of Mathematics and Computer Science
Department of Computer Science

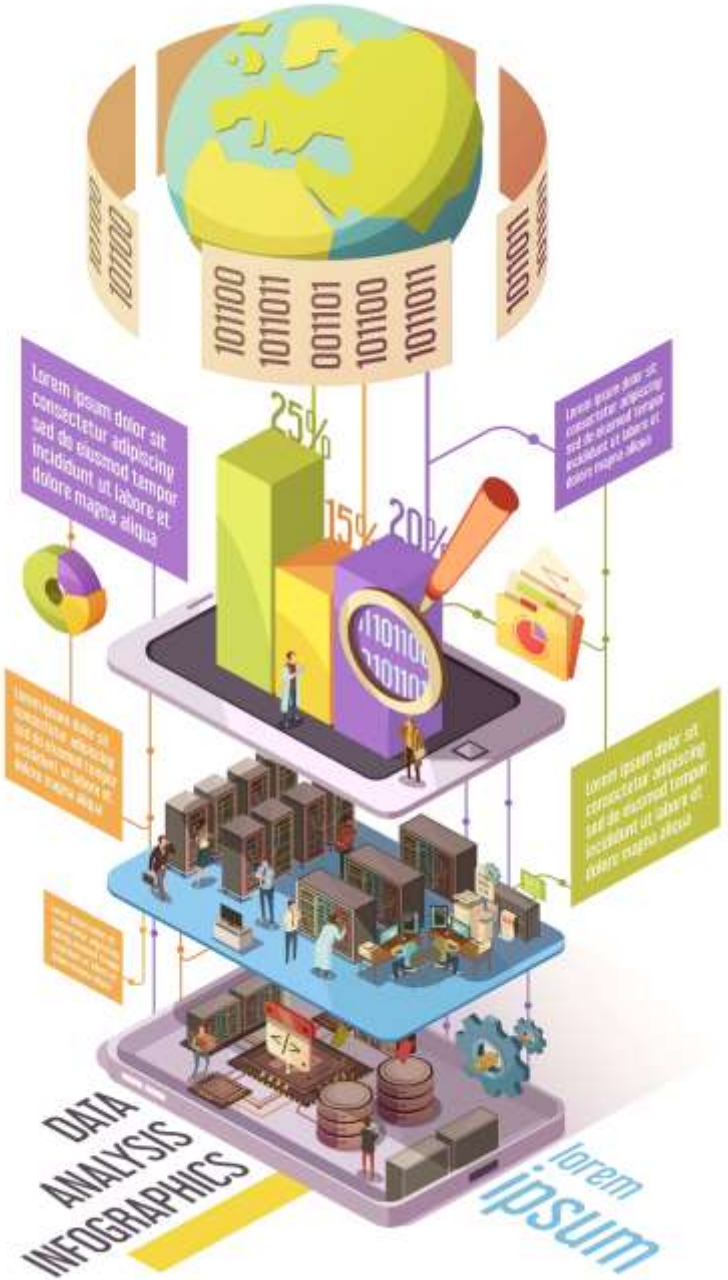
Mater 1

Big Data Analytics

Dr. N.CHALABI

October 2025

Big Data Architecture & Ecosystem

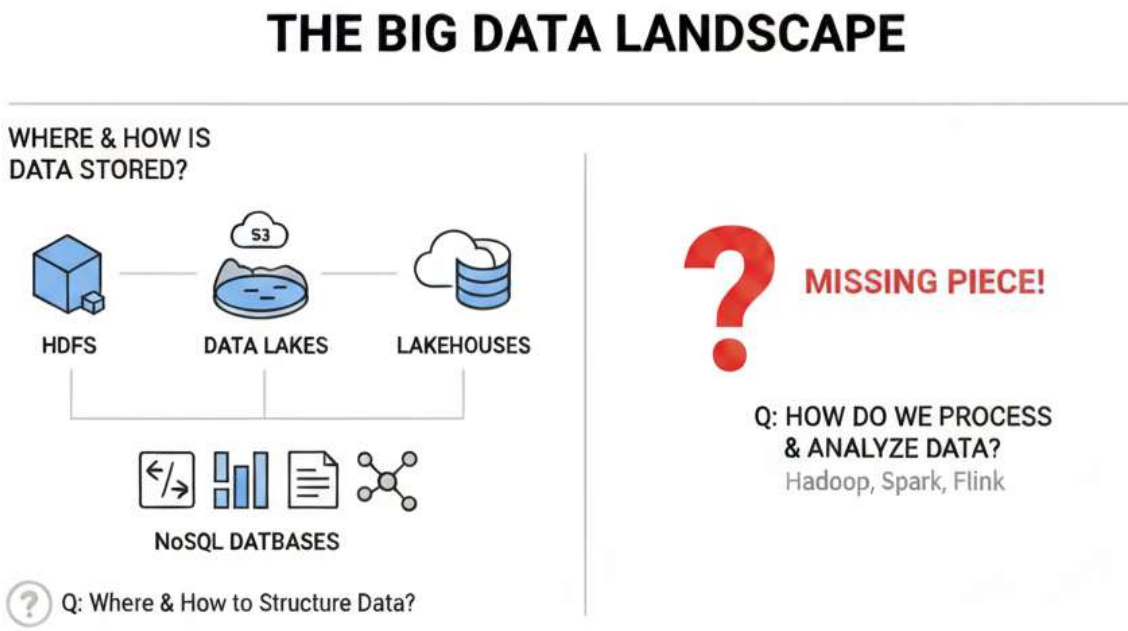


Hadoop, Spark, and Flink

Up to now, we have studied how Big Data is stored (HDFS, S3, Data Lakes, Lakehouses) and how it can be organized using NoSQL databases (key–value, column, document, graph). These components answer two basic questions:

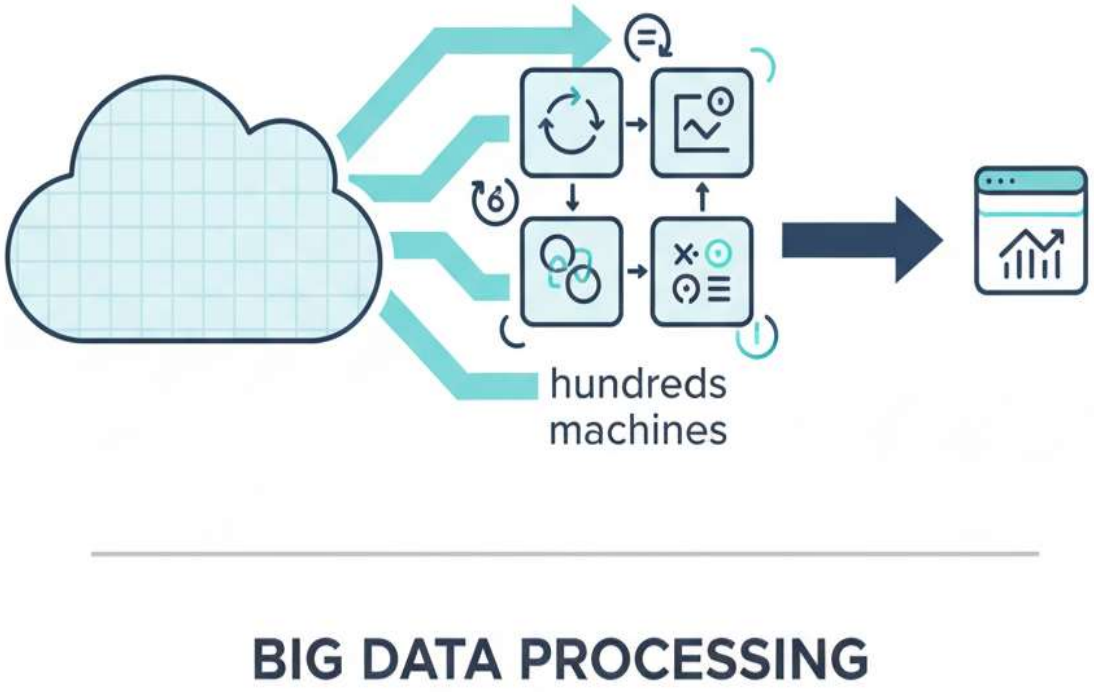
- **Where do we keep massive amounts of data?**
- **How do we structure and access that data efficiently?**

But there is still a missing piece: **How do we process and analyze such huge volumes of data?**



Imagine we have petabytes of logs, images, or sensor readings stored across hundreds of machines. If we try to process this data with a single computer, it would take years or be impossible due to memory limits. This is where **Big Data processing frameworks** come in.

Three of the most important frameworks are:



Hadoop : one of the earliest systems that showed how to process Big Data in parallel using the MapReduce model.

Apache Spark : a newer system that improved on Hadoop by making processing faster (in-memory), easier to program, and more versatile.

Apache Flink : a system designed for real-time and continuous data processing, going beyond the batch-oriented nature of Hadoop and Spark.

Each of these frameworks was created to solve the same problem " *how to process massive datasets across many machines"* but they take different approaches and have different strengths.

Why These Frameworks Exist

The rise of Hadoop, Spark, and Flink is closely tied to the limitations of older tools. Traditional data processing followed a simple model:

Data is stored in a relational database.

Queries (like SQL) are executed on a single server.

The system assumes the dataset fits on that server's disk and memory. This works for small or medium data, but it fails with Big Data because:

- **Volume**: Datasets are often in terabytes or petabytes, far beyond the capacity of one machine.
- **Velocity**: Data arrives continuously, sometimes faster than it can be written to disk.
- **Variety**: Data is not only tables but also images, logs, graphs, videos, and streams.
- **Fault tolerance**: If a single machine crashes, the entire process should not fail.

Big Data frameworks solve these challenges by:

- Splitting large tasks into **smaller tasks** that can run in parallel.
- Coordinating execution across **clusters of machines**.
- Handling **machine failures automatically**.
- Providing **programming models** that are easier to use than manually writing parallel code.

In other words, these frameworks are the *engines* that power Big Data analytics.

Hadoop: Batch Processing at Scale

What Problem Did Hadoop Solve?

In the early 2000s, companies started collecting **huge amounts of data** (terabytes or petabytes).

The question was:

How can we store and process all this data using many ordinary computers instead of one expensive machine?

Hadoop's Two Big Ideas:

1.HDFS (Hadoop Distributed File System)

1. Splits large files into **small blocks** (e.g., 128 MB each).
2. Distributes these blocks across **many computers (nodes)**.
3. Keeps **backup copies** of each block for fault tolerance.
4. If one machine fails, data can still be recovered from others.

2.MapReduce (Processing Model)

1. A way to **process large datasets in parallel**.
2. It breaks the task into two main steps:
    1. **Map:** Each computer processes a small piece of the data.
    2. **Reduce:** The partial results are collected and combined into the final answer.

HDFS (Hadoop Distributed File System) Code

```
# 1. Create a directory in HDFS
hdfs dfs -mkdir /user/student

# 2. Upload a local file to HDFS
hdfs dfs -put /home/student/bigdata.txt /user/student/

# 3. Check the file in HDFS
hdfs dfs -ls /user/student/

# 4. See where Hadoop stored the file blocks
hdfs fsck /user/student/bigdata.txt -files -blocks -locations

# 5. Check replication factor (default is 3)
hdfs dfs -stat %r /user/student/bigdata.txt
```

MapReduce (Processing Model)

1. Splitting:
means dividing a large dataset into smaller parts called *input splits*. Each split can be processed separately by different machines at the same time. This makes the work faster and allows big data to be handled in parallel across many computers.

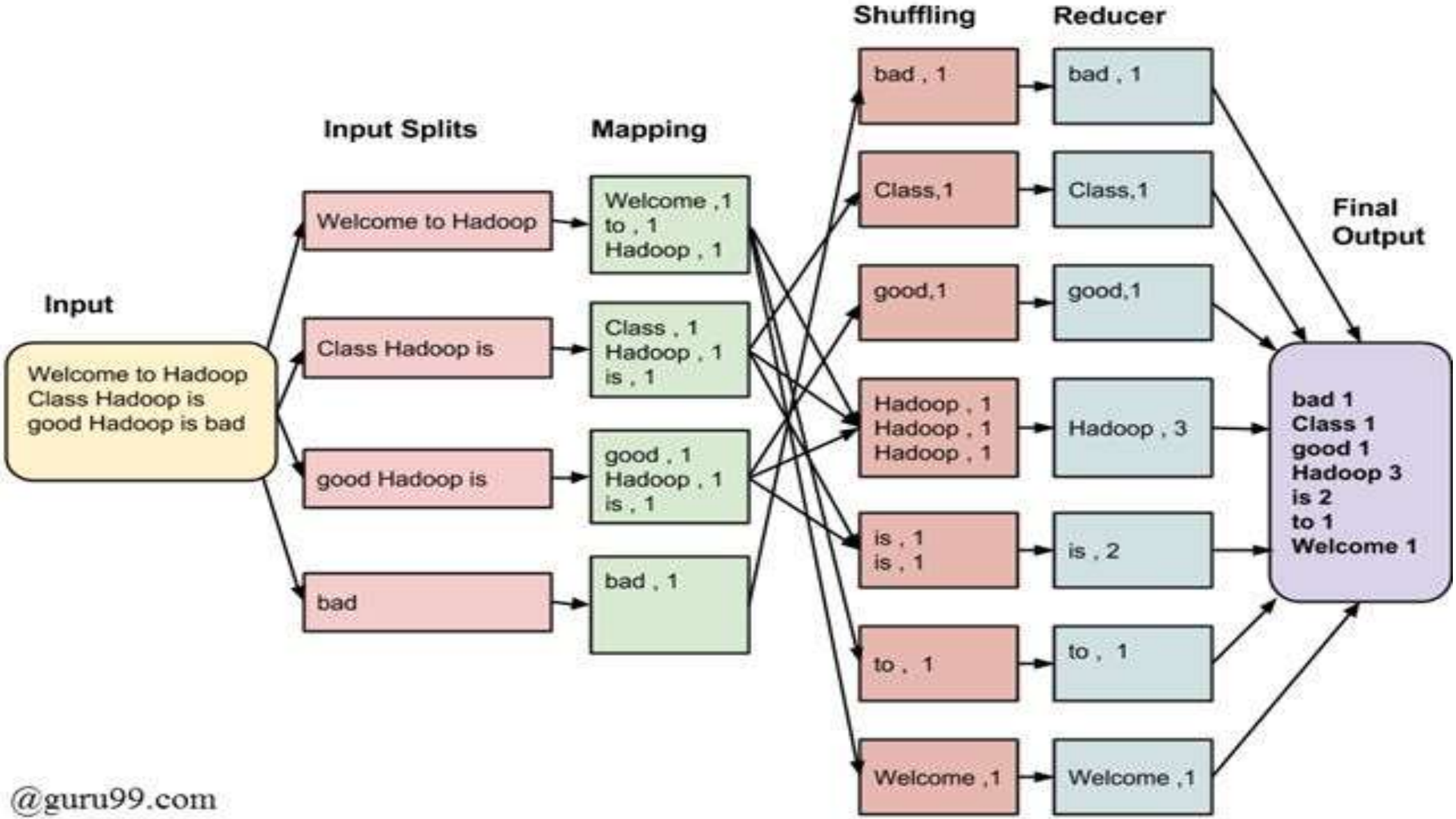
2. Mapping:
means applying a small function to each data split.
Each piece of data is transformed into **key–value pairs** like (word, 1).
This happens on many computers at once, so each split is processed in parallel, making the work much faster.

3. Shuffling:
happens after mapping.
The system **groups all the same keys together** — for example, all (k, v) pairs are sent to the same machine.
This step makes sure that each key and its values are in one place before reducing.
It may involve sorting, moving, and reorganizing data across nodes, which can take time. This step prepares the data for the **reduce phase**.

4. Reducing:

In the final *“reducing”* phase, data with the same key is grouped together, and a user-defined reduce function is applied to these key-value pairs. The reduce function is responsible for aggregating, filtering, or performing any necessary operations on the data.

The key-value pairs produced during the mapping phase are now processed to produce the final output. The reducing phase is where the desired computation on the data is performed. Like the other phases, the reducing phase also occurs in parallel, further accelerating data processing.


Input
Welcome to Hadoop
Class Hadoop is
good Hadoop is bad
Input Splits
Welcome to Hadoop
Class Hadoop is
good Hadoop is
bad
Mapping
Welcome ,1
to , 1
Hadoop , 1
Class , 1
Hadoop , 1
is , 1
good , 1
Hadoop , 1
is , 1
bad , 1
Shuffling
bad , 1
Class,1
good,1
Hadoop , 1
Hadoop , 1
Hadoop , 1
is , 1
is , 1
to , 1
Welcome ,1
Reducer
bad , 1
Class,1
good,1
Hadoop , 3
is , 2
to , 1
Welcome ,1
Final Output
bad 1
Class 1
good 1
Hadoop 3
is 2
to 1
Welcome 1

Hadoop is strong because it can store and process huge amounts of data and keep working even if some machines fail.
However, it is designed for **batch processing** — it analyzes data collected over time, not live data.
Since MapReduce reads and writes to disk often, it can be **slow** for tasks that need many repeated steps, like training machine learning models.
For example, Hadoop is ideal for creating **monthly sales reports** from millions of records processed in one big batch.

Apache Spark: Fast and Flexible Processing

Development and Origin

spark, as a data processing framework, was initially developed by Matei Zaharia at **UC Berkeley's AMPLab** in **2009**.

Open Source

In **2010**, the project became an **open-source** project under a Berkeley software distribution license.

Text sample here

- In **2013**, the project was donated to the **Apache Software Foundation**.
- The license was changed to **Apache 2.0**.
- In February **2014**, Spark became an **Apache top-level project**.

Text sample here

DataBricks and World Record: By November **2014**, Spark was used by the engineering team at **DataBricks** (a company founded by Spark's creators) to set a **world record in large-scale sorting**.

Current ly

At present, Spark exists as a **Next Generation real-time and batch processing framework**. DataBricks now provides commercial support and certification for taking the Spark programming test.

Batch Processing

- **Data Volume and Timing:** A large amount of data or transactions are processed in a **single run over a time period**.
- **Execution:** Associated jobs generally run entirely **without any manual intervention**.
- **Data Input:** The entire data set is **pre-selected** and fed using command line parameters and scripts.
- **Use Cases (Offline Workflow):** It is used to execute multiple operations, handle heavy data loads, generate reports, and manage data workflow that is **offline**.
- **Example:** Creating **daily or hourly reports** to aid decision-making.

Real-Time Processing

Real-time processing is defined in contrast to batch processing:

- **Timing:** It occurs **instantaneously** on data entry or command receipt.
- **Execution Constraint:** It needs to execute within **stringent response time constraints**.
- **Example: Fraud detection**.

Limitation of MapReduce

Performance & Processing:

- **Not real-time suitable:** Too slow for instant responses; designed for batch processing.
- **Slow due to disk I/O:** Writes intermediate data to disk, causing high latency.
- **Inefficient for small files:** Struggles with many small files.

Rigidity & Complexity:

- **Inflexible workflow:** Fixed two-phase model hinders complex, multi-step, or iterative jobs (e.g., machine learning).
- **Challenging for certain problems:** Difficult for non-partitionable problems or multi-input operations like joins.

Development & Operations:

- **Complex programming:** Requires extensive coding and deep understanding of the framework, even for simple tasks.

SO…

Definition: Spark is an **open-source cluster computing framework**.

Purpose: Spark is designed as a **next-generation framework** for both **real-time** and **batch processing**.

Key Advantage (over MapReduce):

- It was created to **address the limitations of MapReduce**.
- It can provide **up to 100 times faster performance** than MapReduce for certain applications.
- This speed comes from its use of **in-memory primitives**, making it particularly suitable for:
  - Iterative execution
  - Graph processing
  - Real-time processing
  - Machine learning.

Ecosystem of Libraries: Spark comes with a rich set of libraries that extend its capabilities:

- **Spark core**: Spark utilizes **Resilient Distributed Datasets (RDDs)**, which are logically partitioned collections of data.
- **Spark SQL:** For structured data processing using SQL queries.
- **Spark Streaming:** For processing live streams of data.
- **MLlib:** A scalable machine learning library.
- **GraphX:** For graph-parallel computation.

Spark Component:

Spark Core

•The **foundation** of the entire framework.

•Handles **basic functions** like:

- Task scheduling
- Memory management
- Fault tolerance
- Interacting with storage systems (HDFS, S3, etc.)

•Provides the main abstraction: **RDD (Resilient Distributed Dataset)** — a fault-tolerant collection of data for parallel processing.

Spark SQL

- Used for processing **structured and semi-structured data**.
- Lets you use **SQL queries** alongside regular Spark code.
- Introduces **DataFrames** and **Datasets**, which are optimized, tabular-like data structures.
- Works well with data stored in Parquet, ORC, Hive, or relational databases.

Spark Streaming

- Designed for **real-time data processing**.
- Processes data streams (e.g., logs, sensor data, or tweets) in small batches almost instantly.
- Now extended by **Structured Streaming**, which uses the same DataFrame API as Spark SQL.

MLlib (Machine Learning Library)

•Spark's **built-in machine learning** library.

•Provides scalable algorithms for:

- Classification
- Regression
- Clustering
- Collaborative filtering
- Dimensionality reduction

•Includes tools for feature extraction and evaluation.

GraphX

•A library for **graph processing and analysis** (networks of nodes and edges).

•Supports operations like PageRank, connected components, and shortest paths.

•Lets you combine graph computation with standard Spark operations.

An everyday example of Spark's capabilities can be found in services like

Netflix or 

Spotify, 

where user behavior is analyzed continuously to generate personalized recommendations in near real time.

Flink: Real-Time, Event-Driven Processing

What is Apache Flink?

Apache Flink is an **open-source framework** for **stream processing** and **batch processing** of big data.

It was designed to handle **data in motion** — that is, data that's continuously produced (like sensor readings, financial transactions, or social media feeds).

Flink can process **massive, continuous data streams** in **real time**, with **high accuracy and low latency**.

History of Apache Flink

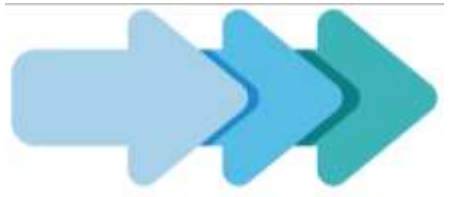
- **Origin:** Started as a research project called **Stratosphere** at the Technical University of Berlin.
- **2014:** Donated to the Apache Software Foundation and renamed **Apache Flink**.
- **2015:** Graduated as a **top-level Apache project**.
- Since then, it's become one of the leading tools for **real-time stream processing**, used by companies like Alibaba, Netflix, and Uber.

Unlike Spark, which was first built for batch processing and later extended for streams, **Flink was built for streaming from the start.**
Batch processing is just a **special case of streaming** in Flink's design.

Main Features

- **True Stream Processing:** Processes events one by one as they arrive (not in micro-batches).
- **Low Latency:** Real-time reaction to data with very little delay.
- **Exactly-Once Semantics:** Ensures accurate results even if system failures occur.
- **Event-Time Processing:** Handles data based on when events actually happened, not when they were received.
- **Scalability and Fault Tolerance:** Can run on large clusters and recover automatically from failures.

Main Components



Component	Description
Flink Core	Manages distributed execution and communication.
DataStream API	For **stream processing** (real-time data).
DataSet API	For **batch processing** (static data).
Table API & SQL	Unified interface for querying both streams and batches.
Flink ML	Library for machine learning on streaming data.
CEP (Complex Event Processing)	Detects event patterns (e.g., fraud detection).

Comparing the Three Frameworks

To summarize their differences:

Feature	**Hadoop (MapReduce)**	**Spark**	**Flink**
Processing style	Batch only	Batch + micro-batch streaming	Batch + true streaming
Speed	Slower (disk-based)	Fast (in-memory)	Real-time, very low latency
Ecosystem focus	Storage + processing	General-purpose analytics	Real-time event processing
Typical use cases	Log analysis, batch jobs	Machine learning, SQL, batch + streaming apps	IoT, fraud detection, real-time systems

References

1. Apache Hadoop. (n.d.). *HDFS Architecture Guide.* Retrieved September 26, 2025, from https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
2. Amazon Web Services. (n.d.). *Amazon S3 Overview.* Retrieved September 26, 2025, from https://aws.amazon.com/s3/
3. Databricks. (n.d.). *Data Lake vs Lakehouse.* Retrieved September 26, 2025, from https://www.databricks.com/glossary/data-lakehouse
4. MongoDB. (n.d.). *Introduction to MongoDB.* Retrieved September 26, 2025, from https://www.mongodb.com/nosql-explained
5. Apache Cassandra. (n.d.). *Getting Started with Cassandra.* Retrieved September 26, 2025, from https://cassandra.apache.org/doc/latest/
6. Neo4j. (n.d.). *Graph Database Basics.* Retrieved September 26, 2025, from https://neo4j.com/developer/graph-database/
7. Redis. (n.d.). *Redis Documentation.* Retrieved September 26, 2025, from https://redis.io/docs/
8. IBM. (n.d.). *Hadoop vs Spark vs Flink.* Retrieved September 26, 2025, from https://www.ibm.com/blog/hadoop-vs-spark-vs-flink
9. Apache Spark. (n.d.). *Official Documentation.* Retrieved September 26, 2025, from https://spark.apache.org/docs/latest/
10. Apache Flink. (n.d.). *Official Documentation.* Retrieved September 26, 2025, from https://nightlies.apache.org/flink/flink-docs-release-1.17/