Course Support: Big Data

Chapter 1 + 2 +3

Dr. Brahim Benabderrahmane

Academic Year 2025/2026

Contents

Chapter 1

Introduction to Big Data

1.1 Introduction: The Era of Data Explosion

We are currently witnessing a digital revolution characterized by a massive explosion in data generation. In the past decades, data creation was primarily the domain of human operators entering text into systems manually. Today, the paradigm has shifted significantly: machines, sensors, and automated processes now generate the vast majority of digital information, operating continuously without human intervention.

Traditional Relational Database Management Systems (RDBMS), such as Oracle or MySQL, have served as the backbone of enterprise storage for over forty years. However, these systems are becoming increasingly insufficient for modern requirements. They struggle to handle the sheer **volume** of data generated by the web, the **velocity** at which it arrives, and the **variety** of unstructured formats it takes. This fundamental limitation has given rise to the discipline of **Big Data**.

Definition

Big Data is a term applied to data sets whose size or type is beyond the ability of traditional relational databases to capture, manage, and process with low-latency. It requires new forms of processing to enable enhanced decision making, insight discovery, and process optimization [2].

1.2 Data Sources: The Drivers of Growth

The exponential growth of data is not accidental; it is driven by three primary categories of sources that have emerged and matured over the last decade.

1.2.1 1. The Social Web (Human-Generated)

This category encompasses the massive digital footprint left by human interaction online. Unlike the early web, which was read-only, the modern social web allows every user to be a content creator. Every click, like, share, comment, and upload contributes to a continuous stream of unstructured data.

- **Search Engines:** Google processes over **99,000 queries every second**. Each query represents user intent, location, and context.
- **Video Content:** YouTube users upload over **500 hours of video per minute**. This video data is opaque to traditional databases and requires complex processing to index.
- **Social Media:** Platforms like TikTok, Instagram, and X (Twitter) generate petabytes of media and textual data daily, creating a real-time pulse of global sentiment.

1.2.2 2. The Internet of Things (Machine-Generated)

Machine-generated data is growing faster than human data because machines do not sleep. The ”Internet of Things” (IoT) refers to the network of physical objects embedded with sensors and software.

- **Sensors:** Industrial sensors in factories (Industry 4.0) monitor machinery health 24/7, predicting failures before they occur.
- **Telemetry:** Modern logistics fleets and smartphones transmit GPS coordinates, speed, and environmental data constantly.
- **Health Trackers:** Wearable devices monitor biological metrics like heart rate and sleep patterns, creating longitudinal health datasets for millions of users.

1.2.3 3. Transactional Data

While traditional, this category has scaled up significantly due to the digitization of the economy. It includes e-commerce records (Amazon, AliExpress), high-frequency banking transactions, and digital payment logs. Unlike social data, this data is highly structured and requires strict consistency (ACID properties).

1.3 The 5 Vs of Big Data

To characterize a dataset as ”Big Data,” we often refer to the 5 Vs model. This model was originally proposed as the ”3 Vs” by Doug Laney in 2001 [1] and later expanded by IBM and others.

1.3.1 1. Volume (Scale)

Volume refers to the magnitude of data generation and storage. We have transitioned from the Gigabyte/Terabyte era to the Petabyte (PB) and Exabyte (EB) era. The implication of this volume is physical: a single hard drive or even a single server's RAID array cannot store this amount of information. This necessitates the use of **Distributed File Systems** (like HDFS), which spread data across hundreds or thousands of commodity disks to bypass physical storage limits.

1.3.2 2. Velocity (Speed)

Velocity describes the rate at which data is generated and the speed at which it must be processed to remain valuable.

- **Batch Processing:** Dealing with data at rest. For example, generating end-of-day banking reports. The data is collected, stored, and processed hours later.
- **Stream Processing:** Dealing with data in motion. For example, fraud detection algorithms must analyze a credit card swipe in milliseconds to approve or decline the transaction. High velocity demands systems that can ingest data with low latency (e.g., Apache Kafka).

1.3.3 3. Variety (Complexity)

Data is no longer just rows and columns. Variety refers to the structural heterogeneity of modern datasets.

- **Structured:** Relational data fitting neatly into SQL tables (e.g., financial ledgers).
- **Semi-Structured:** Data with tags or markers but no rigid schema. Examples include JSON, XML, and System Logs. This requires flexible parsing capabilities.
- **Unstructured:** Data with no internal structure identifiable by standard algorithms. Examples include Images, Audio, Video, and PDF files. This accounts for approximately 80% of modern data and requires advanced processing (e.g., Computer Vision, NLP) to extract meaning.

1.3.4 4. Veracity (Trustworthiness)

This refers to the quality, accuracy, and uncertainty of data. Big Data is inherently "messy." It contains noise, typos, missing values, sensor errors, and biases. Unlike small, curated datasets, Big Data cannot be manually verified. Significant computational effort must be spent on data cleaning and normalization before analysis can begin.

1.3.5 5. Value (Utility)

Added later to the model, Value is arguably the most important dimension. Collecting petabytes of data is a liability (in terms of storage and maintenance costs) unless actionable insights can be extracted. The ultimate goal of any Big Data initiative is to turn raw data into business value, such as predictive maintenance, personalized recommendations, or optimized logistics.

1.4 The Hardware Bottleneck: Memory Hierarchy

To understand why Big Data systems like Hadoop are designed the way they are, we must look at the limitations of computer architecture, specifically the Memory Hierarchy.

1.4.1 The Latency Gap

There is a massive speed difference between the CPU and Storage mechanisms:

1. **CPU Registers/Cache:** Access takes nanoseconds. This is where computation happens.
2. **RAM (Main Memory):** Access takes tens of nanoseconds. It is fast but volatile and expensive per gigabyte.
3. **Disk (HDD/SSD):** Access takes milliseconds. While SSDs have improved this, storage remains the slowest component of the stack.

1.4.2 The I/O Bottleneck and Parallelism

In Big Data scenarios, datasets are too large to fit in RAM. We are forced to use Disk storage. However, disks are physically slow due to mechanical limitations (seek time) or interface limits.

- Reading 1 TB from a single modern disk at 100 MB/s takes approximately **2.5 hours**. This is unacceptable for analytics.
- **The Solution (Parallelism):** If we split that 1 TB file into small chunks and spread them across 100 disks, and read them simultaneously, the total read time drops to approximately **1.5 minutes**.

This principle of **Parallel I/O** is the foundational concept behind Google's GFS [10] and Hadoop's HDFS.

1.5 Scaling Strategies

When a database server reaches 100% CPU utilization or fills its storage capacity, administrators face two fundamental choices for growth.

1.5.1 Vertical Scaling (Scale Up)

Vertical scaling involves adding more resources to the existing single node—upgrading from 16GB to 128GB RAM, or buying a CPU with more cores.

Advantages: The primary advantage is simplicity. The software architecture does not need to change; the database remains on a single machine, preserving ACID transactions and data consistency without complex network logic.

Disadvantages: Vertical scaling suffers from an exponential cost curve; high-end hardware is disproportionately expensive. Furthermore, it has a hard physical limit—one cannot simply buy a CPU with infinite cores. Finally, it represents a Single Point of Failure (SPOF); if that one massive server fails, the entire application goes offline.

1.5.2 Horizontal Scaling (Scale Out)

Horizontal scaling involves adding more machines (nodes) to the system to form a distributed cluster. Instead of one super-computer, we use ten commodity computers working together.

Advantages: This approach offers linear scalability; to double capacity, one simply adds double the nodes. It is cost-effective as it uses standard hardware. Crucially, it offers high fault tolerance; if one node fails, the system can redirect work to the surviving nodes.

Disadvantages: The complexity shifts from hardware to software. Developers must manage data distribution, synchronization, network latency, and consistency across the cluster. This necessitates the use of specialized distributed systems like Hadoop or Spark.

1.6 Big Data Disciplines

The Big Data ecosystem is divided into specialized disciplines to handle different aspects of the distributed problem:

1.6.1 1. Distributed Storage (HDFS)

The first challenge is simply storing the data. Distributed file systems break large files into blocks (chunks) and distribute them across multiple nodes. This ensures that no single disk becomes a bottleneck for storage or throughput.

1.6.2 2. High Performance Computing (HPC)

This discipline uses supercomputers or GPU clusters for massive computational tasks. Unlike Big Data, which is often I/O bound (waiting for data), HPC is typically CPU/GPU bound. It is used for weather forecasting, deep learning training, and complex simulations.

1.6.3 3. NoSQL Databases

Since SQL databases struggle with horizontal scaling and unstructured data, a new generation of databases emerged.

Document Stores (e.g., MongoDB): These databases store data as flexible JSON-like documents. They are ideal for Content Management Systems and catalogs where the schema may change frequently.

Key-Value Stores (e.g., Redis): These act as massive distributed hash maps. They are optimized for speed and are commonly used for caching user sessions and real-time leaderboards.

Column-Family Stores (e.g., Cassandra): These store data by columns rather than rows. This architecture is optimized for write-heavy workloads and analytical queries over large datasets (e.g., time-series data).

Graph Databases (e.g., Neo4j): These are optimized for storing relationships (nodes and edges) as first-class citizens. They are essential for social networks, recommendation engines, and fraud detection systems where the connection between data points is more important than the data points themselves.

Chapter 2

Hadoop Systems

2.1 Introduction and Motivation

As discussed in the previous chapter, the volume of data generated globally has outpaced the capabilities of vertical scaling. Traditional storage systems (SAN/NAS) and processing paradigms are prohibitively expensive and difficult to scale to the Petabyte range.

To address this, Doug Cutting and Mike Cafarella created **Apache Hadoop** in 2006, inspired by Google's white papers on the Google File System (GFS) and MapReduce. Hadoop is an open-source software framework designed for distributed storage and processing of very large datasets on computer clusters built from commodity hardware.

Definition: Commodity Hardware

The term "commodity hardware" does not imply low-quality components. Rather, it refers to standardized, affordable, and widely available server hardware (e.g., standard x86 servers) as opposed to specialized, proprietary supercomputers or mainframes. Hadoop achieves reliability through software replication, not through expensive hardware redundancy.

The core philosophy of Hadoop is to bring the **computation to the data**, rather than moving the data to the computation. In traditional HPC, data is moved across the network to the CPU, which creates a bottleneck. In Hadoop, the code is small enough to be sent to the node where the data resides, minimizing network congestion.

2.2 Hadoop Distributed File System (HDFS)

HDFS is the storage layer of Hadoop. It is designed to store vast amounts of data reliably even when hardware failures are the norm, not the exception.

2.2.1 Architecture Overview

HDFS follows a **Master/Slave** architecture:

- **NameNode (Master):** This is the centerpiece of an HDFS file system. It keeps the directory tree of all files in the file system, and tracks where across the cluster the file data is kept. It stores metadata (filenames, permissions, modification times) in RAM for fast access.
- **DataNode (Slave):** There are typically many DataNodes per cluster. They store the actual data. A DataNode manages the storage attached to the node that it runs on. It performs read and write requests from the file system's clients and performs block creation, deletion, and replication upon instruction from the NameNode.

2.2.2 Data Blocks and Replication

Unlike a standard OS file system where block sizes are 4KB or 8KB, HDFS uses a very large block size, typically **128 MB** (or 256 MB in modern setups).

Why large blocks? The goal is to minimize the cost of seeks. If a block is large enough, the time to transfer the data from the disk can be significantly longer than the time to seek to the start of the block. This makes HDFS optimized for throughput (streaming large files) rather than latency (random access to many small files).

Replication Strategy: To ensure fault tolerance, HDFS replicates each block multiple times (default is 3). The standard placement policy is:

1. The first replica is placed on the local node writing the data.
2. The second replica is placed on a different rack to protect against rack failure (e.g., switch failure).
3. The third replica is placed on a different node in the same rack as the second replica to minimize network traffic.

2.3 The MapReduce Paradigm

MapReduce is the processing layer of the original Hadoop stack. It is a programming model for processing large data sets with a parallel, distributed algorithm on a cluster.

2.3.1 The Philosophy

MapReduce abstracts away the complexity of distributed computing. The developer only needs to write two functions:

- **Map:** Processes input key/value pairs to generate a set of intermediate key/value pairs.
- **Reduce:** Merges all intermediate values associated with the same intermediate key.

[Image of MapReduce data flow diagram]

2.3.2 Execution Flow

The execution of a MapReduce job typically follows these stages:

1. **Input Splits:** The input file is split into chunks (usually equal to the HDFS block size).
2. **Mapping:** The Map function runs on every node containing a data block. This exploits data locality. The output is buffered in memory.
3. **Shuffling and Sorting:** This is the ”magic” phase. The framework moves the output of the mappers across the network to the reducers. Crucially, the data is **sorted by key**. All values for ”Key A” are guaranteed to arrive at the same Reducer.
4. **Reducing:** The Reduce function iterates over the sorted values, aggregating them (e.g., summing numbers, concatenating strings) to produce the final output.
5. **Output:** The result is written back to HDFS.

2.4 Hadoop Architecture Evolution: From 1.0 to YARN

The original Hadoop (version 1.0) had a significant limitation: the MapReduce engine was tightly coupled with resource management. A component called the **JobTracker** managed both the scheduling of tasks and the monitoring of resources. This created a scalability bottleneck (the JobTracker became overwhelmed around 4,000 nodes) and restricted Hadoop to only running MapReduce jobs.

2.4.1 Hadoop 2.0 and YARN

To solve this, Hadoop 2.0 introduced **YARN (Yet Another Resource Negotiator)**. YARN separates the resource management layer from the processing layer.

ResourceManager (Global Master): It arbitrates resources among all the applications in the system. It is purely a scheduler.

NodeManager (Per-Node Slave): It is responsible for containers, monitoring their resource usage (CPU, memory, disk, network) and reporting the same to the ResourceManager.

ApplicationMaster (Per-Application): A specific library that negotiates resources from the ResourceManager and works with the NodeManager(s) to execute and monitor the tasks.

This architecture allows Hadoop to support multiple processing engines simultaneously. Today, a Hadoop cluster can run MapReduce, Apache Spark, and Apache Flink side-by-side, all sharing the same HDFS storage and managed by YARN.

2.5 The Hadoop Ecosystem

Hadoop is rarely used in isolation. An entire ecosystem has developed around it to solve specific problems:

- **Apache Hive:** Provides a SQL-like interface (HQL) to query data stored in HDFS. It translates SQL queries into MapReduce or Tez jobs, making Hadoop accessible to analysts who do not know Java.
- **Apache Pig:** A high-level platform for creating MapReduce programs used with Hadoop. The language used for this platform is called Pig Latin.
- **Apache HBase:** A distributed, scalable, big data store. It allows random, real-time read/write access to your Big Data, modeled after Google's BigTable.
- **Apache Zookeeper:** A centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services.
- **Apache Sqoop:** A tool designed for efficiently transferring bulk data between Apache Hadoop and structured datastores such as relational databases.

Chapter 3

Apache Spark

3.1 Introduction and Motivation

While Hadoop MapReduce revolutionized data processing by allowing massive scalability on commodity hardware, it was not without flaws. MapReduce was designed primarily for batch processing, where the intermediate results of every computation step are written to the hard disk. For iterative algorithms, common in machine learning and graph processing, this constant Disk I/O became a severe performance bottleneck.

Apache Spark, developed at UC Berkeley's AMPLab in 2009, was created to address these limitations. It is a unified analytics engine for large-scale data processing that performs computation in-memory (RAM) rather than on disk.

Why Spark?

Spark is often reported to be **100x faster** than MapReduce for in-memory operations and 10x faster on disk. Beyond speed, it offers a unified stack: developers can mix SQL queries, streaming, and machine learning in the same application using high-level APIs in Python (PySpark), Scala, Java, and R.

3.2 Core Concepts and Architecture

Spark follows a master-slave architecture similar to Hadoop but with different terminology and execution flow.

3.2.1 The Spark Application Architecture

A Spark application consists of a **Driver Program** and multiple **Executors** distributed across the cluster.

- **Driver:** The process running the `main()` function of the application. It maintains information about the SparkContext, converts the user's code into tasks, and schedules them on executors.

- **Cluster Manager:** An external service (like Hadoop YARN, Kubernetes, or Apache Mesos) that allocates resources to the application.
- **Executors:** Processes running on the worker nodes. They are responsible for executing the actual tasks and storing data in memory or disk.

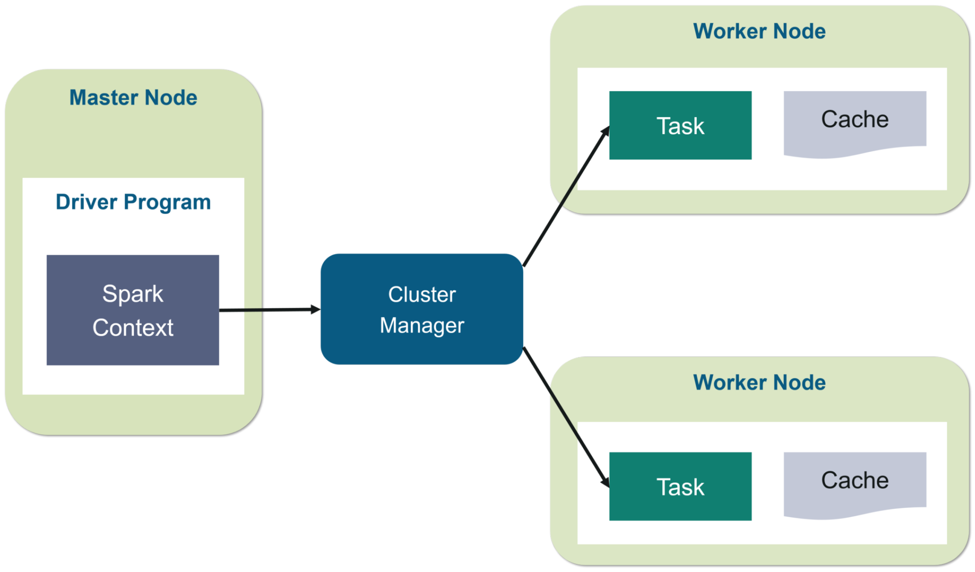

Figure 3.1: Spark Architecture: The Driver coordinates tasks running on Executors.

3.2.2 Resilient Distributed Datasets (RDDs)

The fundamental data structure of Spark is the **RDD**. It is an immutable distributed collection of elements of your data, partitioned across nodes in your cluster that can be operated on in parallel.

- **Resilient:** If a node fails, Spark can reconstruct the lost data partitions by replaying the transformation history (Lineage).
- **Distributed:** Data is split into chunks and stored across the cluster.
- **Dataset:** It can hold any type of data (strings, objects, integers).

3.3 The Execution Model: DAG and Lazy Evaluation

One of Spark's key innovations is its execution engine, which is based on a **Directed Acyclic Graph (DAG)**.

3.3.1 Transformations vs. Actions

Spark operations are divided into two categories:

Transformations (Lazy): Operations that define a new RDD from an existing one (e.g., `map`, `filter`, `groupBy`). When you call a transformation, Spark *does not* execute it immediately. It simply records the operation in the lineage graph.

Actions (Eager): Operations that trigger computation and return a result to the driver or write data to storage (e.g., `count`, `collect`, `saveAsTextFile`).

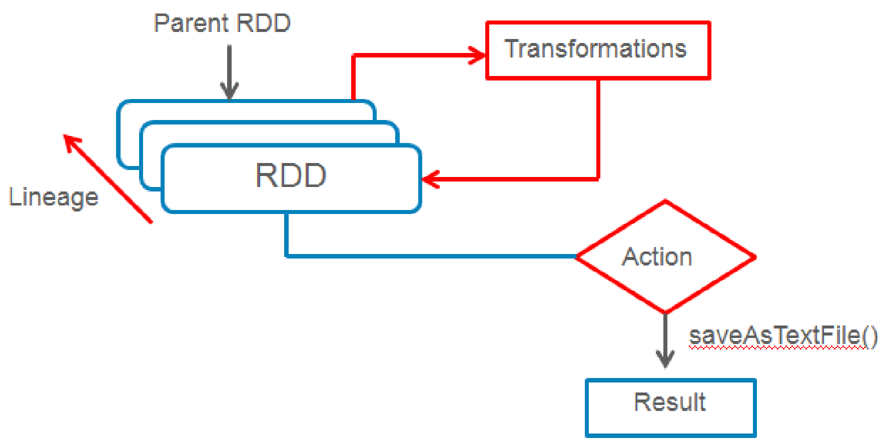

Figure 3.2: Lazy Evaluation: Spark builds a plan (DAG) and only executes when an Action is called.

3.3.2 Benefits of Lazy Evaluation

Lazy evaluation allows Spark to optimize the entire data flow before execution. For example, if you filter a 1TB dataset and then ask for only the top 10 rows, Spark can optimize the read operation to stop early, rather than processing the entire terabyte first.

3.4 Structured Data: DataFrames and Spark SQL

While RDDs offer low-level control, modern Spark development relies on **DataFrames**. A DataFrame is conceptually equivalent to a table in a relational database or a data frame in Python/R, but with richer optimizations under the hood.

3.4.1 The Catalyst Optimizer

When you write code using DataFrames or Spark SQL, Spark uses the **Catalyst Optimizer** to determine the most efficient execution plan. It can reorder operations, optimize joins, and prune unnecessary columns automatically—optimizations that are difficult to achieve with raw RDDs.

Listing 3.1: PySpark DataFrame Example

```python
# PySpark Example
df = spark.read.json("logs.json")

# Filter and Group By (Optimized by Catalyst)
errors = df.filter(df['status'] == 'ERROR') \
            .groupBy('date') \
            .count()

errors.show()
```

3.5 Machine Learning with MLlib

Spark includes **MLlib**, a library designed to make practical machine learning scalable. Unlike single-node libraries (like Scikit-Learn), MLlib algorithms are designed to run in parallel across a cluster.

3.5.1 Key Components of MLlib

MLlib standardizes APIs for machine learning algorithms to make it easier to combine multiple algorithms into a single pipeline.

- **Transformer:** An algorithm which can transform one DataFrame into another DataFrame. E.g., an ML model is a Transformer which transforms a DataFrame with features into a DataFrame with predictions.
- **Estimator:** An algorithm which can be fit on a DataFrame to produce a Transformer. E.g., a learning algorithm is an Estimator which trains on a DataFrame and produces a model.
- **Pipeline:** A chain of multiple Transformers and Estimators specified as an ML workflow.

3.5.2 Feature Engineering and Preprocessing

Before training a model, data must be prepared. MLlib provides powerful feature transformers:

1. VectorAssembler

Most ML algorithms in Spark require a single ”features” column containing a vector of all input variables. `VectorAssembler` merges multiple columns into this single vector column.

```python
from pyspark.ml.feature import VectorAssembler

assembler = VectorAssembler(
    inputCols=["age", "salary", "hours_worked"],
    outputCol="features"
)
output_df = assembler.transform(df)
```

2. StringIndexer

Encodes a string column of labels to a column of label indices. This is necessary for handling categorical variables (e.g., converting ”Yes”/”No” to 1.0/0.0).

```python
from pyspark.ml.feature import StringIndexer

indexer = StringIndexer(inputCol="category", outputCol="categoryIndex")
indexed_df = indexer.fit(df).transform(df)
```

3. StandardScaler

Standardizes features by removing the mean and scaling to unit variance. This is critical for algorithms like Support Vector Machines (SVM) or Logistic Regression which are sensitive to the scale of input data.

3.5.3 Example: Building a Logistic Regression Pipeline

Here is a complete example of predicting a binary outcome (e.g., churn) using a pipeline.

Listing 3.2: Logistic Regression Pipeline

```python
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import VectorAssembler

# 1. Prepare Data
# Assume 'df' has columns: 'age', 'balance', 'churn' (0 or 1)
train_data, test_data = df.randomSplit([0.7, 0.3])

# 2. Define Stages
# Stage 1: Assemble features into a vector
```

```
assembler = VectorAssembler(
    inputCols=["age", "balance"],
    outputCol="features"
)

# Stage 2: Define the Model (Estimator)
lr = LogisticRegression(labelCol="churn", featuresCol="features")

# 3. Build Pipeline
pipeline = Pipeline(stages=[assembler, lr])

# 4. Train Model
model = pipeline.fit(train_data)

# 5. Make Predictions
predictions = model.transform(test_data)
predictions.select("churn", "prediction", "probability").show(5)
```

3.6 Spark in the Modern Landscape

Today, Spark has evolved beyond a simple processing engine into a unified ecosystem known as the ”Lakehouse” foundation.

3.6.1 Spark vs. Flink

While Spark is dominant, **Apache Flink** is another major player. The key difference lies in their streaming model:

- **Spark Streaming:** Uses ”Micro-batching.” It collects data for a small window (e.g., 1 second) and processes it as a batch. It offers high throughput but slightly higher latency.
- **Flink:** Uses ”True Streaming.” It processes every event as it arrives. It offers ultra-low latency but can be more complex to manage.

3.6.2 Future Trends

Spark continues to evolve with version 3.x and 4.x, focusing on:

1. **Kubernetes Integration:** Native support for running Spark on K8s clusters.
2. **Delta Lake:** Adding ACID transactions to Spark data lakes, solving the problem of data reliability in distributed systems.

3. **Serverless Spark:** Cloud providers (AWS, Databricks) offering Spark as a service where users don't manage clusters at all.

Bibliography

[1] Laney, D. (2001). *3D Data Management: Controlling Data Volume, Velocity and Variety*. META Group Research Note, 6(70).

[2] Gartner IT Glossary. *Big Data*. Available online at: https://www.gartner.com/it-glossary/big-data/

[3] Ghemawat, S., Gobioff, H., & Leung, S. T. (2003). *The Google File System*. ACM SIGOPS Operating Systems Review, 37(5), 29-43.

[4] Dean, J., & Ghemawat, S. (2004). *MapReduce: Simplified Data Processing on Large Clusters*. Communications of the ACM, 51(1), 107-113.

[5] Humby, C. (2006). *Data is the New Oil*. ANA Senior Marketer's Summit, Kellogg School.

[6] Zaharia, M., et al. (2010). *Spark: Cluster Computing with Working Sets*. USENIX Workshop on Hot Topics in Cloud Computing (HotCloud).

[7] Karau, H., Konwinski, A., Wendell, P., & Zaharia, M. (2015). *Learning Spark: Lightning-Fast Big Data Analysis*. O'Reilly Media.

[8] Chambers, B., & Zaharia, M. (2018). *Spark: The Definitive Guide*. O'Reilly Media.

[9] Armbrust, M., et al. (2015). *Spark SQL: Relational Data Processing in Spark*. ACM SIGMOD International Conference on Management of Data.

[10] Ghemawat, S., Gobioff, H., & Leung, S. T. (2003). *The Google File System*. ACM SIGOPS Operating Systems Review.

[11] Dean, J., & Ghemawat, S. (2004). *MapReduce: Simplified Data Processing on Large Clusters*. Communications of the ACM.

[12] White, T. (2015). *Hadoop: The Definitive Guide*. O'Reilly Media, Inc.

[13] Vavilapalli, V. K., et al. (2013). *Apache Hadoop YARN: Yet Another Resource Negotiator*. ACM Symposium on Cloud Computing.