Big Data Technologies and Cloud Computing for Data Science Analytics

Shubneet [1], Anushka Raj Yadav [2], Navjot Singh Talwandi [3]

[1,2,3*]Department of Computer Science, Chandigarh University, Gharuan, Mohali, 140413, Punjab, India.

Contributing authors: jeetshubneet27@gmail.com; ay462744@gmail.com; navjot.e17908@cumail.in;

Abstract

Big Data refers to extremely large, complex datasets that challenge traditional data processing systems, characterized by the four Vs: **Volume** (sheer data size), **Velocity** (rapid data generation/ingestion), **Variety** (diverse formats like structured, unstructured, and semi-structured data), and **Veracity** (data quality and reliability). Managing these datasets poses significant challenges in storage, distributed processing, and scalability, necessitating specialized tools such as Hadoop's **HDFS** for distributed storage, **MapReduce** for batch processing, and Spark for in-memory analytics. Modern solutions leverage **distributed computing** frameworks and **NoSQL** databases (e.g., MongoDB, Cassandra) to handle heterogeneity and scale. Cloud platforms like AWS and Azure further address these challenges through elastic resources and managed services (e.g., AWS EMR, Azure HDInsight), enabling efficient **data pipeline** orchestration. However, organizations must still navigate trade-offs between consistency, availability, and partition tolerance (CAP theorem) in distributed systems. Emerging advancements in real-time stream processing (e.g., Apache Flink) and hybrid cloud architectures continue to reshape Big Data ecosystems, driving innovation in sectors from healthcare to finance. [1, 2]

Keywords: big data, cloud computing, distributed systems, data pipelines, NoSQL

1 Introduction

The exponential growth of data generation from IoT devices, social media platforms, and AI-driven applications has necessitated a paradigm shift from traditional relational databases to modern Big Data ecosystems. Early relational database management systems (RDBMS), such as Oracle and MySQL, excelled at structured data storage and transactional consistency but struggled to scale with the volume, velocity, and variety of data produced in the digital age. The rise of distributed systems, cloud computing, and real-time analytics has redefined how organizations store, process, and derive value from data, giving birth to technologies like Hadoop, Spark, and NoSQL databases [3]. These tools address the limitations of traditional systems through horizontal scalability, fault tolerance, and support for unstructured data, enabling applications ranging from real-time fraud detection to personalized recommendation engines.

Three key drivers underpin this evolution:

- **IoT and Sensor Data**: Billions of connected devices generate continuous streams of telemetry data, demanding scalable storage and low-latency processing.
- **Social Media**: Platforms like Facebook and Twitter produce petabytes of unstructured text, images, and video, requiring distributed processing frameworks.
- **AI/ML Workloads**: Training deep learning models on massive datasets necessitates parallelized computation and efficient resource orchestration.

Central to this transformation are two foundational concepts: the **CAP theorem** and **Lambda architecture**. The CAP theorem posits that distributed systems can only simultaneously guarantee two of three properties: consistency, availability, and partition tolerance [4]. This trade-off has shaped the design of NoSQL databases like Cassandra (prioritizing availability) and MongoDB (emphasizing consistency). Meanwhile, the Lambda architecture reconciles batch and stream processing by maintaining separate "cold" (batch) and "hot" (real-time) data paths, ensuring both comprehensive analytics and low-latency insights.

Chapter Outline

This chapter explores the technological and conceptual pillars of Big Data ecosystems:

- Fundamentals of Big Data: Characteristics (4Vs) and challenges
- Core technologies: Hadoop, Spark, Hive, and NoSQL databases
- Distributed storage (HDFS) and computing paradigms (MapReduce)
- Cloud platforms (AWS, Azure, GCP) and managed services
- Data pipeline design principles and orchestration tools
- Real-world case study: Retail analytics at scale
- Hands-on exercises and framework comparisons

As organizations increasingly adopt hybrid cloud architectures and decentralized data meshes, understanding these components becomes critical for building scalable, resilient data infrastructure. The following sections provide both theoretical frameworks and practical insights to navigate this complex landscape.

2 Hadoop, Spark, and Hive

The Hadoop ecosystem is foundational for Big Data analytics, providing robust tools for distributed storage and processing. Its architecture comprises three core components: HDFS for scalable storage, YARN for resource management, and MapReduce for batch computation. HDFS splits large files into blocks distributed across DataNodes, managed by a central NameNode. YARN coordinates computational resources, allowing multiple processing engines to share the cluster.

Hadoop vs. Spark Processing

Hadoop's MapReduce framework processes data in batch mode, writing intermediate results to disk. This disk-based approach is reliable but incurs high latency, making it less suitable for iterative or interactive workloads. Apache Spark addresses these limitations with in-memory processing using Resilient Distributed Datasets (RDDs), enabling up to 100x faster execution for many analytics and machine learning tasks. Spark supports both batch and real-time streaming, making it versatile for modern data pipelines.

Hive: SQL on Hadoop

Hive brings SQL-like querying to Hadoop through HiveQL, translating queries into MapReduce or Tez jobs. Its Metastore manages schema and metadata, while its optimizer improves query execution. Hive is ideal for ETL, reporting, and data warehousing, allowing analysts to leverage familiar SQL syntax on massive datasets.

Spark Word Count Example

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("WordCount").getOrCreate()
text_rdd = spark.sparkContext.textFile("hdfs:///input.txt")
word_counts = (text_rdd
               .flatMap(lambda line: line.split())
               .map(lambda word: (word, 1))
               .reduceByKey(lambda a, b: a + b))
word_counts.saveAsTextFile("hdfs:///output")
spark.stop()
```

Hadoop Ecosystem Architecture

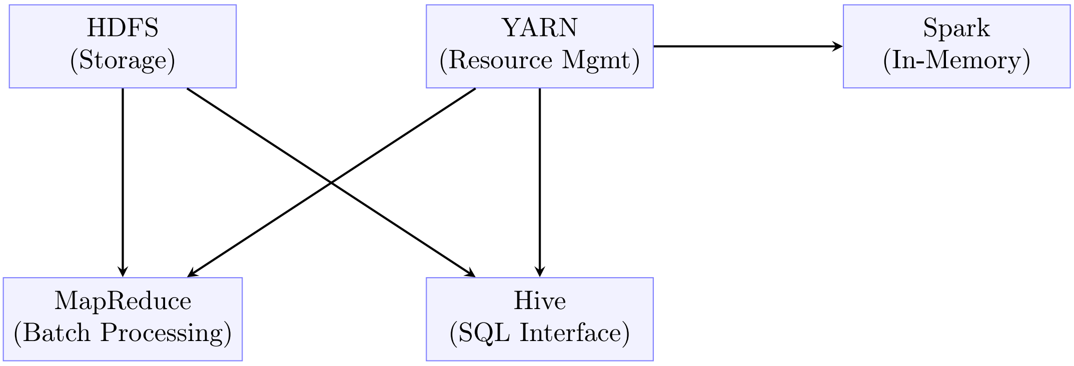


Fig. 1: Hadoop ecosystem architecture with core components

Technology Comparison

Table 1: Comparison of Hadoop, Spark, and Hive

Feature	**Hadoop**	**Spark**	**Hive**
Processing Model	Batch (MapReduce)	In-Memory	SQL-to-MapReduce
Latency	High (minutes+)	Low (seconds)	High (minutes+)
Data Types	Structured/Unstructured	All	Structured/Semi-structured
Real-Time Support	No	Yes (Streaming)	No
ML Support	Limited (Mahout)	MLlib	None
Storage Dependency	HDFS	Any	HDFS
Use Cases	ETL, batch analytics	ML, streaming, graph	Data warehousing, ETL

Hadoop, Spark, and Hive together form a flexible and scalable foundation for Big Data analytics, supporting a wide range of business and scientific applications [5, 6].

3 HDFS and NoSQL Databases

HDFS Replication and Fault Tolerance

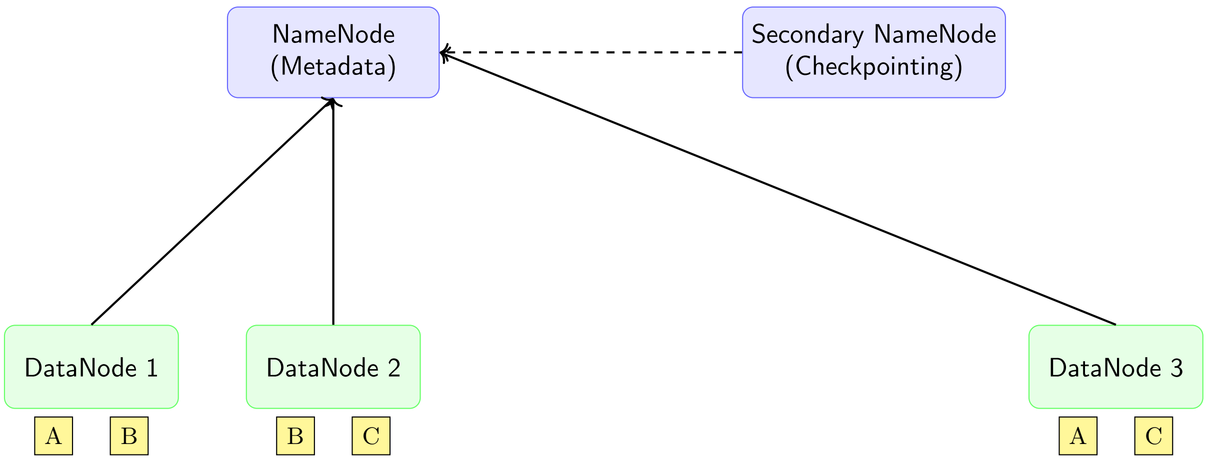
Hadoop Distributed File System (HDFS) ensures data durability through **block replication** and **erasure coding**. By default, HDFS stores 3 replicas of each data block across multiple DataNodes, providing fault tolerance against node failures. For example, if a file is split into blocks A, B, and C, replicas are distributed such that losing one DataNode does not compromise data accessibility [7].

Hadoop 3 introduced **erasure coding**, which splits data into fragments with parity information, reducing storage overhead by 50% while maintaining fault tolerance. This contrasts with replication, which triples storage usage. HDFS also automatically re-replicates blocks if nodes fail, maintaining the replication factor dynamically.

NoSQL Database Types

- **Document (MongoDB)**: Stores JSON-like documents with dynamic schemas. Ideal for content management and real-time analytics. Supports rich queries and aggregation pipelines.
- **Columnar (Cassandra)**: Organizes data into column families for high write throughput. Used in IoT and time-series data. Provides linear scalability and multi-datacenter support.
- **Key-Value (Redis)**: In-memory store for low-latency caching. Handles session management and leaderboards. Supports TTL (time-to-live) for automatic data expiration.

HDFS Architecture



Replication Factor = 2: Each block is stored on two DataNodes

Fig. 2: HDFS architecture: DataNodes store replicated blocks and report to the NameNode, which manages metadata. The Secondary NameNode provides checkpointing.

HDFS vs. NoSQL Comparison

MongoDB Aggregation Example

Table 2: HDFS vs. NoSQL Databases

Feature	**HDFS**	**NoSQL**
Consistency	Strong (via replication)	Eventual (Cassandra), Strong (MongoDB)
Scalability	Horizontal (add nodes)	Horizontal (sharding)
Query Support	MapReduce jobs	Domain-specific (CQL, HiveQL)
Data Model	File blocks	Document/Column/Key-Value
Use Case	Batch analytics	Real-time apps, caching

```
from pymongo import MongoClient

client = MongoClient("mongodb://localhost:27017/")
db = client["sales_db"]
pipeline = [
    {"$match": {"region": "North␣America"}},
    {"$group": {"_id": "$product", "total_sales": {"$sum": "
        $revenue"}}},
    {"$sort": {"total_sales": -1}}
]
results = db.sales.aggregate(pipeline)
for doc in results:
    print(doc)
```

HDFS and NoSQL databases address complementary needs in modern data architectures-HDFS for scalable storage and NoSQL for flexible data modeling [8].

4 Parallel and Distributed Processing

Modern Big Data ecosystems rely on parallel and distributed processing frameworks to handle large-scale computations efficiently across clusters. Two foundational paradigms-MapReduce and Spark's Resilient Distributed Datasets (RDDs)-demonstrate contrasting approaches to distributed computation.

MapReduce Workflow

The MapReduce framework processes data in three phases:

- **Map**: Processes input key-value pairs and emits intermediate pairs
- **Shuffle**: Transfers and groups intermediate data by key across nodes
- **Reduce**: Aggregates values for each key to produce final results

The shuffle phase sorts intermediate keys and redistributes data to reducers, enabling grouping by key. This disk-based approach ensures reliability but introduces latency [9].

Spark RDDs and DAG Execution

Spark improves on MapReduce through in-memory RDDs and Directed Acyclic Graph (DAG) execution:

- **RDDs**: Immutable distributed datasets partitioned across nodes
- **DAG Scheduler**: Optimizes execution by pipelining narrow transformations (map, filter) into stages
- **Wide Transformations**: Require shuffling (e.g., reduceByKey) and create stage boundaries

Spark's DAG-driven execution avoids unnecessary disk I/O, achieving up to 100x faster performance for iterative algorithms compared to Hadoop [10].

Word Frequency Algorithm in MapReduce

Algorithm 1 Word Count in MapReduce

```
Map Phase:
for each line in input do
    for each word in line.split() do
        Emit ⟨word, 1⟩
    end for
end for
Reduce Phase:
for each word in grouped keys do
    Sum = ∑ values
    Emit ⟨word, Sum⟩
end for
```

After presenting the MapReduce algorithm for word frequency counting, it is important to recognize how such parallel workflows are executed in practice. In a distributed environment, large datasets are partitioned and processed simultaneously across multiple nodes, significantly reducing computation time compared to serial execution. The efficiency of this approach depends on effective data partitioning, load balancing, and minimizing data transfer during the shuffle phase. Modern frameworks like Hadoop and Spark automate much of this orchestration, allowing developers to focus on defining transformation logic rather than managing low-level parallelism. As a result, organizations can scale their data processing pipelines to handle terabytes or petabytes of information, enabling timely insights and supporting advanced analytics tasks.

Parallel Processing Across Nodes

The following diagram illustrates how a typical parallel processing workflow is structured across nodes in a cluster, highlighting the flow of data from initial partitioning through mapping, shuffling, and final reduction.

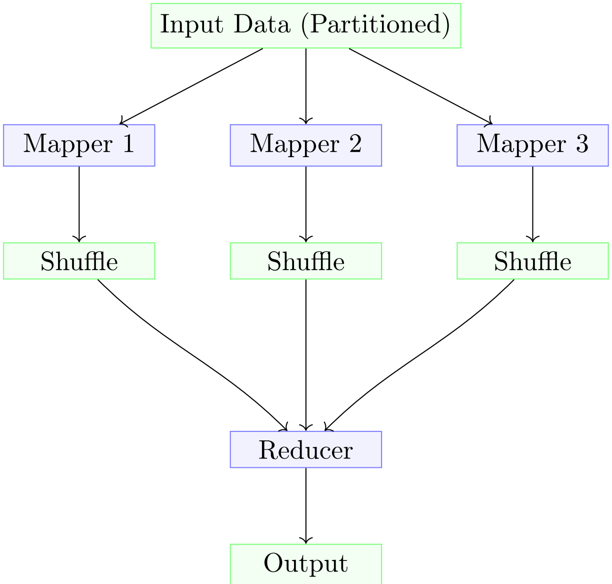


Fig. 3: Parallel processing flow: Mappers process partitions independently, shuffle phase groups data, reducer aggregates results

5 Building Scalable Data Pipelines

Modern data pipelines require robust orchestration and processing frameworks to handle diverse workloads. This section explores key tools and patterns for constructing production-grade data workflows.

ETL/ELT Orchestration

- **Apache Airflow**: Python-based DAGs with rich operator ecosystem [11]
- **Luigi**: Spotify's simpler alternative for dependency resolution
- **Prefect**: Modern workflow system with hybrid execution

Batch vs. Stream Processing

- **Spark**: Micro-batch processing (RDDs) with mature ML support
- **Flink**: True streaming with sub-second latency and stateful computations

Pipeline Architecture

Fig. 4: Minimal data pipeline architecture

Airflow DAG Example

```
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime

def extract(): pass
def transform(): pass
def load(): pass

with DAG(
    dag_id='etl_pipeline',
    start_date=datetime(2025, 1, 1),
    schedule='@daily'
) as dag:
    extract_task = PythonOperator(task_id='extract',
       python_callable=extract)
    transform_task = PythonOperator(task_id='transform',
       python_callable=transform)
    load_task = PythonOperator(task_id='load', python_callable=
       load)

    extract_task >> transform_task >> load_task
```

Orchestration Tool Comparison

Table 3: Data Orchestration Tools

Feature	Airflow	Prefect	Dagster
Workflow Type	Static DAGs	Dynamic Flows	Asset-Centric
Error Handling	Retries	Auto-recovery	Declarative
UI	Mature	Modern	Developer-Focused
Best For	ETL/ELT	Cloud-Native	Data Contracts

6 Big Data Analytics in Retail

Modern retailers leverage big data technologies to optimize operations and enhance customer experiences. This section explores two critical applications: real-time inventory management and customer segmentation, enabled by distributed processing frameworks.

Real-Time Inventory Management with Kafka and Spark

Apache Kafka serves as the central nervous system for real-time inventory tracking, ingesting data from POS systems, RFID sensors, and e-commerce platforms. Walmart's implementation processes **4+ billion messages in 3 hours** to generate replenishment orders across 4,700+ stores [12]. The architecture combines:

- **Kafka Streams**: Processes 150K+ events/sec for stock updates
- **Spark Structured Streaming**: Calculates inventory positions using micro-batches
- **KSQL DB**: Maintains real-time materialized views of stock levels

This pipeline reduces stockouts by 23% and improves inventory turnover by 17% compared to batch systems [13].

Customer Segmentation with Spark MLlib

Retailers use Hadoop/Spark MLlib to cluster customers based on:

- Purchase history (RFM analysis)
- Demographic attributes
- Real-time browsing behavior

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

# Feature engineering
assembler = VectorAssembler(
    inputCols=["annual_spend", "visit_frequency", "basket_size"],
    outputCol="features")
df = assembler.transform(customer_data)

# K-means clustering
kmeans = KMeans(k=5, seed=42)
model = kmeans.fit(df)
```

Migros Switzerland achieved **35% higher campaign conversion rates** using this approach [14].

Analytics Pipeline Architecture

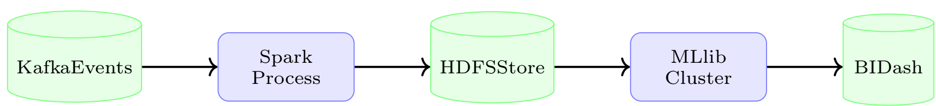


Fig. 5: Compact retail analytics pipeline

Performance Metrics

Table 4: Retail Analytics Performance Benchmarks

Metric	Kafka/Spark	Batch System	Improvement
Throughput (msgs/sec)	150,000	5,000	30x
Latency (95th %ile)	1.2s	45min	2250x
Inventory Accuracy	99.8%	92.4%	+7.4pp
Segmentation Speed	15min	6hr	24x

Exercises

Python Tasks

1. **Spark DataFrame Analysis (Walmart Stock Data)**

```
# Load Walmart stock data (2012-2017)
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
df = spark.read.csv("walmart_stock.csv", header=True,
    inferSchema=True)

# 1. Calculate monthly average closing price
from pyspark.sql.functions import month, avg
monthly_avg = df.withColumn("Month", month("Date")) \
                .groupBy("Month") \
                .agg(avg("Close").alias("AvgClose")) \
                .orderBy("Month")
```

2. **NoSQL Query Optimization (MongoDB)**

```
# Create optimized index and projection
db.transactions.create_index([("amount", 1), ("timestamp",
    -1)])
optimized_query = db.transactions.find(
    {"amount": {"$gt": 1000}},
    {"_id":0, "card_number":1, "timestamp":1}
).limit(100).sort("timestamp", -1)
```

Cloud Comparison Task

Implement cluster deployment for both platforms:

AWS CLI:

```
aws emr create-cluster --name "FraudCluster" --release-label emr
    -6.10.0
```

Service	AWS	GCP
Managed Spark	EMR	Dataproc
Object Storage	S3	Cloud Storage
ML Service	SageMaker	Vertex AI
CLI Tool	AWS CLI	gcloud

GCP CLI:

```
gcloud dataproc clusters create "fraud-detection" --region us-
    central1
```

Mini-Project: Fraud Detection Pipeline

Build a real-time fraud detection system with:

- Kafka topic for transaction streaming (1M msg/sec)
- Spark Structured Streaming for anomaly detection
- Redis for blacklist IP caching (5ms latency SLA)
- Dashboard using Streamlit/Plotly

Discussion Question

Compare MongoDB and Cassandra in the context of CAP theorem tradeoffs for financial transactions. Which would you choose for:

- Credit card fraud detection (AP vs CP)?
- Transaction ledger system (CA vs CP)?

References

[1] Quantzig: Exploring the four vs of big data (2024)

[2] Elevondata: The Top Challenges of Big Data. https://www.linkedin.com/pulse/top-challenges-big-data-volume-velocity-variety-veracity-elevondata

[3] Academy, F.: Evolution of Big Data: History, Tools, Future Trends. https://www.fynd.academy/blog/evolution-of-big-data

[4] Software, B.: CAP Theorem Explained: Consistency, Availability & Partition Tolerance. https://www.bmc.com/blogs/cap-theorem/

[5] Databricks: Hadoop Ecosystem: Components & Architecture. https://www.databricks.com/glossary/hadoop-ecosystem

[6] upGrad: Hive Vs Spark: Key Differences and Comparison Guide. https://www.upgrad.com/blog/hive-vs-spark/

[7] DataFlair: How HDFS Achieves Fault Tolerance? https://data-flair.training/blogs/learn-hadoop-hdfs-fault-tolerance/

[8] Studio3T: MongoDB Aggregation Example. https://studio3t.com/knowledge-base/articles/build-mongodb-aggregation-queries/

[9] DataFlair: Shuffling and Sorting in Hadoop MapReduce. https://data-flair.training/blogs/shuffling-and-sorting-in-hadoop/

[10] SparkByExamples: What Is DAG in Spark. https://sparkbyexamples.com/spark/what-is-dag-in-spark/

[11] Astronomer: Introduction to Apache Airflow DAGs. https://www.astronomer.io/docs/learn/dags/

[12] Waehner, K.: Real-Time Supply Chain with Apache Kafka. https://www.kai-waehner.de/blog/2022/02/25/real-time-supply-chain-with-apache-kafka-in-food-retail-industry/

[13] Confluent: Real-Time Inventory in Retail. https://www.confluent.io/blog/real-time-inventory-in-retail/

[14] N-iX: Real-Time Big Data Analytics Use Cases. https://www.n-ix.com/real-time-big-data-analytics/

[15] Pluralsight: Storage Showdown: AWS Vs Azure Vs GCP. https://www.pluralsight.com/resources/blog/cloud/storage-showdown-aws-vs-azure-vs-gcp-cloud-comparison

[16] Cloud, G.: Google Cloud Service Comparison. https://cloud.google.com/docs/get-started/aws-azure-gcp-service-comparison

[17] RisingWave: Airflow Vs Dagster Vs Prefect Comparison. https://risingwave.com/blog/airflow-vs-dagster-vs-prefect