BIG DATA ANALYTICS

Notes • Syllabus • Lab Manual • Study Material

A complete reference covering the foundations of big data, the Hadoop ecosystem, NoSQL stores, real-time processing with Spark, machine learning at scale, and a hands-on laboratory manual.

Engineering / MCA / Data Science Curriculum • Edition 2026

Table of Contents

Table of Contents 2

About This Material 4

Course Objectives & Outcomes 5

Course Objectives . . . 5

Course Outcomes (CO) . . . 5

Detailed Syllabus 6

Text & Reference Books . . . 7

Unit 1 — Introduction to Big Data 8

1.1 The Data Explosion . . . 8

1.2 What is Big Data? . . . 8

1.3 Characteristics — The V's of Big Data . . . 8

1.4 Types of Data . . . 8

1.5 Sources of Big Data . . . 9

1.6 Big Data vs. Traditional Systems . . . 9

1.7 Types of Big Data Analytics . . . 9

1.8 Challenges of Big Data . . . 9

1.9 Applications & Case Studies . . . 10

Unit 2 — Hadoop & HDFS 11

2.1 What is Hadoop? . . . 11

2.2 Core Components & Ecosystem . . . 11

2.3 HDFS Architecture . . . 11

2.4 Blocks, Replication & Rack Awareness . . . 11

2.5 HDFS Read & Write Flow . . . 12

2.6 Common HDFS Commands . . . 12

2.7 YARN — Yet Another Resource Negotiator . . . 12

2.8 Cluster Modes . . . 13

Unit 3 — MapReduce & NoSQL 14

3.1 The MapReduce Programming Model . . . 14

3.2 Word-Count: The “Hello World” of MapReduce . . . 14

3.3 Anatomy of a MapReduce Job . . . 14

3.4 Limitations of MapReduce . . . 15

3.5 Introduction to NoSQL . . . 15
3.6 CAP Theorem & BASE . . . 15
3.7 HBase — A Column-Family Store . . . 16
Unit 4 — Hive, Pig & Spark 17
4.1 Apache Hive — SQL on Hadoop . . . 17
4.2 Apache Pig — Data-Flow Scripting . . . 17
4.3 Data Ingestion — Sqoop & Flume . . . 18
4.4 Apache Spark — In-Memory Processing . . . 18
Unit 5 — Analytics, ML & Visualization 21
5.1 The Big-Data Analytics Lifecycle . . . 21
5.2 Data Preprocessing at Scale . . . 21
5.3 Machine Learning with Spark MLlib . . . 21
5.4 Mining Massive Datasets . . . 22
5.5 Streaming Analytics . . . 22
5.6 Graph Analytics (GraphX) . . . 22
5.7 Data Visualization . . . 22
5.8 Ethics, Privacy & Governance . . . 23
Laboratory Manual 24
Experiment 1: Install & Configure Hadoop (Pseudo-Distributed) . . . 25
Experiment 2: HDFS File Operations . . . 25
Experiment 3: MapReduce WordCount . . . 26
Experiment 5: Hive — Create & Query Tables . . . 27
Experiment 6: Apache Pig Data Processing . . . 27
Experiment 7: HBase CRUD Operations . . . 27
Experiment 9: Spark RDD WordCount (PySpark) . . . 29
Experiment 10: Spark SQL & DataFrame Analytics . . . 29
Experiment 11: Machine Learning with Spark MLlib . . . 29
Experiment 12: Mini-Project — End-to-End Analytics . . . 30
Study Material & Exam Preparation 31
Important Questions (Long Answer) . . . 31
Two-Mark Questions & Answers . . . 32
Glossary of Key Terms . . . 35
Common Abbreviations . . . 35

About This Material

This document is a self-contained study companion for a one-semester course in Big Data Analytics. It is organised into five teaching units that move from conceptual foundations to applied analytics, followed by a laboratory manual and an exam-oriented study section. Each unit blends concise theory, comparison tables, worked syntax, and revision pointers so it can serve both as a first-read tutorial and a last-minute refresher.

How to use it

- **Skim the syllabus first** to see how topics map to your course outcomes.
- **Read units in order** — later units assume the vocabulary built earlier.
- **Run the lab manual** alongside Units 2–5 to anchor theory in practice.
- **Use the study section** (important questions, two-mark Q&A, glossary) for revision and self-testing.

> **Prerequisites**
> Basic programming (preferably Python or Java), familiarity with the Linux command line, elementary SQL, and a working knowledge of probability and statistics. No prior distributed-systems experience is assumed.

Course Objectives & Outcomes

Course Objectives

1. Understand the nature, sources, and challenges of big data and the limits of traditional data-processing systems.
2. Explain the architecture of the Hadoop ecosystem including HDFS, MapReduce, and YARN.
3. Apply NoSQL data models and query big datasets using Hive, Pig, and HBase.
4. Develop scalable data pipelines and analytics using Apache Spark.
5. Apply machine-learning and visualization techniques to derive insight from large-scale data.

Course Outcomes (CO)

CO	On completion the learner can…	Bloom Level
CO1	Describe big-data characteristics, sources and use-cases.	Understand
CO2	Explain and operate the HDFS / MapReduce / YARN stack.	Apply
CO3	Model and query data with NoSQL, Hive, Pig and HBase.	Apply
CO4	Build batch and streaming pipelines using Spark.	Apply / Analyse
CO5	Perform analytics, ML and visualization on big data.	Analyse / Create

Detailed Syllabus

The syllabus below is representative of a standard Big Data Analytics course (typically 45–60 contact hours). Sequence and depth may vary by institution; map each unit to your own course plan.

UNIT I

Introduction to Big Data

Suggested: 9 hrs

Data explosion and evolution of data management; what is big data; the V's (volume, velocity, variety, veracity, value, variability); structured, semi-structured and unstructured data; sources of big data; big-data vs traditional BI/RDBMS; challenges — storage, processing, scalability, quality; big-data analytics types (descriptive, diagnostic, predictive, prescriptive); applications and case studies; the big-data technology landscape.

UNIT II

Hadoop & HDFS

Suggested: 10 hrs

History and design goals of Hadoop; Hadoop core components and ecosystem overview; HDFS architecture — NameNode, DataNode, Secondary NameNode, blocks, replication, rack awareness; read/write data flow; HDFS commands; fault tolerance and high availability; YARN architecture — ResourceManager, NodeManager, ApplicationMaster; Hadoop cluster modes and configuration.

UNIT III

MapReduce & NoSQL

Suggested: 10 hrs

MapReduce programming model; map, shuffle/sort and reduce phases; anatomy of a MapReduce job; combiners, partitioners and counters; word-count and other patterns; limitations of MapReduce; introduction to NoSQL; CAP theorem and BASE; key-value, document, column-family and graph stores; HBase architecture and data model; comparison with RDBMS.

UNIT IV

Hive, Pig & Spark

Suggested: 11 hrs

Data warehousing on Hadoop with Hive — architecture, HiveQL, tables, partitions and buckets; Pig and Pig Latin data-flow scripting; Sqoop and Flume for ingestion; limitations of disk-based processing; Apache Spark architecture; RDDs, transformations and actions; DataFrames and Spark SQL; Spark Streaming fundamentals; in-memory computing advantages.

UNIT V

Analytics, ML & Visualization

Suggested: 10 hrs

Big-data analytics lifecycle; data preprocessing and feature engineering at scale; mining large datasets; recommendation and clustering; classification and regression with Spark MLlib; streaming analytics; graph analytics (GraphX) overview; data visualization principles and tools; ethics, privacy and governance in big data.

Text & Reference Books

- Tom White, *Hadoop: The Definitive Guide*, O'Reilly.
- Seema Acharya & Subhashini Chellappan, *Big Data and Analytics*, Wiley.
- Bill Franks, *Taming the Big Data Tidal Wave*, Wiley.
- Holden Karau et al., *Learning Spark*, O'Reilly.
- Jure Leskovec, Anand Rajaraman, Jeff Ullman, *Mining of Massive Datasets*, Cambridge University Press.

Unit 1 — Introduction to Big Data

1.1 The Data Explosion

Every digital interaction — a search query, a swipe, a sensor reading, a transaction — leaves a data trail. The cost of generating and storing data has fallen sharply while the number of data-producing devices has grown exponentially. Social platforms, e-commerce, the Internet of Things (IoT), scientific instruments and machine logs now generate data at a scale and speed that classical single-machine databases were never designed to handle. **Big data** is the discipline of capturing, storing, and extracting value from datasets that are too large, too fast, or too varied for traditional tools to process on a single computer.

1.2 What is Big Data?

Big data refers to data assets whose volume, velocity and variety demand cost-effective, innovative forms of processing that enable enhanced insight, decision-making and process automation. The key idea is not merely “large size” — it is that the data outgrows the capacity of conventional systems, forcing a shift to **distributed, horizontally-scalable** architectures where work is spread across clusters of commodity machines.

1.3 Characteristics — The V's of Big Data

Big data is most often described through a set of defining properties, the “V's”. The original three were proposed by analyst Doug Laney; additional V's were added later as the field matured.

Characteristic	Meaning	Example
Volume	Sheer quantity of data, from terabytes to petabytes and beyond.	Years of clickstream logs from a large website.
Velocity	Speed at which data is generated and must be processed.	Stock-market ticks or sensor streams in milliseconds.
Variety	Many formats: structured, semi-structured, unstructured.	Tables, JSON, text, images, audio, video.
Veracity	Trustworthiness, quality and consistency of data.	Noisy, biased or incomplete user-generated content.
Value	The usable insight extracted relative to cost.	Fraud detection that saves money.
Variability	Changes in meaning, structure or flow over time.	Seasonal spikes; evolving log schemas.

> **Exam tip**
> If asked to “list the characteristics of big data”, name the V's and give a one-line definition plus example for each. The first three (Volume, Velocity, Variety) are the classic core; Veracity, Value and Variability are common extensions.

1.4 Types of Data

Type	Description	Examples
Structured	Fixed schema, fits neatly into rows and columns.	RDBMS tables, CSV, spreadsheets.
Semi-structured	Has tags/markers but no rigid table schema.	JSON, XML, log files, e-mail headers.
Unstructured	No predefined model; the bulk of big data.	Free text, images, audio, video, social posts.

Industry estimates suggest the large majority of enterprise data is unstructured or semi-structured, which is precisely why big-data tools that handle schema-on-read and arbitrary formats became essential.

1.5 Sources of Big Data

- **Human-generated:** social media, e-mails, documents, photos, videos.
- **Machine-generated:** server and application logs, sensors, IoT devices, GPS, RFID, telemetry.
- **Business/transactional:** point-of-sale, banking, e-commerce orders, CRM and ERP systems.
- **Public/open data:** government datasets, weather, scientific and research archives.

1.6 Big Data vs. Traditional Systems

Aspect	Traditional RDBMS / BI	Big Data Systems
Data size	Gigabytes to a few terabytes	Terabytes to petabytes+
Schema	Schema-on-write (defined first)	Schema-on-read (flexible)
Data type	Mostly structured	Structured + semi + unstructured
Scaling	Vertical (bigger server)	Horizontal (more nodes)
Processing	Centralised	Distributed / parallel
Cost model	Costly specialised hardware	Commodity clusters / cloud
Examples	Oracle, MySQL, SQL Server	Hadoop, Spark, NoSQL, cloud DW

1.7 Types of Big Data Analytics

Analytics maturity is commonly described along four ascending levels. Each answers a different question and delivers progressively greater business value at greater complexity.

Type	Question answered	Techniques
Descriptive	What happened?	Aggregation, reporting, dashboards.
Diagnostic	Why did it happen?	Drill-down, correlation, root-cause.
Predictive	What is likely to happen?	Regression, classification, forecasting.
Prescriptive	What should we do about it?	Optimisation, simulation, recommendation.

1.8 Challenges of Big Data

- **Storage & cost** — reliably retaining petabytes affordably.
- **Processing speed** — analysing data fast enough to be useful.
- **Scalability** — growing capacity without re-architecting.
- **Data quality & veracity** — cleaning noisy, inconsistent data.
- **Integration** — unifying many sources and formats.
- **Security & privacy** — protecting sensitive data and meeting regulation.
- **Skills gap** — scarcity of trained data engineers and scientists.

1.9 Applications & Case Studies

- **Retail & e-commerce:** recommendation engines, demand forecasting, dynamic pricing.
- **Banking & finance:** real-time fraud detection, credit-risk scoring, algorithmic trading.
- **Healthcare:** patient-outcome prediction, genomics, epidemic tracking.
- **Telecom:** network optimisation, churn prediction, call-detail-record analysis.
- **Manufacturing/IoT:** predictive maintenance, quality control, supply-chain visibility.
- **Social & media:** sentiment analysis, content personalisation, trend detection.

> **Unit 1 in one line**
> Big data is data that outgrows traditional tools in volume, velocity and variety, forcing a move to distributed systems; its value is unlocked through descriptive, diagnostic, predictive and prescriptive analytics.

Unit 2 — Hadoop & HDFS

2.1 What is Hadoop?

Apache Hadoop is an open-source framework for the distributed storage and processing of very large datasets across clusters of commodity hardware. It was inspired by two Google papers — the Google File System (GFS) and MapReduce — and is designed around three principles: move computation to the data rather than data to computation, tolerate hardware failure as the norm, and scale out simply by adding more nodes.

2.2 Core Components & Ecosystem

Component	Role
HDFS	Distributed, fault-tolerant storage layer.
YARN	Cluster resource management and job scheduling.
MapReduce	Batch distributed-processing programming model.
Hadoop Common	Shared libraries and utilities.

Around this core sits a broad ecosystem: **Hive** (SQL-on-Hadoop), **Pig** (data-flow scripting), **HBase** (NoSQL column store), **Sqoop** and **Flume** (ingestion), **Oozie** (workflow), **ZooKeeper** (coordination), and **Spark** (in-memory processing).

2.3 HDFS Architecture

HDFS follows a **master–slave** design. A single master coordinates metadata while many slaves store the actual data blocks.

Daemon	Type	Responsibility
NameNode	Master	Stores the filesystem namespace and metadata (file→block→DataNode map); never stores actual data.
DataNode	Slave	Stores and serves the actual data blocks; reports to the NameNode via heartbeats and block reports.
Secondary NameNode	Helper	Periodically merges the edit log with the filesystem image (checkpointing); it is NOT a hot standby/backup.

> **Common misconception**
> The Secondary NameNode is not a failover NameNode. It only performs checkpoint housekeeping. True high availability uses an Active/Standby NameNode pair coordinated by ZooKeeper and a shared edit log.

2.4 Blocks, Replication & Rack Awareness

- Files are split into large fixed-size **blocks** (commonly 128 MB, configurable). Large blocks reduce metadata overhead and seek time.
- Each block is **replicated** (default factor 3) across DataNodes for fault tolerance.

- **Rack awareness** places replicas across different racks so the data survives a whole-rack failure while keeping network traffic efficient.
- If a DataNode fails, the NameNode detects missing heartbeats and **re-replicates** the affected blocks to restore the replication factor.

2.5 HDFS Read & Write Flow

Write path

1. Client asks the NameNode to create a file; the NameNode checks permissions and returns target DataNodes for the first block.
2. Client writes the block to the first DataNode, which **pipelines** it to the second, and the second to the third.
3. Acknowledgements flow back along the pipeline; the process repeats per block until the file is complete and closed.

Read path

1. Client asks the NameNode for the block locations of a file.
2. NameNode returns, for each block, the DataNodes holding it, ordered by network proximity to the client.
3. Client reads each block directly from the nearest available DataNode, bypassing the NameNode for the bulk data transfer.

2.6 Common HDFS Commands

HDFS shell

```
# list a directory
hdfs dfs -ls /user/data

# make a directory
hdfs dfs -mkdir -p /user/data/raw

# copy a local file into HDFS
hdfs dfs -put sales.csv /user/data/raw/

# read a file from HDFS
hdfs dfs -cat /user/data/raw/sales.csv | head

# copy from HDFS back to local disk
hdfs dfs -get /user/data/raw/sales.csv ./

# check replication / health of the filesystem
hdfs fsck /user/data -files -blocks -locations
```

2.7 YARN — Yet Another Resource Negotiator

YARN decouples resource management from the processing model, allowing MapReduce, Spark and other engines to share a single cluster. Its key daemons are:

Component	Responsibility
ResourceManager (RM)	Global master; arbitrates cluster resources between competing applications via the Scheduler.
NodeManager (NM)	Per-node agent; manages containers, monitors resource usage and reports to the RM.
ApplicationMaster (AM)	Per-application; negotiates containers from the RM and coordinates task execution.
Container	A bundle of resources (CPU, memory) on a node in which a task actually runs.

Job flow: the client submits an application to the RM, which launches an AM in a container; the AM requests further containers from the RM and runs the application's tasks inside them on various NodeManagers.

2.8 Cluster Modes

Mode	Description	Use
Standalone (local)	Single JVM, no daemons, local filesystem.	Quick testing/debugging.
Pseudo-distributed	All daemons on one machine, HDFS active.	Learning, single-box dev.
Fully distributed	Daemons spread across many nodes.	Production clusters.

Unit 2 in one line

Hadoop stores data reliably with HDFS (NameNode metadata + replicated DataNode blocks) and runs distributed jobs through YARN, which manages cluster resources for MapReduce, Spark and other engines.

Unit 3 — MapReduce & NoSQL

3.1 The MapReduce Programming Model

MapReduce is a programming model for processing large datasets in parallel across a cluster. The programmer expresses computation as two functions — **map** and **reduce** — and the framework handles parallelisation, data distribution, fault tolerance and load balancing. All data moves as **key–value pairs**.

Phase	What happens
Map	Each input split is processed by a mapper that emits intermediate key–value pairs.
Shuffle & Sort	The framework groups all values by key and sorts them, transferring data across the network to reducers.
Reduce	Each reducer receives a key and the list of its values and emits the final aggregated output.

3.2 Word-Count: The “Hello World” of MapReduce

Counting word frequencies in a large corpus illustrates the model cleanly:

WordCount pseudocode

```
# MAP: for each line, emit (word, 1)
map(key=lineOffset, value=line):
    for word in value.split():
        emit(word, 1)

# SHUFFLE & SORT (done by framework):
#   ('big', [1,1,1])  ('data', [1,1])  ...

# REDUCE: sum the counts per word
reduce(key=word, values=[1,1,1,...]):
    emit(word, sum(values))
```

3.3 Anatomy of a MapReduce Job

1. **Input Splits** — input is divided into logical splits, one per mapper.
2. **RecordReader** — converts a split into key–value records.
3. **Mapper** — produces intermediate pairs.
4. **Combiner (optional)** — a mini-reducer that aggregates locally to cut network traffic.
5. **Partitioner** — decides which reducer each key goes to (default: hash).
6. **Shuffle & Sort** — transfers and sorts intermediate data.
7. **Reducer** — produces final output written to HDFS.

Helper	Purpose
Combiner	Local aggregation on the map side to reduce shuffle volume (must be associative & commutative).
Partitioner	Controls key-to-reducer assignment for balanced load.
Counter	Lightweight global statistic for tracking events/quality.

3.4 Limitations of MapReduce

- Heavy **disk I/O** — intermediate results are written to disk between stages, making iterative algorithms slow.
- **High latency** — unsuitable for interactive or real-time queries.
- **Rigid two-stage model** — complex workflows need many chained jobs.
- **Verbose** — low-level Java code for simple operations (motivating Hive, Pig and Spark).

3.5 Introduction to NoSQL

NoSQL (“not only SQL”) databases relax the rigid relational model to achieve horizontal scalability, flexible schemas and high availability for big-data and web-scale workloads. They typically favour **schema-on-read** and distribute data across many nodes.

Family	Data model	Examples	Good for
Key–Value	Simple key → value pairs	Redis, DynamoDB	Caching, sessions, fast lookups
Document	Self-describing JSON/BSON docs	MongoDB, CouchDB	Catalogs, content, profiles
Column-family	Rows with dynamic column groups	HBase, Cassandra	Time-series, wide sparse data
Graph	Nodes & relationships	Neo4j, JanusGraph	Social/recommendation networks

3.6 CAP Theorem & BASE

The **CAP theorem** (Brewer) states that a distributed data store can guarantee at most two of three properties simultaneously when a network partition occurs:

- **Consistency (C):** every read sees the most recent write.
- **Availability (A):** every request receives a (non-error) response.
- **Partition tolerance (P):** the system keeps working despite dropped messages between nodes.

Because partitions are unavoidable in real networks, practical systems trade off C against A (e.g. CP vs AP). Many NoSQL stores follow **BASE** (Basically Available, Soft state, Eventual consistency) rather than the relational **ACID** guarantees — accepting temporary inconsistency in exchange for availability and scale.

ACID (RDBMS)	BASE (many NoSQL)
Atomicity, Consistency, Isolation, Durability	Basically Available, Soft state, Eventual consistency
Strong consistency, immediate	Eventual consistency, high availability
Vertical scaling, transactions	Horizontal scaling, partition tolerance

3.7 HBase — A Column-Family Store

HBase is a distributed, scalable NoSQL database modelled on Google's Bigtable and built on top of HDFS. It provides real-time random read/write access to very large tables (billions of rows × millions of columns).

Concept	Meaning
Row key	Unique, sorted identifier for a row; primary access path.
Column family	Group of columns stored & tuned together; defined at table creation.
Column qualifier	A column within a family; can be added dynamically.
Cell / version	Value at (row, column) keyed by timestamp; multiple versions retained.
Region	A horizontal slice of a table served by a RegionServer.

Feature	RDBMS	HBase
Schema	Fixed, rigid	Flexible, sparse columns
Scaling	Vertical	Horizontal (regions)
Transactions	Full ACID, multi-row	Atomic per row only
Query	Rich SQL, joins	Key-based get/scan, no joins
Best for	Structured OLTP	Massive sparse, random access

Unit 3 in one line

MapReduce processes data in parallel via map → shuffle/sort → reduce but is disk-heavy and high-latency; NoSQL stores (key-value, document, column-family, graph) trade strict ACID for the scalability described by the CAP theorem and BASE.

Unit 4 — Hive, Pig & Spark

4.1 Apache Hive — SQL on Hadoop

Hive is a data-warehouse layer over Hadoop that lets analysts query data with a familiar SQL-like language called **HiveQL**. Hive translates queries into execution plans (originally MapReduce, now often Tez or Spark) so users get the power of distributed processing without writing low-level code. It is optimised for **batch analytical** queries, not low-latency transactions.

Hive component	Role
Metastore	Stores table schemas and metadata (often in a relational DB).
Driver	Manages the lifecycle of a HiveQL statement.
Compiler	Parses and turns HiveQL into an execution plan / DAG.
Execution engine	Runs the plan on MapReduce / Tez / Spark.

Tables, Partitions & Buckets

- **Managed vs external tables:** dropping a managed table deletes its data; dropping an external table leaves the underlying files intact.
- **Partitioning** splits data by a column value (e.g. date) into separate directories so queries scan only relevant partitions.
- **Bucketing** further hashes rows into a fixed number of files for efficient sampling and joins.

```HiveQL
CREATE TABLE sales (id INT, amount DOUBLE, region STRING)
PARTITIONED BY (sale_date STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';

LOAD DATA INPATH '/user/data/sales2026.csv'
INTO TABLE sales PARTITION (sale_date='2026-01-15');

SELECT region, SUM(amount) AS total
FROM sales
WHERE sale_date='2026-01-15'
GROUP BY region
ORDER BY total DESC;
```

4.2 Apache Pig — Data-Flow Scripting

Pig offers a high-level scripting language, **Pig Latin**, for expressing data-transformation pipelines as a sequence of steps. It suits ETL and exploratory work where the procedural, step-by-step style is more natural than SQL. Pig scripts compile down to MapReduce/Tez/Spark jobs.

```
Pig Latin — word count

lines  = LOAD '/data/book.txt' AS (line:chararray);
words  = FOREACH lines GENERATE FLATTEN(TOKENIZE(line)) AS word;
grp    = GROUP words BY word;
counts = FOREACH grp GENERATE group AS word, COUNT(words) AS n;
sorted = ORDER counts BY n DESC;
STORE sorted INTO '/data/wordcount';
```

	Hive	Pig
Language	Declarative (SQL-like)	Procedural (data-flow)
Best user	Analysts, BI	ETL developers, researchers
Schema	Schema required (warehouse)	Schema optional (schema-on-read)
Typical use	Structured reporting	Pipelines, semi-structured data

4.3 Data Ingestion — Sqoop & Flume

- **Sqoop** bulk-transfers structured data between relational databases and Hadoop (import/export), parallelising via MapReduce.
- **Flume** reliably collects, aggregates and streams large volumes of event/log data (e.g. server logs) into HDFS or HBase.

4.4 Apache Spark — In-Memory Processing

Spark is a fast, general-purpose distributed computing engine that keeps intermediate data **in memory**, making it dramatically faster than disk-based MapReduce for iterative and interactive workloads. It offers a unified stack for batch, SQL, streaming, machine learning and graph processing.

Spark module	Purpose
Spark Core	Engine, scheduling, and the RDD abstraction.
Spark SQL	Structured queries over DataFrames/Datasets.
Spark Streaming	Near-real-time processing of data streams.
MLlib	Scalable machine-learning algorithms.
GraphX	Graph-parallel computation.

4.4.1 RDDs — Resilient Distributed Datasets

An RDD is an immutable, partitioned collection of records that can be operated on in parallel. RDDs achieve fault tolerance through **lineage** — the recorded sequence of transformations used to rebuild lost partitions — rather than by replicating data.

Operation type	Behaviour	Examples
Transformation	Lazy; defines a new RDD, not executed yet.	map, filter, flatMap, groupByKey, join
Action	Triggers execution and returns a result.	collect, count, reduce, saveAsTextFile

Lazy evaluation

Spark builds a DAG of transformations and only runs it when an action is called. This lets the engine optimise the whole pipeline before executing.

4.4.2 Spark Word Count (PySpark)

PySpark

```
from pyspark import SparkContext
sc = SparkContext('local', 'wordcount')

counts = (sc.textFile('/data/book.txt')
            .flatMap(lambda line: line.split())
            .map(lambda w: (w, 1))
            .reduceByKey(lambda a, b: a + b))

counts.saveAsTextFile('/data/spark_wordcount')
```

4.4.3 DataFrames & Spark SQL

DataFrames are distributed tables with a schema, optimised by Spark's **Catalyst** query optimiser and **Tungsten** execution engine. They give higher-level, more efficient APIs than raw RDDs.

Spark SQL / DataFrames

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('sales').getOrCreate()

df = spark.read.csv('/data/sales.csv', header=True, inferSchema=True)
df.groupBy('region').sum('amount').orderBy('sum(amount)', ascending=False).show()

df.createOrReplaceTempView('sales')
spark.sql('SELECT region, SUM(amount) t FROM sales GROUP BY region').show()
```

4.4.4 Spark vs MapReduce

Aspect	MapReduce	Spark
Data sharing	Disk between stages	In-memory (RDD/DataFrame)
Speed	Slower (heavy I/O)	Up to ~10–100× faster for iterative jobs
Model	Rigid map→reduce	Rich DAG of operators
Workloads	Batch	Batch, SQL, streaming, ML, graph
Ease of use	Verbose Java	Concise APIs in Scala/Python/Java/R

Unit 4 in one line

Hive (SQL) and Pig (data-flow) make Hadoop accessible without low-level code, while Spark's in-memory RDD/DataFrame model unifies batch, SQL, streaming and ML far faster than disk-based MapReduce.

Unit 5 — Analytics, ML & Visualization

5.1 The Big-Data Analytics Lifecycle

Real projects follow an iterative lifecycle that turns raw data into deployed insight. A common formulation has six phases:

Phase	Activity
1. Discovery	Frame the business problem, hypotheses and success metrics.
2. Data preparation	Acquire, clean, transform and stage data ('data wrangling').
3. Model planning	Choose techniques, features and evaluation strategy.
4. Model building	Train, tune and validate models at scale.
5. Communicate results	Visualise and explain findings to stakeholders.
6. Operationalise	Deploy, monitor and maintain the solution in production.

5.2 Data Preprocessing at Scale

- **Cleaning:** handle missing values, duplicates, outliers and noise.
- **Integration:** merge heterogeneous sources into a consistent view.
- **Transformation:** normalise, scale, encode categoricals, derive features.
- **Reduction:** sampling, dimensionality reduction (e.g. PCA), aggregation.
- **Feature engineering:** craft predictive variables — often the single biggest driver of model quality.

5.3 Machine Learning with Spark MLlib

MLlib provides distributed implementations of common algorithms so models can be trained on data far larger than one machine's memory. The main task families are:

Task	Goal	Example algorithms
Classification	Predict a category/label.	Logistic regression, decision trees, random forest, naive Bayes
Regression	Predict a continuous value.	Linear regression, GBT regression
Clustering	Group similar items (unsupervised).	K-means, Gaussian mixture, bisecting K-means
Recommendation	Suggest items to users.	Collaborative filtering (ALS)
Dimensionality reduction	Compress features.	PCA, SVD

Spark MLlib pipeline

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

assembler = VectorAssembler(
    inputCols=['age', 'balance', 'tenure'], outputCol='features')
data = assembler.transform(df).select('features', 'label')

train, test = data.randomSplit([0.8, 0.2], seed=42)
model = LogisticRegression().fit(train)
predictions = model.transform(test)
```

5.4 Mining Massive Datasets

- **Association rule mining** (market-basket): find items frequently bought together using support, confidence and lift (Apriori, FP-Growth).
- **Clustering**: segment customers, documents or sensor readings without labels.
- **Frequent-pattern & sequence mining**: detect recurring behaviours over time.
- **Recommendation**: collaborative filtering and content-based methods at web scale.

5.5 Streaming Analytics

Streaming analytics processes data continuously as it arrives, enabling real-time decisions (fraud alerts, live dashboards, anomaly detection). Spark handles this via **micro-batching** (Structured Streaming), while frameworks like Apache Kafka, Storm and Flink target true event-at-a-time processing. Key concepts include **windowing** (tumbling/sliding), **watermarks** for late data, and exactly-once delivery guarantees.

5.6 Graph Analytics (GraphX)

GraphX models data as vertices and edges and runs graph-parallel algorithms such as PageRank, connected components and shortest paths — valuable for social networks, fraud rings, recommendation and influence analysis.

5.7 Data Visualization

Visualization translates analytical results into a form humans can grasp quickly. Good charts respect a few principles:

- **Match chart to intent** — trends → line; comparison → bar; distribution → histogram; relationship → scatter; composition → stacked/area.
- **Maximise data-ink** — remove clutter that does not convey information.
- **Label and contextualise** — titles, units, and a clear takeaway.
- **Avoid distortion** — honest axes and scales.

Tool	Strength
Tableau / Power BI	Interactive business dashboards, drag-and-drop.
Matplotlib / Seaborn	Programmatic statistical charts in Python.
Plotly / D3.js	Interactive and web-based visualizations.
Apache Superset / Kibana	Open-source dashboards over big-data stores.

5.8 Ethics, Privacy & Governance

- **Privacy:** minimise, anonymise and protect personal data; honour consent and regulations (e.g. GDPR-style laws).
- **Bias & fairness:** guard against models that amplify historical discrimination.
- **Security:** encryption, access control and auditing across the pipeline.
- **Governance:** data lineage, cataloguing, quality and accountability.
- **Transparency:** explainable models and responsible deployment.

> **Unit 5 in one line**
> Big-data analytics follows a lifecycle from discovery to deployment; Spark MLlib brings classification, clustering and recommendation to scale, streaming enables real-time insight, and effective, ethical visualization turns results into decisions.

Laboratory Manual

The following experiments build practical skills progressively, from environment setup to a small end-to-end analytics project. Each experiment lists an aim, prerequisites, procedure and expected output. A common structure for every lab record is: Aim, Theory, Procedure/Code, Output, and Result/Conclusion.

Lab setup options
You can run these on (a) a single-node Hadoop/Spark install in pseudo-distributed mode on Linux, (b) a pre-built sandbox VM/Docker image, or (c) a cloud cluster. For most experiments a single-node setup is sufficient to learn the concepts.

#	Experiment	Maps to
1	Install & configure Hadoop (pseudo-distributed); verify daemons.	Unit 2
2	Perform HDFS file operations from the command line.	Unit 2
3	Write & run a MapReduce WordCount job.	Unit 3
4	Matrix / sales aggregation with MapReduce (or weather max-temp).	Unit 3
5	Create & query tables in Hive (partitions, aggregation).	Unit 4
6	Process data with Apache Pig (Pig Latin script).	Unit 4
7	Basic CRUD on HBase tables.	Unit 3/4
8	Import data from RDBMS to HDFS using Sqoop.	Unit 4
9	RDD operations & WordCount in Spark (PySpark).	Unit 4
10	DataFrames & Spark SQL analytics.	Unit 4
11	Build an ML model with Spark MLlib.	Unit 5
12	Mini-project: end-to-end analytics + visualization.	Unit 5

Experiment 1: Install & Configure Hadoop (Pseudo-Distributed)

Aim: To install Hadoop in pseudo-distributed mode on a single Linux machine and verify the daemons are running.

Prerequisites: Linux, Java (JDK 8/11), SSH configured for passwordless localhost login.

Procedure

1. Install Java and verify with `java -version`.
2. Download and extract a Hadoop release; set `HADOOP_HOME` and `JAVA_HOME`.
3. Edit core-site.xml, hdfs-site.xml, mapred-site.xml and yarn-site.xml for pseudo-distributed mode.
4. Format the NameNode (first time only).
5. Start HDFS and YARN, then verify with `jps`.

```bash
# configure passwordless ssh
ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys

# format the namenode (ONLY the first time)
hdfs namenode -format

# start the cluster
start-dfs.sh && start-yarn.sh

# verify the running daemons
jps   # expect: NameNode, DataNode, SecondaryNameNode,
      #         ResourceManager, NodeManager
```

Expected output: `jps` lists all five daemons; the HDFS web UI is reachable on the NameNode's HTTP port.

Experiment 2: HDFS File Operations

Aim: To perform basic file and directory operations on HDFS.

Prerequisites: A running HDFS (Experiment 1).

Procedure

1. Create a directory hierarchy in HDFS.
2. Upload a local file and list it.
3. Display the file, then download a copy back to local disk.
4. Inspect replication and block placement; then clean up.

```
HDFS shell
```

```
hdfs dfs -mkdir -p /lab/input
echo 'big data analytics lab' > sample.txt
hdfs dfs -put sample.txt /lab/input/
hdfs dfs -ls /lab/input
hdfs dfs -cat /lab/input/sample.txt
hdfs dfs -get /lab/input/sample.txt ./copy.txt
hdfs fsck /lab/input/sample.txt -files -blocks -locations
hdfs dfs -rm -r /lab/input
```

Expected output: The file is visible in HDFS, its contents print correctly, and fsck shows the configured replication factor.

Experiment 3: MapReduce WordCount

Aim: To count word frequencies in a text file using a MapReduce program.

Prerequisites: Running Hadoop; input text in HDFS.

Procedure

1. Place an input text file in HDFS.
2. Run the bundled example WordCount jar (or compile your own mapper/reducer).
3. Inspect the output directory in HDFS.

```
Run WordCount
```

```
hdfs dfs -mkdir -p /wc/in
hdfs dfs -put book.txt /wc/in/

# run the example jar shipped with Hadoop
hadoop jar $HADOOP_HOME/share/hadoop/mapreduce/\
  hadoop-mapreduce-examples-*.jar wordcount /wc/in /wc/out

hdfs dfs -cat /wc/out/part-r-00000 | head
```

Expected output: A part-r-00000 file containing each distinct word with its total count.

Experiment 5: Hive — Create & Query Tables

Aim: To create a partitioned Hive table, load data and run analytical queries.

Prerequisites: Hive installed over Hadoop.

Procedure

1. Launch the Hive shell / Beeline.
2. Create a database and a table.
3. Load data and run GROUP BY / aggregation queries.

```HiveQL
CREATE DATABASE IF NOT EXISTS lab;
USE lab;
CREATE TABLE employees (id INT, name STRING, dept STRING, salary DOUBLE)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
LOAD DATA LOCAL INPATH 'emp.csv' INTO TABLE employees;

SELECT dept, AVG(salary) AS avg_sal, COUNT(*) AS n
FROM employees GROUP BY dept ORDER BY avg_sal DESC;
```

Expected output: Per-department average salary and head-count, sorted descending.

Experiment 6: Apache Pig Data Processing

Aim: To process a dataset using a Pig Latin script.

Prerequisites: Pig installed; input file in HDFS.

Procedure

1. Load the data with a schema.
2. Filter, group and aggregate.
3. Store and inspect the result.

```Pig Latin
emp = LOAD '/lab/emp.csv' USING PigStorage(',')
      AS (id:int, name:chararray, dept:chararray, salary:double);
byDept = GROUP emp BY dept;
avgSal = FOREACH byDept GENERATE group AS dept,
         AVG(emp.salary) AS avg_salary;
STORE avgSal INTO '/lab/out_pig' USING PigStorage(',');
```

Expected output: A file with each department and its average salary.

Experiment 7: HBase CRUD Operations

Aim: To create a table and perform create, read, update and delete operations in HBase.

Prerequisites: HBase running over HDFS.

Procedure

1. Open the HBase shell.
2. Create a table with a column family.
3. Put, get, scan and delete cells.

```
HBase shell
```

```
create 'student', 'info'
put 'student', '1', 'info:name', 'Asha'
put 'student', '1', 'info:dept', 'CSE'
get  'student', '1'
scan 'student'
delete 'student', '1', 'info:dept'
disable 'student'  ;  drop 'student'
```

Expected output: The row is created, retrieved, the column is deleted, and the table is dropped successfully.

Experiment 9: Spark RDD WordCount (PySpark)

Aim: To perform WordCount using Spark RDD transformations and actions.

Prerequisites: Spark + PySpark installed.

Procedure

1. Create a SparkContext / SparkSession.
2. Read the text file as an RDD.
3. Apply flatMap, map and reduceByKey, then collect.

PySpark

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('wc').getOrCreate()
sc = spark.sparkContext

rdd = sc.textFile('book.txt')
counts = (rdd.flatMap(lambda l: l.lower().split())
             .map(lambda w: (w, 1))
             .reduceByKey(lambda a, b: a + b))
for w, c in counts.takeOrdered(10, key=lambda x: -x[1]):
    print(w, c)
```

Expected output: The ten most frequent words with their counts printed to the console.

Experiment 10: Spark SQL & DataFrame Analytics

Aim: To load a CSV into a DataFrame and run SQL-style analytics.

Prerequisites: Spark; a CSV dataset.

Procedure

1. Read the CSV with header and schema inference.
2. Register a temporary view.
3. Run aggregation queries with the DataFrame API and SQL.

Spark SQL

```
df = spark.read.csv('sales.csv', header=True, inferSchema=True)
df.printSchema()
df.createOrReplaceTempView('sales')

spark.sql('''SELECT region, ROUND(SUM(amount),2) AS total
             FROM sales GROUP BY region
             ORDER BY total DESC''').show()
```

Expected output: Total sales per region, sorted from highest to lowest.

Experiment 11: Machine Learning with Spark MLlib

Aim: To train and evaluate a classification model on a dataset using Spark MLlib.

Prerequisites: Spark MLlib; a labelled dataset.

Procedure

1. Assemble feature columns into a vector.
2. Split into training and test sets.
3. Fit a classifier and evaluate accuracy/AUC.

Spark MLlib

```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.evaluation import BinaryClassificationEvaluator

va = VectorAssembler(inputCols=['f1','f2','f3'], outputCol='features')
data = va.transform(df).select('features','label')
train, test = data.randomSplit([0.8, 0.2], seed=1)
model = LogisticRegression().fit(train)
pred  = model.transform(test)
auc = BinaryClassificationEvaluator().evaluate(pred)
print('AUC =', auc)
```

Expected output: A trained model and a printed evaluation metric (e.g. AUC) on the test set.

Experiment 12: Mini-Project — End-to-End Analytics

Aim: To build a small end-to-end pipeline: ingest → store → process → analyse → visualise.

Suggested flow

1. Pick a public dataset (e.g. retail transactions, taxi trips, tweets).
2. Ingest into HDFS (or read directly from CSV/JSON).
3. Clean and transform with Spark DataFrames.
4. Compute insights (top categories, trends, a simple prediction).
5. Export aggregated results and visualise them (Matplotlib / a BI tool).
6. Document findings as a short report with charts and conclusions.

> **Assessment rubric (typical)**
> Correctness of pipeline (30%), depth of analysis (25%), code quality and reproducibility (20%), visualization and communication (15%), report and viva (10%).

Study Material & Exam Preparation

This section collects exam-oriented resources: important long-answer questions, quick two-mark Q&A, a glossary and abbreviations. Use it to self-test after reading each unit.

Important Questions (Long Answer)

Unit 1

- Define big data. Explain its characteristics (the V's) with examples.
- Differentiate big-data systems from traditional RDBMS/BI systems.
- Describe the four types of analytics and the questions each answers.
- Discuss the major challenges and applications of big data.

Unit 2

- Explain HDFS architecture with the roles of NameNode and DataNode.
- Describe the HDFS write and read data flow.
- What is YARN? Explain ResourceManager, NodeManager and ApplicationMaster.
- Explain block replication and rack awareness in HDFS.

Unit 3

- Explain the MapReduce model with the WordCount example.
- Describe the anatomy of a MapReduce job including combiner and partitioner.
- State and explain the CAP theorem; compare ACID and BASE.
- Explain the HBase data model and compare HBase with an RDBMS.

Unit 4

- Explain Hive architecture and the use of partitions and buckets.
- Compare Hive and Pig with their typical use-cases.
- What are RDDs? Distinguish transformations from actions with examples.
- Compare Spark with MapReduce and explain why Spark is faster.

Unit 5

- Describe the big-data analytics lifecycle.
- Explain classification, clustering and recommendation with Spark MLlib.
- Discuss streaming analytics and windowing.
- Explain visualization principles and ethical/privacy concerns in big data.

Two-Mark Questions & Answers

Question	Answer
Q1. Define big data.	Data whose volume, velocity and variety exceed the capacity of traditional tools, requiring distributed processing to extract value.
Q2. Name the original three V's of big data.	Volume, Velocity and Variety.
Q3. What is HDFS?	The Hadoop Distributed File System — a fault-tolerant, distributed storage layer that splits files into replicated blocks across DataNodes.
Q4. What is the role of the NameNode?	It is the HDFS master that stores filesystem metadata and the file-to-block-to-DataNode mapping; it holds no actual data.
Q5. What is the default HDFS replication factor?	Three — each block is stored on three DataNodes by default.
Q6. What is YARN?	Yet Another Resource Negotiator — Hadoop's cluster resource-management and job-scheduling layer.
Q7. List the phases of MapReduce.	Map, Shuffle & Sort, and Reduce.
Q8. What is a combiner?	An optional map-side mini-reducer that aggregates locally to reduce the volume of data shuffled to reducers.
Q9. State the CAP theorem.	A distributed store can guarantee at most two of Consistency, Availability and Partition tolerance at the same time.
Q10. What does BASE stand for?	Basically Available, Soft state, Eventual consistency.
Q11. What is HBase?	A distributed, column-family NoSQL database built on HDFS, modelled on Google Bigtable, for real-time random read/write on huge tables.
Q12. What is HiveQL?	Hive's SQL-like query language for analysing data stored in Hadoop.
Q13. Difference between partitioning and bucketing in Hive?	Partitioning splits data into directories by a column value; bucketing hashes rows into a fixed number of files.
Q14. What is an RDD?	A Resilient Distributed Dataset — Spark's immutable, partitioned, fault-tolerant collection processed in parallel.
Q15. Transformation vs action in Spark?	Transformations are lazy and define new RDDs (map, filter); actions trigger execution and return results (count, collect).
Q16. Why is Spark faster than MapReduce?	Spark keeps intermediate data in memory and optimises a DAG, avoiding the repeated disk I/O of MapReduce.
Q17. What is Sqoop used for?	Bulk transfer of structured data between relational databases and Hadoop.
Q18. Name the four types of analytics.	Descriptive, Diagnostic, Predictive and Prescriptive.

Question	Answer
Q19. What is Spark MLlib?	Spark's scalable machine-learning library for classification, regression, clustering and recommendation.
Q20. Give one ethical concern in big data.	Protecting individual privacy and avoiding biased or discriminatory models when using personal data.

Glossary of Key Terms

Term	Meaning
Big Data	Datasets too large/fast/varied for traditional tools.
Cluster	A group of networked machines working as one system.
Commodity hardware	Inexpensive, standard servers used at scale.
DataNode	HDFS slave that stores actual data blocks.
DataFrame	A distributed table with a schema in Spark.
HDFS	Hadoop Distributed File System.
Lineage	Record of transformations used to rebuild lost RDD partitions.
MapReduce	Distributed batch model of map then reduce.
Metastore	Hive's catalogue of table schemas/metadata.
NameNode	HDFS master holding filesystem metadata.
NoSQL	Non-relational databases for scale and flexible schemas.
Partition	A logical slice of data or a directory split in Hive.
RDD	Resilient Distributed Dataset — Spark's core abstraction.
Replication	Storing multiple copies of a block for fault tolerance.
Schema-on-read	Structure applied when data is read, not written.
Shuffle	Network transfer of intermediate data between map and reduce.
YARN	Hadoop's resource manager and scheduler.

Common Abbreviations

Abbreviation	Full form
HDFS	Hadoop Distributed File System
YARN	Yet Another Resource Negotiator
RDD	Resilient Distributed Dataset
CAP	Consistency, Availability, Partition tolerance
ACID	Atomicity, Consistency, Isolation, Durability
BASE	Basically Available, Soft state, Eventual consistency
ETL	Extract, Transform, Load
IoT	Internet of Things
ML	Machine Learning
RM / NM / AM	ResourceManager / NodeManager / ApplicationMaster

End of study material. This document is intended as a learning aid; always cross-check specifics (default values, ports, versions) against the official Apache documentation and your course syllabus, as they evolve over time.