Big Data

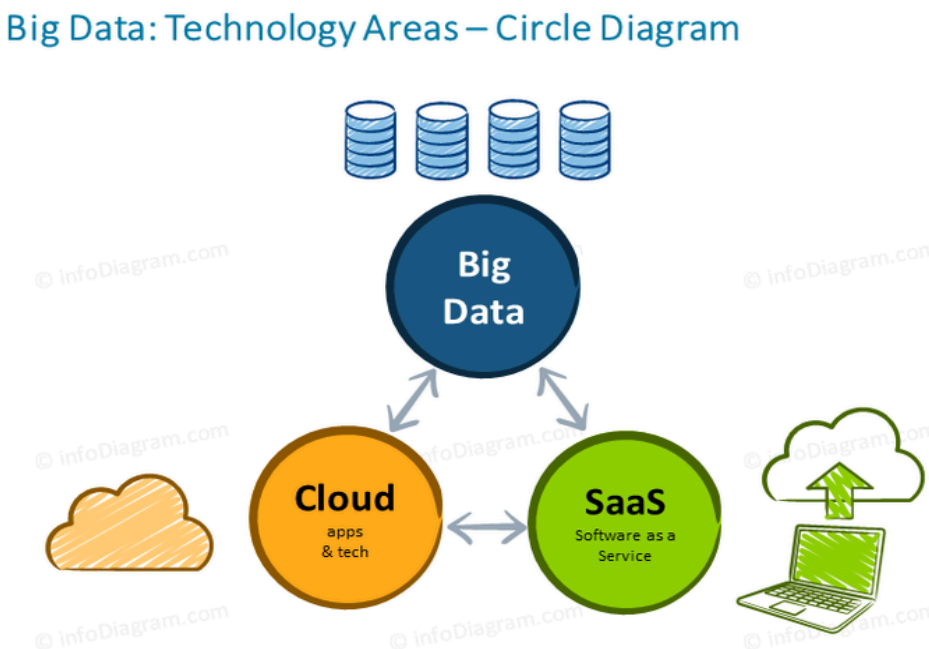
Big Data Overview

---

1. What's Big Data?

Definition:
Big Data refers to **extremely large, complex, and diverse datasets** that cannot be easily captured, stored, processed, or analyzed using traditional data management tools (like RDBMS).

These datasets come from sources like social media, IoT devices, sensors, financial transactions, healthcare systems, e-commerce, and more.



Key point: Big Data isn't just about "large size," but also about **complexity, variety, and processing speed**.

Example:

- Facebook generates **4+ petabytes of data per day** (posts, videos, likes, clicks).
- A jet engine generates **1 TB of data in 3 hours of flight**.

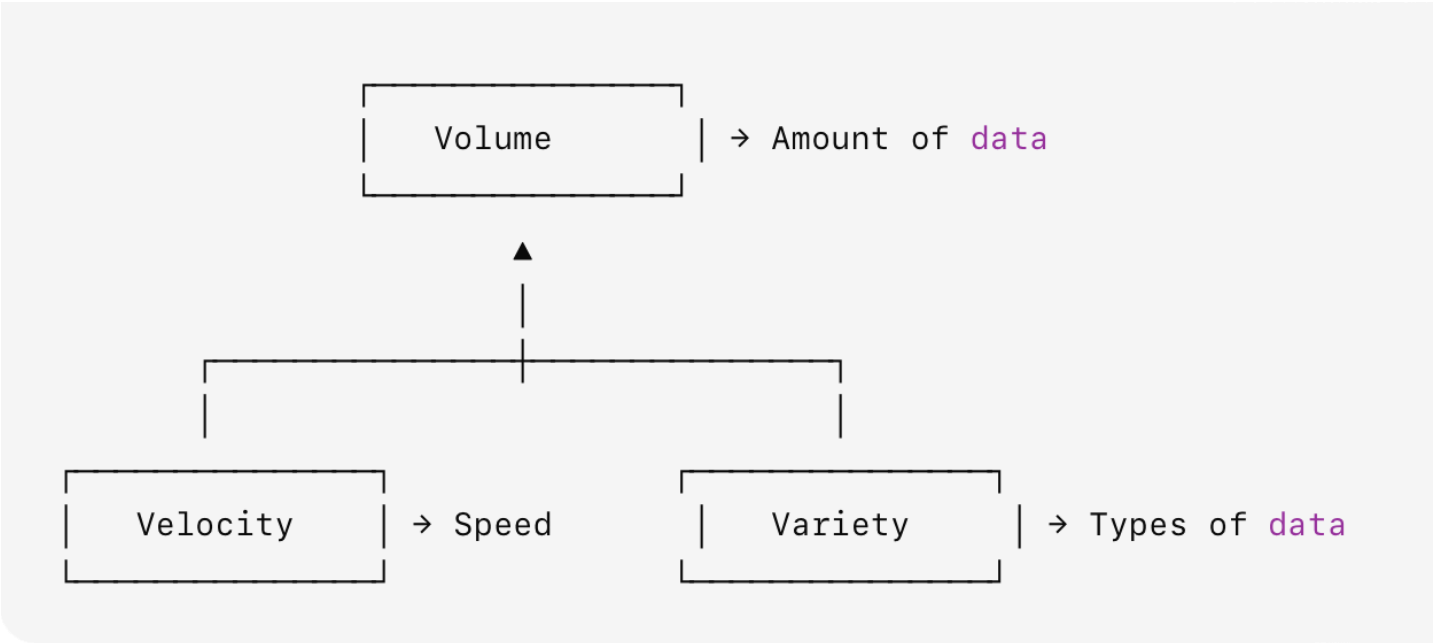
---

2. Big Data: The 3 V's

The **core characteristics** of Big Data are often described using the **3V model**:

1. **Volume** – Massive amount of data (TB → PB → EB scale).
    - Example: YouTube uploads **500+ hours of video every minute**.
2. **Velocity** – Speed at which data is generated and processed.
    - Example: Stock trading platforms process millions of transactions **per second**.
3. **Variety** – Different types of data (structured, semi-structured, unstructured).
    - Structured: Database tables (customer records).
    - Semi-structured: JSON, XML logs.
    - Unstructured: Images, videos, audio, emails.

Diagram: 3V's of Big Data



---

3. Explosion of Data

Data growth is happening at an **exponential rate**:

- **Past**: Data mostly came from structured business transactions (databases).
- **Now**: Social media, IoT devices, sensors, cloud apps, GPS, e-commerce clicks, healthcare records.
- **Future**: AI, self-driving cars, smart cities → generating **zettabytes (ZB)** of data.

Example:
IDC reports that the **global datasphere will reach 175 ZB by 2025**.

---

4. What's Driving Big Data?

- **IoT Devices & Sensors**: Billions of devices streaming real-time data.
- **Social Media**: Billions of users sharing posts, videos, likes.
- **Mobile & Cloud Computing**: Always-connected apps generating data.
- **Cheap Storage**: Cloud providers (AWS, GCP, Azure) make it easy to store petabytes of data.
- **Advanced Analytics & AI**: Need for insights from large, complex datasets.

Example:

- Amazon tracks **every click** you make for personalized recommendations.
- Hospitals collect **real-time patient vitals** via wearable devices.

---

5. Applications for Big Data Analytics

1. **Retail** → Recommendation engines (Amazon, Flipkart).
2. **Finance** → Fraud detection, real-time risk analysis.
3. **Healthcare** → Predicting disease outbreaks, patient monitoring.
4. **Manufacturing** → Predictive maintenance using IoT sensors.
5. **Smart Cities** → Traffic management, energy optimization.
6. **Entertainment** → Netflix uses viewing history to recommend shows.

---

6. Big Data Use Cases

- **Fraud Detection** (Banking) – Detect unusual transactions instantly.
- **Predictive Analytics** (Healthcare) – Forecast patient readmission risks.

- **Customer 360° View** (Retail) – Personalized offers & recommendations.
- **Supply Chain Optimization** (Logistics) – Reduce delays, track shipments.
- **Sentiment Analysis** (Social Media) – Track customer feedback & brand reputation.

---

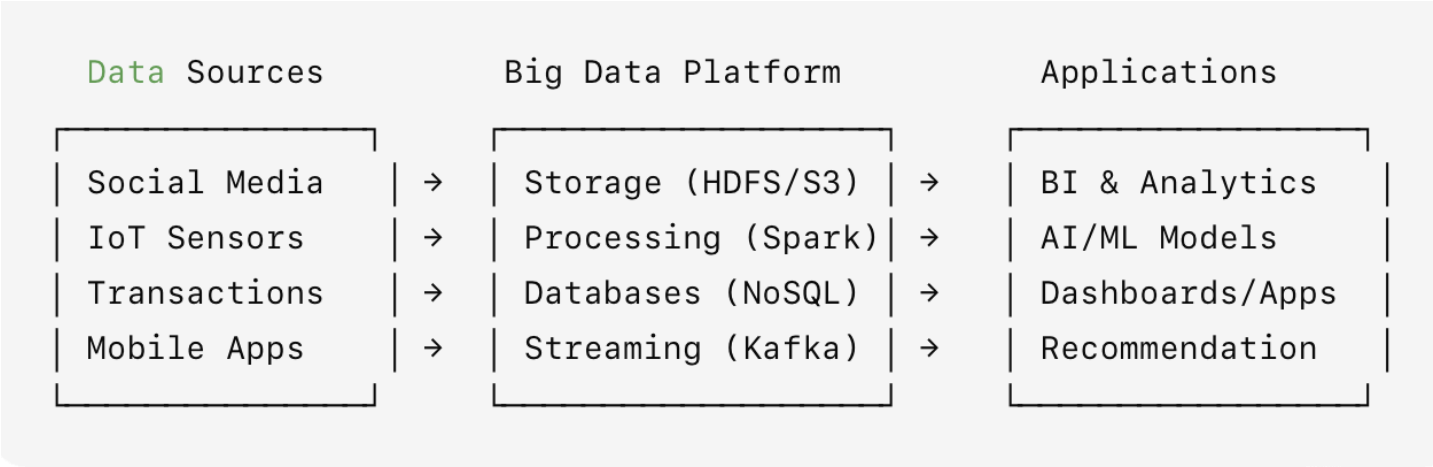
7. Benefits of Big Data

✔ **Better Decision-Making** – Data-driven insights → faster and more accurate decisions.
✔ **Improved Customer Experience** – Personalized recommendations & services.
✔ **Operational Efficiency** – Automating processes, reducing downtime.
✔ **Fraud & Risk Reduction** – Detect threats in real time.
✔ **Innovation** – Discovering new products, services, and revenue streams.

Example:

- Netflix saves **$1B+ annually** by using Big Data to recommend content.
- UPS saves **millions of gallons of fuel** using route optimization algorithms.

High-Level Diagram of Big Data Ecosystem



---

1) Brief History of Hadoop

Origins

- **2003**: Google publishes the **GFS (Google File System)** paper.
- **2004**: Google publishes **MapReduce**.
- **2002–2005**: The **Nutch** web-crawler project (Doug Cutting, Mike Cafarella) needs a web-scale store & compute → early HDFS & MapReduce clones.
- **2006–2008**: Yahoo! invests heavily; Hadoop migrates to the **Apache Software Foundation** → **Top-Level Project (2008)**.
- **2011**: Hadoop **1.x GA** (MapReduce v1 with JobTracker/TaskTracker).
- **~2013**: Hadoop **2.x** introduces **YARN** (decouples resource management from processing).
- **2017+**: Hadoop **3.x** adds **erasure coding**, **multiple active NameNodes (federation/HA)**, containerization improvements, and more.

Why it mattered: It democratized Google's ideas for commodity hardware clusters, making petabyte-scale analytics feasible for everyone.

---

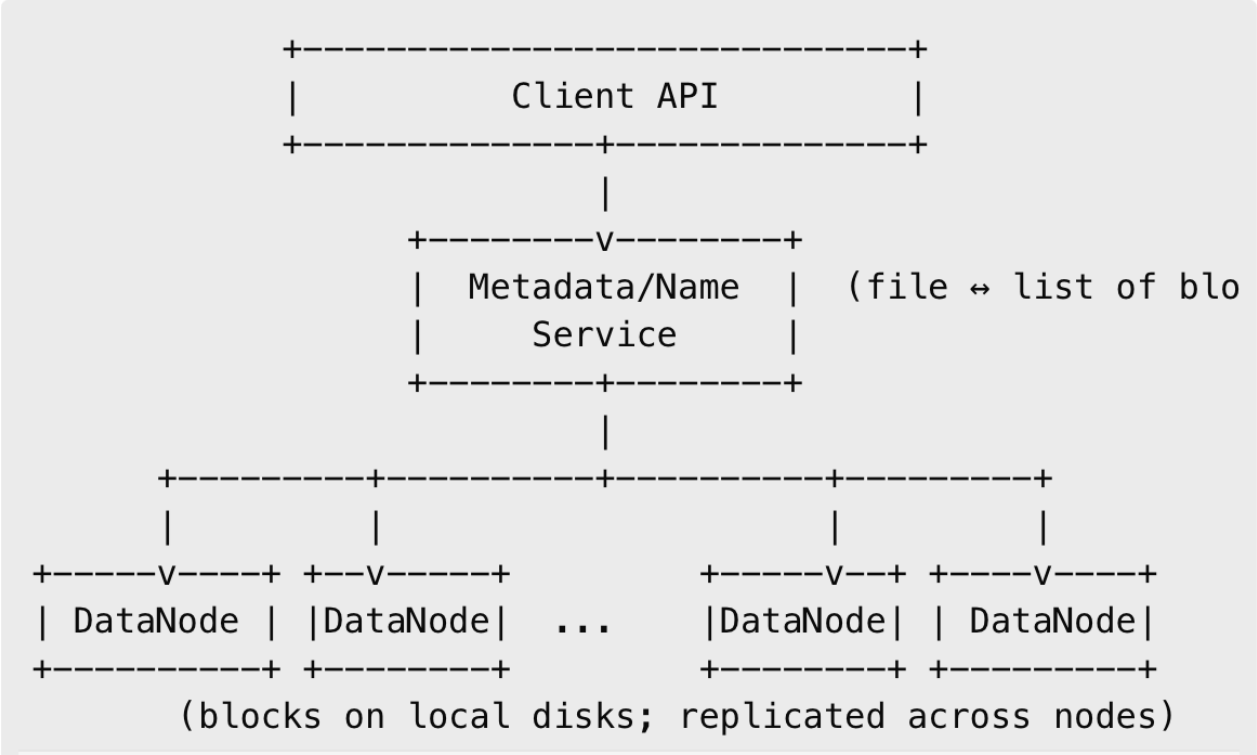
2) What is a Distributed File System (DFS)?

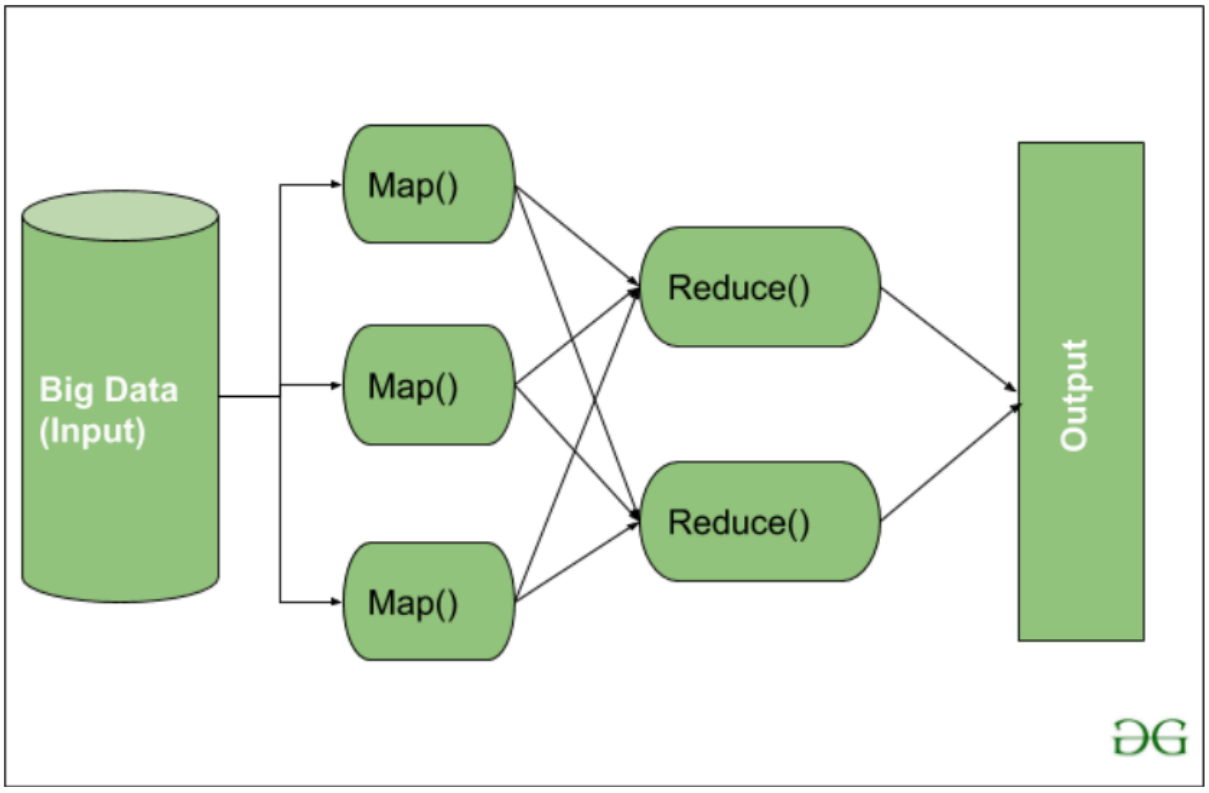
A **DFS** stores files across many machines but exposes a **single namespace** to clients.

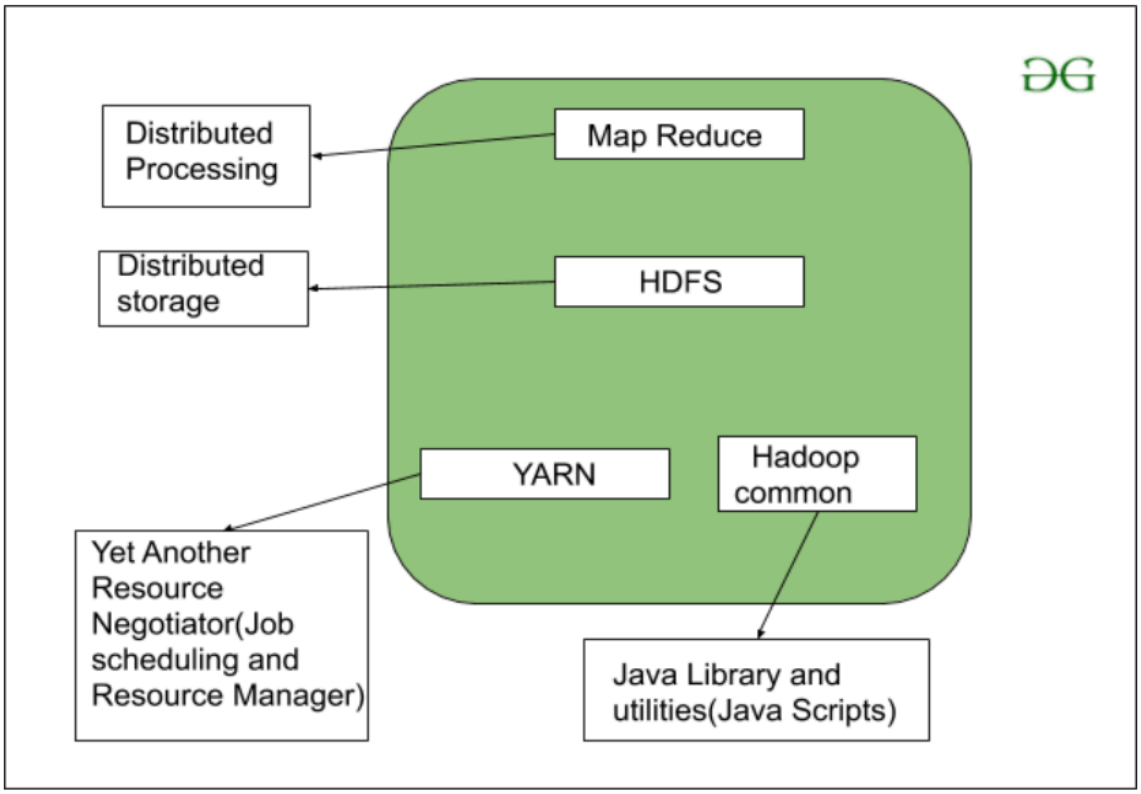
Key properties

- **Data partitioning** into **blocks** spread across nodes.
- **Replication** for fault tolerance.
- **Name service** (metadata) separated from **data service** (blocks).
- **Scale-out**: add nodes to increase capacity and throughput.

Concept Diagram (generic DFS)







---

3) What is Hadoop?

Apache Hadoop is an open-source framework for **reliable, scalable, distributed computing** on commodity hardware.

Core modules

- **HDFS** — Hadoop Distributed File System (storage).
- **YARN** — Yet Another Resource Negotiator (cluster resource manager & scheduler).
- **MapReduce** — Batch processing framework (one of many engines on YARN).
- **Hadoop Common** — shared utilities & libraries.

Ecosystem (runs on/with Hadoop)

- **Hive** (SQL on data lake), **HBase** (NoSQL), **Pig** (dataflow), **Oozie/Airflow** (orchestration), **Sqoop**(RDBMS↔HDFS), **Flume/Kafka** (ingest), **Spark/Flink** (compute), **ZooKeeper** (coordination), **Ranger/Atlas**(security & governance).

---

4) Characteristics of Hadoop (HDFS + YARN + Ecosystem)

- **Scalability**: horizontal scale to thousands of nodes.
- **Fault tolerance**: block replication, heartbeat health checks; task re-execution.
- **High throughput**: large block size, streaming I/O, data locality (move compute to data).
- **Cost-effective**: commodity hardware; cloud object stores as alternative backends.
- **Schema-on-read**: store raw data first, interpret later (esp. with Hive/Spark).
- **Write-once, read-many** semantics** (appends supported; random updates discouraged).

---

5) RDBMS vs Hadoop (Hive)

Capability	Traditional RDBMS	Hadoop + Hive (Data Lake SQL)
Data model	Strict schemas, normalized tables	Semi-structured/unstructured supported; schema-on-read
Workloads	OLTP (short transactions)	OLAP/ELT (scans, aggregates over huge data)
Latency	Milliseconds	Seconds → minutes (interactive engines can be sub-second with caches)
Scale	Vertical/limited horizontal	Massive horizontal scale

Transactions	**Strong ACID**	**Hive supports ACID for transactional tables (e.g., ORC) but typically for analytics**
Storage	**Proprietary files on SAN/NAS**	**Distributed blocks on HDFS / object store**
Indexing	**Rich indexes**	**Partitioning/bucketing; stats, predicate pushdown via file formats**
Cost	**Often high license/hardware**	**Commodity/cost-optimized/cloud**

When to use what?

- **RDBMS** for **OLTP** (orders, payments, inventory updates).
- **Hadoop/Hive** for **large-scale analytics** (clickstreams, logs, IoT, historical facts).

Hive sample

```
-- Create a partitioned ORC table
CREATE TABLE sales (
  order_id BIGINT,
  customer_id BIGINT,
  amount DECIMAL(12,2),
  channel STRING
)
PARTITIONED BY (dt DATE)
STORED AS ORC
TBLPROPERTIES (
  'transactional'='true'  -- ACID table
);

-- Load from staged data (external table or files)
INSERT INTO TABLE sales PARTITION (dt='2025-08-15')
SELECT order_id, customer_id, amount, channel
FROM stage_sales_2025_08_15;

-- Query with partition pruning
SELECT channel, SUM(amount) AS revenue
FROM sales
```

```
WHERE dt BETWEEN DATE '2025-08-01' AND DATE '2025-08-15'
GROUP BY channel;
```

---

6) ETL vs ELT

ETL (Extract → Transform → Load)

- Transform data **before** loading into the warehouse.
- Common in legacy DWH appliances; transformation on ETL servers.

ELT (Extract → Load → Transform)

- Load raw data first (to **HDFS/data lake**), then transform **in-place** using engines (Hive/Spark/Flink).
- Suits semi/unstructured and very large datasets; keeps raw history.

Diagram

```
ETL:  Sources -> [ETL Tool] -> Curated DW
ELT:  Sources -> Data Lake (HDFS/Object Store) -> [Hive/Spark SQL] -> Curated Models
```

Example (ELT with Sqoop + Spark SQL)

```
# 1) Extract & Load from MySQL to HDFS (as Parquet) via Sqoop
sqoop import \
  --connect jdbc:mysql://db:3306/retail \
  --username etl --password *** \
  --table orders \
  --as-parquetfile \
  --target-dir /data/raw/orders/dt=2025-08-15 \
  --m 4
-- 2) Transform in-place using Spark SQL / Hive
CREATE TABLE bronze_orders USING parquet LOCATION '/data/raw/orders';
CREATE TABLE silver_orders USING parquet AS
SELECT *, amount * 1.18 AS amount_gst FROM bronze_orders;
```

---

7) Hadoop Generations (High Level)

1. **Hadoop 1.x (MRv1)**
    - **JobTracker/TaskTracker** (single JobTracker → bottleneck).
    - MapReduce tightly coupled to cluster resource management.
2. **Hadoop 2.x (YARN)**
    - **YARN** separates **ResourceManager/NodeManager** from compute frameworks.
    - Multiple engines (MapReduce v2, Tez, Spark, etc.) can share the cluster.
3. **Hadoop 3.x**
    - **Erasure Coding** (storage efficiency vs replication).
    - **Federation/HA improvements** (multiple NameNodes).
    - **GPU & container enhancements**, timeline service v2, more.

---

8) Components of Hadoop (Core + Ecosystem)

Core: HDFS, YARN, MapReduce, Common.

Data Ingestion: Sqoop, Flume, Kafka Connect.

Processing Engines: MapReduce, Spark, Tez, Flink, Hive (on Tez/Spark), Presto/Trino.

Storage/Formats: HDFS, HBase; file formats **Parquet/ORC/Avro**.

Workflow/Orchestration: Oozie, Airflow, NiFi.

Security/Governance: Kerberos, Ranger, Knox, Atlas.

Coordination: ZooKeeper.

---

9) HDFS Blocks & Replication

- **Block size**: typically **128 MiB** (older clusters: 64 MiB). Configurable per file or cluster.
- **Replication factor**: default **3** (1 primary + 2 replicas). Per-file configurable.
- **Placement policy**: replicas spread across racks/nodes to survive failures.
- **Checksums**: each block has checksums for integrity; clients verify on read.

Diagram — file → blocks → replicas

File: /data/logs/app.log (400 MiB)
Split into 4 blocks (example 100 MiB each for illustration)

B1 -> DN1 (rack A), DN3 (rack B), DN5 (rack B)
B2 -> DN2 (rack A), DN4 (rack B), DN6 (rack B)
B3 -> DN3 (rack B), DN5 (rack B), DN1 (rack A)
B4 -> DN4 (rack B), DN2 (rack A), DN6 (rack B)

Set per-file replication

```
hdfs dfs -setrep -w 2 /data/logs/app.log
```

Write with a custom block size

```
# Example: 256 MiB block size while putting a file
hdfs dfs -D dfs.blocksize=$((256*1024*1024)) -put bigfile.parquet
/data/warehouse/
```

---

10) How Files Are Stored (Write & Read Path)

Write Path

1. Client asks **NameNode** for block locations.
2. NameNode returns a **pipeline** of DataNodes for each block.
3. Client streams data in **packets** to DN1 → DN1 forwards to DN2 → DN2 to DN3 (replication pipeline).
4. **Acks** flow back DN3 → DN2 → DN1 → client; on failure, pipeline reconfigures.
5. NameNode records block metadata; DataNodes send **block reports** periodically.

Read Path

1. Client asks NameNode for block locations.
2. Reads from the **closest replica** (data locality).
3. Checksums verified; if corrupted, fallback to another replica.

ASCII Flow

Client -> **NameNode** : need locations for /path/file

Client -> **DN1 -> DN2 -> DN3** : data packets (**write**)
DN3 -> DN2 -> DN1 -> Client : **acknowledgements**

---

11) HDFS Commands (Cheat Sheet)

Namespace & navigation

```
hdfs dfs -ls /                  # list
hdfs dfs -ls -R /data           # recursive list
hdfs dfs -mkdir -p /data/raw    # make dirs
hdfs dfs -rm -r -skipTrash /tmp/x  # delete
```

Copy & view

```
hdfs dfs -put local.csv /data/raw/
hdfs dfs -get /data/raw/local.csv ./
hdfs dfs -cat /data/raw/local.csv | head
hdfs dfs -text /data/raw/file.gz   # view compressed text
```

Space & replication

```
hdfs dfs -du -h /data/warehouse
hdfs dfs -df -h /
hdfs dfs -setrep -w 2 /data/warehouse/table.parquet
```

Admin & health

```
hdfs dfsadmin -report           # cluster summary
hdfs dfsadmin -safemode get|enter|leave
hdfs fsck / -files -blocks -locations  # check consistency
```

---

12) Hadoop Daemons (Processes) & How to Start/Stop

HDFS Layer

- **NameNode (NN)**: metadata, namespace, block mapping, permissions.
- **Secondary NameNode / Checkpoint Node**: merges **FsImage + EditLogs** into checkpoints (not a hot standby).

- **DataNode (DN)**: stores blocks, serves read/write, sends heartbeats/block reports.

YARN Layer

- **ResourceManager (RM)**: global resources, scheduler.
- **NodeManager (NM)**: per-node container executor & monitor.
- **ApplicationMaster (per app)**: negotiates resources, tracks app tasks.

MapReduce Service

- **JobHistoryServer**: retains finished job metadata & UI.

Data Structure Types

Structure Type	Examples
Structured	Tabular data (databases, CSV, Excel)
Semi-Structured	JSON, XML, Email, HTML
Unstructured	Logs, Images, Videos, Audio, Social Media posts

Data Frequency

Frequency Type	Description	Example Use Cases
Real Time (Streaming)	Data is processed instantly as it arrives	Financial trading, IoT sensors
Near Real Time	Data processed with minimal delay (seconds or minutes)	Monitoring, dashboards
Batch	Data is collected and processed at regular intervals	ETL jobs, backups, reporting

Types of Files

File Type	Description
Fixed Width	Columns have a set width, no delimiters
Delimited	Columns separated by a character (e.g., CSV, TSV)
Mainframe files	EBCDIC encoded files from mainframe systems
AVRO / ORC / PARQUET	Columnar storage formats used in big data

Compression Techniques

Compression Type	Description
Gzip	Standard, widely used file compression (works on various formats)

File Level Compression	Compress the entire file as a single unit
Block Level Compression	Compress blocks of data separately (used in ORC, Parquet, etc.)

Partitioning of Data

Partitioning allows you to distribute data for parallel processing and scalability.

- Random Partitioning: Data is distributed randomly across partitions. Each data point is assigned to a partition without consideration of any attribute.
- Hash Partitioning: A hash function is applied to a key column, and the result determines which partition the data goes to. Ensures related data is colocated.

Example: Hash Partitioning with Python

Below is a simplified illustration using Python and pandas:

```
import pandas as pd
# Example data
data = {'user_id': [10, 22, 37, 42, 55, 60],
        'value': [1, 2, 3, 4, 5, 6]}
df = pd.DataFrame(data)
# Number of partitions
num_partitions = 3
# Hash partitioning
df['partition'] = df['user_id'].apply(lambda x: hash(x) % num_partitions)
# Show partitioned DataFrame
print(df)
```

Output:

Each row gets assigned a partition from 0, 1, or 2 depending on the hash value of `user_id` **modulo the number of partitions.**

Example: Random Partitioning with Python

```
import pandas as pd
import numpy as np
# Example data
data = {'user_id': [10, 22, 37, 42, 55, 60]}
df = pd.DataFrame(data)
# Number of partitions
num_partitions = 3
# Random partitioning
np.random.seed(0)  # For reproducibility
df['partition'] = np.random.randint(0, num_partitions, size=len(df))
# Show partitioned DataFrame
```

```
print(df)
```

Output:

Rows are assigned randomly to partitions, without considering the content.

Hadoop 1.0 vs Hadoop 2.0

Feature	Hadoop 1.x	Hadoop 2.x (YARN)
Processing Model	MapReduce only	YARN supports MapReduce & other engines
Resource Management	Done by JobTracker with MapReduce	Separated by YARN ResourceManager
NameNode	Single NameNode (single point of failure)	Multiple NameNodes (active/standby)
Scalability	Up to 4,000 nodes/cluster	Up to 10,000+ nodes/cluster
Fault Tolerance	Limited	Improved; automatic NameNode recovery
OS Support	Linux only	Linux and Windows
Data Processing Type	Batch only	Batch & real-time
Ecosystem	Limited	Broader ecosystem (Hive, HBase, Spark)
Containerization	Not available	Generic containers for multiple tasks
Storage	HDFS	HDFS Federation

Key New Components in Hadoop 2.x:

- YARN (Yet Another Resource Negotiator): Separates resource management from data processing.
- MapReduce v2: Now runs as an application on YARN, not as a cluster manager.
- HDFS Federation: Supports multiple NameNodes for scalability and fault tolerance.

Major Vendors (2025):

- Cloudera
- Hortonworks
- Amazon EMR
- Microsoft (Azure HDInsight)
- MapR
- IBM InfoSphere BigInsights

YARN/MRv2 — Main Components

Component	Role
ResourceManager (RM)	Allocates cluster resources to applications

NodeManager (NM)	Manages nodes, monitors resource usage
ApplicationMaster (AM)	Manages a single application's lifecycle
Container	Logical compute units where tasks (like mappers/reducers) run

How YARN Works:

1. Client submits job (application).
2. ResourceManager assigns resources (containers).
3. ApplicationMaster is launched in a container to manage job execution.
4. ApplicationMaster requests containers for map/reduce tasks.
5. NodeManagers on different nodes launch containers.
6. ApplicationMaster monitors/report progress to ResourceManager.
7. Upon completion, resources are released.

Hadoop 2.x Configuration Files

- core-site.xml: Core Hadoop configuration (e.g., filesystem settings).
- hdfs-site.xml: HDFS-specific settings (replication, name/data node config).
- yarn-site.xml: YARN settings (resource tracking, scheduling).
- mapred-site.xml: MapReduce job and application configuration.

Cluster Management & Monitoring

- Vendor tools: Cloudera Manager, Ambari (Hortonworks), MapR Control System.
- Open-source: Ganglia, Nagios (monitoring resource usage, job tracking).

Hadoop Downloads:
Available from the [Apache Hadoop official website] and through vendor distributions (Cloudera, Hortonworks, etc.).

MapReduce

What is Distributed Computing?
Splitting large data and computation tasks across multiple machines or nodes for parallel processing.
MapReduce Introduction
A programming paradigm for processing large-scale data by dividing work into Map and Reduce tasks.
Components:

- Mapper: Processes input data and emits key-value pairs.
- Reducer: Aggregates results from Mappers and produces output.
- Driver: Controls the overall job (submits, configures, monitors).

MapReduce Workflow

1. Input data split into blocks.
2. Mappers execute in parallel on each split.
3. Output of Mapper is shuffled and sorted.
4. Reducers aggregate results.
5. Final output written to HDFS.

Sample Word Count Code (Hadoop Streaming - Python)
Mapper:

```
import sys
for line in sys.stdin:
    for word in line.strip().split():
        print(f"{word}\t1")
Reducer:
python
import sys
from collections import defaultdict
word_count = defaultdict(int)
for line in sys.stdin:
    word, count = line.strip().split('\t')
    word_count[word] += int(count)
for word, count in word_count.items():
    print(f"{word}\t{count}")
Command to run:
```

```
hadoop jar /path/to/hadoop-streaming.jar \
  -input <input_dir> -output <output_dir> \
  -mapper <mapper.py> -reducer <reducer.py>
```

Suitable Use Cases:

- Batch analytics
- Log processing
- ETL tasks

Unsuitable Use Cases:

- Low-latency requirements
- Real-time analytics
- Interactive querying

Sqoop

Architecture
Sqoop bridges relational databases and Hadoop, leveraging MapReduce for parallelized data import/export.
Basic Syntax

- Import entire table:
  text
  sqoop import --connect jdbc:mysql://host/db --table tablename --username user --password pass --target-dir /hdfs/path

- Import results of a SQL query:
  text
  sqoop import --connect jdbc:mysql://host/db --username user \
  --password pass --query 'SELECT id, name FROM people WHERE $CONDITIONS' \
  --target-dir /hdfs/path --split-by id

- Import to Hive:
  sqoop import --connect jdbc:mysql://host/db --username user \
  --password pass --table tablename --hive-import --hive-table db.hivetable

- Export HDFS data to RDBMS:
  text
  sqoop export --connect jdbc:mysql://host/db --table tablename --username user \
  --password pass --export-dir /hdfs/path

Flume
Start a Flume Agent
Given a configuration file (agent.conf):

flume-ng agent --conf conf --conf-file agent.conf --name <agent_name>
Memory Channel Configuration
In configuration file:

```
agent.channels = memoryChannel
agent.channels.memoryChannel.type = memory
agent.channels.memoryChannel.capacity = 10000
agent.channels.memoryChannel.transactionCapacity = 3000
```

Hive Programming Overview

Hive lets you run SQL-like queries on HDFS data.

Create Database/Table

```
CREATE DATABASE sales;

USE sales;


CREATE TABLE orders (
  order_id INT, customer STRING, amount FLOAT
) ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' STORED AS TEXTFILE;
```

1. Load Data

   ```
   LOAD DATA INPATH '/data/orders.csv' INTO TABLE orders;
   ```

2. Query Data

   ```
   SELECT customer, SUM(amount) FROM orders GROUP BY customer;
   ```

3. Insert Into Another Table

   ```
   CREATE TABLE rich_customers AS
   SELECT customer, SUM(amount) as total
   FROM orders
   GROUP BY customer
   ```

```
HAVING total > 10000;
```

Explanation:

- Hive translates SQL queries into MapReduce, Tez, or Spark jobs.
- Able to join, aggregate, and filter data using familiar SQL syntax.
- Supports partitioning, bucketing, and custom UDFs for extensibility.