

# DATA ENGINEERING

Mahela Panduka Bandara

# Table of Contents

MySQL .....	20
Introduction to MySQL .....	20
What is MySQL? .....	20
Use Cases and Industry Adoption .....	20
MySQL vs Other RDBMS.....	21
Installing and Setting Up MySQL .....	21
Case Sensitivity in MySQL .....	23
Identifiers (Table Names, Column Names, Aliases).....	23
MySQL Architecture Deep Dive .....	25
Client Layer.....	25
Server Layer .....	26
Storage Layer .....	26
Storage Engines (InnoDB vs MyISAM).....	27
Query Execution Process .....	28
Database Design Principles.....	29
Understanding Relational Models .....	29
Normalization (1NF, 2NF, 3NF, BCNF).....	29
Relationships: One-to-One, One-to-Many, Many-to-Many.....	32
Data Types and Table Structures .....	34
Numeric Data Types .....	34
String Data Types.....	34
Date and Time Data Types .....	35
Creating and Altering Tables in MySQL.....	35
Creating Tables .....	35
Altering Tables.....	36
Constraints in MySQL.....	37
PRIMARY KEY .....	37
UNIQUE .....	38
FOREIGN KEY .....	38
DEFAULT, INDEX, and CANDIDATE KEY.....	40
DEFAULT.....	40
INDEX .....	40
CANDIDATE KEY.....	41
ENUMs, SETs, and JSON Fields in MySQL.....	42
ENUM (Enumeration).....	42
SET .....	42
JSON.....	42
CRUD Operations: INSERT (Create) .....	44

INSERT Statement .....	44
SELECT (Read).....	45
UPDATE .....	47
DELETE .....	48
Safe Deletion Techniques and Advanced Use Cases .....	49
Filtering and Sorting with WHERE and ORDER BY .....	51
WHERE Clause – Filtering Rows .....	51
ORDER BY Clause – Sorting Results.....	51
Using SQLite with Python .....	53
What is SQLite? .....	53
Installing SQLite .....	53
Connecting to a Database.....	53
Creating a Table .....	53
Inserting Data.....	53
Querying Data (SELECT).....	54
Updating Data .....	54
Deleting Data .....	54
Using with Context Manager (with) .....	54
SQLite Features Summary .....	55
Best Practices .....	55
Big Data Engineering.....	56
Foundations of Big Data .....	56
What is Big Data? .....	56
5 Vs of Big Data .....	56
Big Data vs Traditional Data .....	57
Characteristics of Big Data Systems.....	58
Data Lifecycle & Pipeline Overview.....	58
Data Storage Systems .....	60
File Systems.....	60
Data Formats .....	61
Data Lakes vs Data Warehouses .....	62
Database vs Data Warehouse vs Data Lake.....	64
Distributed Computing & Frameworks .....	66
Hadoop Ecosystem .....	66
Apache Spark.....	66
Apache Flink vs Spark.....	67
Python-Native Alternatives .....	68
Summary Table .....	69
ETL and ELT Pipelines in Data Engineering.....	70

What is ETL?.....	70
Stages of a Data Pipeline.....	73
Typical Stages in a Data Pipeline .....	73
Hadoop Architecture.....	76
What is Hadoop?.....	76
Hadoop Was Inspired by GFS.....	76
Core Components of Hadoop Architecture.....	76
Hadoop Components (Core Ecosystem).....	79
HDFS (Hadoop Distributed File System) – Storage Layer .....	79
MapReduce – Processing Layer.....	80
YARN (Yet Another Resource Negotiator) – Resource Management Layer.....	80
Hadoop Common – Utilities and Libraries.....	80
Apache Hive?.....	82
Key Features of Hive .....	82
Hive Architecture.....	82
Hive Components.....	83
HiveQL – Query Language .....	83
When to Use Hive .....	84
Real-World Use Cases .....	84
Hive vs SQL Databases .....	84
Summary .....	85
Apache Pig.....	85
What is Apache Pig? .....	85
Pig Latin – The Language .....	86
When to Use Apache Pig.....	87
Pig vs Hive .....	87
Pig Execution Modes.....	87
Summary .....	88
Apache Sqoop .....	88
What is Apache Sqoop? .....	88
Key Features .....	88
Common Sqoop Commands .....	89
Summary Table.....	90
Apache Oozie .....	91
What is Apache Oozie? .....	91
Key Features of Apache Oozie.....	91
Scheduling with Coordinator.....	93
Real-World Use Cases.....	93
Apache HBase .....	94

What is Apache HBase?.....	94
Key Components .....	95
HBase Data Model.....	95
HBase vs HDFS .....	96
HBase vs RDBMS.....	96
Real-World Use Cases.....	96
Summary .....	97
Apache Flume.....	97
What is Apache Flume? .....	97
Why Use Apache Flume? .....	97
Core Components of Flume .....	98
Flow.....	98
Flume Sources, Channels, and Sinks .....	98
Real-World Use Cases.....	100
Flume vs Kafka .....	100
Summary .....	100
What is Apache ZooKeeper?.....	101
Components .....	102
ZooKeeper in Action: Common Use Cases .....	103
How ZooKeeper Ensures Consistency .....	103
Hadoop Terminologies .....	105
File System .....	105
Block .....	106
Cluster.....	106
Node.....	107
Example Workflow in Hadoop .....	107
Summary Table .....	108
Process.....	108
Daemon Process .....	109
Metadata .....	109
Replication.....	110
HDFS Architecture .....	111
Main Components.....	111
How It Works: Step-by-Step .....	111
HDFS Architecture Diagram .....	112
Replication in HDFS.....	112
Properties of HDFS .....	113
Racks.....	113
HDFS Rack-Aware Block Placement Policy.....	114

NameNode vs DataNode .....	115
Temporary vs Permanent Node Failure in Distributed Systems .....	116
HDFS Configuration Property Changes and Their Effects.....	117
NameNode Failure in HDFS .....	118
NameNode vs Secondary NameNode vs Standby NameNode .....	120
HDFS High Availability (HA) Architecture .....	121
Read and Write in HDFS .....	124
HDFS Write Flow .....	124
HDFS Read Flow.....	126
Linux Commands .....	129
File Management Commands.....	129
File Types .....	130
Linux File Permissions Commands Table .....	130
Permission Bits Breakdown .....	130
File Mode Format Example.....	131
Numeric (Octal) Permissions .....	131
File Renaming in Linux.....	131
Linux Networking Commands Table.....	132
Mastering HDFS Commands .....	133
Setting Up Hadoop for HDFS .....	133
What Is HDFS Setup and Why Do We Need It? .....	133
Installing Prerequisites .....	133
Download and Configure Hadoop .....	134
Configure Hadoop Environment .....	134
Configure Hadoop XML Files .....	135
Format the NameNode (First Time Only).....	136
Starting and Stopping HDFS .....	136
Using the HDFS Shell.....	137
Basic HDFS Commands .....	137
General Syntax.....	137
-ls: List Files and Directories .....	138
-mkdir: Create Directories .....	138
-put: Upload Files to HDFS.....	139
-get: Download Files from HDFS .....	139
-cat: View Contents of an HDFS File .....	139
-touchz: Create Empty Files .....	140
Summary Table .....	140
Intermediate File Operations in HDFS .....	141
-cp: Copy Files within HDFS .....	141

-mv: Move or Rename Files .....	141
-du: Show Disk Usage of Files/Directories .....	141
-dus: Show Summary Disk Usage .....	142
-stat: Show File Status Information .....	142
-tail: Show Last Part of a File .....	142
-text: Convert File to Readable Format .....	143
<b>Directory Management in HDFS.....</b>	<b>144</b>
Recursive Directory Operations .....	144
-count: Count Files, Directories, and Space Used.....	144
Quota Management.....	145
Setting Permissions on Directories.....	146
<b>Permissions and Access Control in HDFS .....</b>	<b>148</b>
HDFS File Permission Model.....	148
-chmod: Change Permissions .....	148
-chown: Change Ownership.....	149
-chgrp: Change Group Ownership .....	149
Working with Superusers in HDFS .....	149
<b>Advanced File Operations in HDFS .....</b>	<b>151</b>
-appendToFile: Append to an Existing HDFS File .....	151
-setrep: Set Replication Factor for Files .....	151
-getmerge: Merge Multiple Files into One (Local) .....	152
-copyToLocal and -copyFromLocal .....	152
Working with Compressed Files in HDFS .....	153
Listing Files Using Wildcards (Globs / Regex) .....	153
Summary Table .....	154
<b>File System Metadata Operations in HDFS .....</b>	<b>154</b>
-checksum: Verifying File Integrity .....	154
File Modification Time and Status (-stat).....	155
File Block Locations .....	155
<b>HDFS Data Recovery and Troubleshooting.....</b>	<b>158</b>
Handling Corrupted Blocks .....	158
Enable Snapshots for Recovery.....	159
Common HDFS Errors and Fixes .....	160
<b>File System Administration in HDFS.....</b>	<b>161</b>
Safe Mode Operations (-safemode).....	161
Balancer: Rebalancing Data Across DataNodes .....	161
Reporting with dfsadmin -report.....	162
Checking File System Health (fsck) .....	162
Upgrading and Downgrading HDFS.....	163

Performance and Optimization in HDFS.....	164
Optimizing Replication Factor .....	164
Data Locality Awareness.....	164
HDFS Caching (In-Memory Data Caching) .....	165
HDFS in a Cloud-Native World.....	167
HDFS on Azure HDInsight .....	167
HDFS on Google Cloud Dataproc .....	167
Map Reduce .....	170
Data Locality and Code Locality .....	170
Map Reduce Architecture.....	173
What is MapReduce?.....	173
MapReduce Architecture Overview .....	173
Components in MapReduce Architecture .....	173
Zero Reducer .....	178
What is a Zero Reducer in MapReduce?.....	178
Input Splits .....	180
What are Input Splits in Hadoop MapReduce? .....	180
InputFormat and Input Splits.....	181
YARN .....	182
Introduction.....	182
What is YARN? .....	182
Google Cloud .....	186
Google Cloud Dataproc.....	186
Core Features .....	186
Architecture Overview .....	186
Components of a Dataproc Cluster.....	187
How It Works.....	187
Use Cases.....	188
Pricing Highlights.....	188
Summary .....	189
Guide.....	189
Higher Order Functions in Python.....	197
Example 1: Passing Functions as Arguments .....	197
Example 2: Returning Functions from Functions .....	198
Built-in Higher Order Functions in Python .....	198
1. map(function, iterable) .....	198
SPARK.....	200
Introduction to Apache Spark (Beginner-Friendly) .....	200
Why Do We Need Spark? .....	200

What is Apache Spark?.....	200
Common Questions.....	202
Do I need to know Hadoop and YARN to learn Spark?.....	202
Can Spark work with other file systems (not just HDFS)?.....	202
Some practical use cases of Spark? .....	202
Spark on Pseudo-distributed system vs Clusters? .....	202
How is Spark different from traditional databases (e.g., MySQL) or Pandas? .....	203
Why not just horizontally split MySQL database without using Spark?.....	203
Why use Databricks?.....	204
Limitations of MapReduce.....	204
Apache Spark Ecosystem.....	205
Optional Tools Often Used with Spark .....	206
Resilient Distributed Datasets .....	207
Data Sharing is Slow in MapReduce.....	207
Iterative Operations on MapReduce .....	207
Interactive Operations on MapReduce .....	208
Data Sharing using Spark RDD .....	208
Iterative Operations on Spark RDD .....	208
Interactive Operations on Spark RDD .....	209
Spark Architecture Deep Dive .....	210
Spark Components: Driver, Executors, Cluster Manager .....	210
Understanding DAG (Directed Acyclic Graph) .....	211
Logical Plan vs Physical Plan.....	211
Execution Flow & Fault Tolerance.....	212
Fault Tolerance in Spark .....	212
RDD – Resilient Distributed Dataset .....	214
Transformations vs Actions.....	215
Let's Dive Deeper: What Happens Under the Hood? .....	217
Core Properties of RDDs in Spark .....	221
Processing in Spark: Local vs Distributed vs Parallel .....	223
Examples Using <code>getNumPartitions()</code> in Spark .....	226
DataFrames, Datasets, and Optimizations.....	228
What are DataFrames and Datasets? .....	228
DataFrame vs Dataset — Simple Comparison .....	228
When Should You Use Each? .....	229
Spark's Superpowers: Optimizations .....	229
Spark Programming - Python (PySpark) .....	231
What is PySpark?.....	231
Why Use PySpark?.....	231

Basic PySpark Setup.....	231
SparkSession in PySpark .....	233
What is SparkSession? .....	233
Why is SparkSession Important? .....	233
How to Create a SparkSession .....	233
Example: Using SparkSession .....	234
Useful Methods in SparkSession .....	234
Real-Life Use Cases .....	234
SparkSession with Pandas .....	235
spark.sparkContext.parallelize().....	236
RDD Operations.....	238
Set Up PySpark and Pandas .....	238
More .....	242
Example : Employee Records .....	245
Narrow vs Wide Transformations .....	248
What Are Transformations in Spark?.....	248
Narrow Transformations .....	248
Wide Transformations .....	249
Working with HDFS in Spark .....	251
Jobs, Stages, and Tasks in Spark UI.....	254
Job .....	254
Stage .....	254
Task.....	255
Example .....	256
groupByKey() and reduceByKey() .....	259
Syntax.....	259
Under the Hood .....	259
groupByKey() .....	259
reduceByKey() .....	260
Performance: Why reduceByKey() Is Better .....	260
Pro Tip: Use DataFrames Instead .....	261
Text Diagram: groupByKey() .....	261
Text Diagram: reduceByKey() .....	262
Exercise .....	264
Number of partitions .....	267
Why Do Partitions Matter? .....	267
When to Increase or Decrease Partitions? .....	267
coalesce() and repartition() .....	269
Syntax.....	269

When to Use repartition() .....	269
When to Use coalesce() .....	270
Performance Tip: .....	270
Higher-Level APIs in Apache Spark .....	271
What Are Higher-Level APIs? .....	271
DataFrame API .....	271
Spark SQL Tables .....	272
Example .....	274
Spark DataFrame function .....	275
Reading Data from HDFS in Spark .....	279
Example .....	280
Schema Enforcement in Apache Spark .....	282
What is a Schema in Spark? .....	282
Why Schema Enforcement Matters .....	282
Two Modes of Schema Application .....	282
Schema Inference vs Schema Enforcement .....	282
How to Enforce Schema in Spark (with Example) .....	283
Strict Type Checking (Fail Fast) .....	283
Schema-on-Read vs Schema-on-Write .....	284
Schema Tools in Spark .....	284
DDL (Data Definition Language) Schema in Spark .....	285
Benefits of DDL Schema .....	287
Limitations of DDL Schema .....	287
Read Modes in Spark .....	288
1. PERMISSIVE Mode (default) .....	288
2. FAILFAST Mode .....	289
3. DROPMALFORMED Mode .....	289
Comparison Summary .....	289
writing a CSV file into HDFS using Apache Spark .....	290
Step-by-Step: Writing CSV to HDFS with PySpark .....	290
Handling data types in Spark DataFrames .....	292
Viewing Data Types in a DataFrame .....	292
Explicit Schema Definition .....	292
Cast Data Types (Change Types) .....	293
Complex Types .....	293
Date and Timestamp in Spark .....	295
Creating a DataFrame with Date or Timestamp .....	295
Common Date Functions in PySpark .....	295
Convert String to Date/Timestamp with Format .....	296

Filtering with Dates .....	296
Handling Nulls & Bad Formats .....	296
Sorting by Date .....	296
Writing Dates to Files (CSV, Parquet).....	296
Tips for Working with Dates .....	297
Spark SQL and Spark Tables .....	298
What is Spark SQL? .....	298
Why use Spark SQL?.....	298
Using SQL on Spark Data .....	298
Temporary Table (Session-Scoped) .....	298
Global Temporary Table .....	299
Persistent Table (Managed Table).....	299
Table Type Comparison Table .....	299
Spark SQL operations .....	301
Spark SQL Tables: Managed vs External .....	304
Key Differences .....	304
Example – Create Managed Table.....	304
Example – Create External Table .....	304
When to Use.....	305
DROP Behavior Demo.....	305
Caching and Persisting in Spark.....	307
Why Caching or Persisting Is Needed in Spark?.....	307
Cache vs Persist – Key Differences .....	307
Table: Spark Storage Levels .....	307
Visual Summary .....	308
How and Where Does Caching Happen in Spark? .....	309
Why Spark Uses Local Disk, Not HDFS for Caching? .....	309
When to Use Caching .....	309
When Not to Use Caching .....	310
Where Can You See Cached Data in Spark?.....	310
Example: Caching in PySpark .....	310
Does Caching Decrease Performance? .....	311
Best Practices .....	311
Caching RDDs .....	312
Caching Data-frames .....	313
Caching on Spark SQL tables .....	317
Spark Architecture .....	320
Spark Run Modes (Deployment Modes) .....	320
What Makes Spark Fast?.....	323

Spark Architecture Components .....	325
Spark Standalone Architecture .....	329
Spark Deployment Modes .....	333
Introduction to Databricks.....	336
What is Databricks?.....	336
Key Features .....	336
Supported Cloud Platforms .....	336
Databricks Architecture.....	336
Core Components of Databricks.....	337
Common Use Cases.....	338
Data Access.....	338
Databricks vs Alternatives .....	339
Control Plane vs Data Plane.....	339
Databricks File System (DBFS) .....	340
Additional Note I.....	341
Example DataFrame Creation .....	341
Using .withColumn() .....	341
Getting Unique Values .....	342
Pivoting Data.....	342
Using Window Functions .....	343
Aggregation — Oldest & Newest Customer per City .....	343
Summary Table .....	344
Spark Project 1 .....	345
Spark Project 2 : Brazilian E-Commerce Dataset.....	351
Data Exploration.....	352
Data Cleaning and Transformation.....	353
Data Integration and Aggregation .....	355
Spark Optimization: Broadcasting, Caching.....	356
The Join Operation in Spark .....	356
Broadcasting in Spark.....	356
Caching in Spark .....	356
Optimization Summary .....	357
Spark Session Builder Configs.....	358
Spark Optimization A–Z (Beginner Friendly) .....	360
Bucketing .....	360
Handling Data Skew .....	360
Predicate Pushdown .....	361
Column Pruning.....	361
Adaptive Query Execution (AQE) .....	361

Summary Cheat Sheet .....	362
GCP vs AWS vs Azure – Service Comparison Table .....	363
Spark SQL .....	365
Spark SQL Basics .....	365
SparkSession .....	365
DataFrame API vs SQL API .....	366
Loading Data into Spark SQL.....	366
DataFrame Operations .....	367
Viewing and Exploring Data .....	367
Putting It Together: Example .....	368
Spark SQL Data Types .....	369
Primitive Data Types .....	369
Complex Data Types.....	370
Schema Inference and Manual Schema Specification.....	372
Spark SQL – Working with SQL Queries [Transformations] .....	373
Sample DataFrame.....	373
Creating Temporary Views .....	373
Spark SQL Functions.....	378
Built-in Functions .....	378
Spark SQL Window Functions.....	381
Introduction .....	381
Syntax.....	381
Sample DataFrame.....	382
Example 2 – LEAD & LAG.....	383
Example 3 – Running Total .....	383
Spark SQL Optimization Techniques .....	384
Caching and Persistence .....	384
Broadcast Joins .....	384
Partition Pruning.....	384
Bucketing.....	384
Handling Skewed Data.....	385
Predicate Pushdown.....	385
Column Pruning .....	385
Adaptive Query Execution (AQE).....	385
Monitoring with Spark UI .....	386
Cost-Based Optimizer (CBO) Hints .....	386
Spark SQL – Time Series & Date Processing.....	387
Date and Timestamp Functions .....	387
Handling Time Zones .....	387

Extracting Date Parts .....	388
Window Aggregations on Time Series .....	388
<b>Spark SQL – Aggregations and Grouping.....</b>	<b>389</b>
Basic Aggregations with groupBy and agg .....	389
Multi-Column Grouping .....	390
Rollups and Cubes .....	390
Pivot Tables.....	391
<b>Spark SQL Best Practices .....</b>	<b>392</b>
Use Columnar Formats (Parquet, ORC).....	392
Partition Large Datasets for Parallelism.....	392
Avoid Wide Transformations When Possible .....	392
Use AQE for Dynamic Optimizations .....	393
Minimize Shuffles .....	393
Use Broadcast Joins for Small Tables .....	393
Use Caching Wisely .....	393
Summary Table of Best Practices .....	394
<b>Hive .....</b>	<b>395</b>
Introduction to Apache Hive .....	395
What is Hive? .....	395
History and Evolution.....	395
When to Use Hive? .....	395
Hive vs RDBMS (Differences).....	396
Hive Architecture.....	396
Simplified Flow of a Hive Query .....	397
Hive vs Spark .....	398
Architecture .....	398
Performance .....	398
Use Cases.....	399
Latency .....	399
Ecosystem .....	399
Summary Table .....	400
How Hive Makes Big Data Processing Easier .....	401
The Challenge of Big Data .....	401
The Solution: Hive.....	401
Hive Architecture (Simplified Flow) .....	401
Example – Without Hive (using raw MapReduce in Java) .....	401
Example – With Hive .....	402
Behind the Scenes.....	402
Common Questions & Misconceptions about Hive.....	403

“Hive is a database just like MySQL or PostgreSQL.” .....	403
“Hive can perform row-level transactions like SQL databases.” .....	404
“Hive works only on HDFS.” .....	404
“Hive is a real-time query engine like MySQL.” .....	405
“Hive stores data itself.” .....	405
“Hive is fast for all queries.” .....	405
“Hive can replace OLTP databases.” .....	406
“Hive only works with MapReduce.” .....	406
Connecting to Hive – CLI & Beeline .....	407
Hive CLI (Old way – hive command).....	407
Hive Beeline (Modern & Recommended) .....	407
Difference Between Hive CLI & Beeline.....	408
Hive with CSV File .....	408
Prepare a Sample CSV File.....	408
Hive Metadata .....	411
What is Hive Metadata? .....	411
Why is Metadata Important? .....	411
Hive Metastore .....	411
Accessing Hive Metadata.....	412
Best Practices for Metadata .....	413
Hive architecture .....	414
Hive Clients in Hive Architecture .....	414
Hive Server & Hive Driver .....	416
HiveServer (HS2 – HiveServer2).....	416
Hive Driver .....	417
HiveServer vs Hive Driver .....	418
Hive Clients, Hive Server, and Hive Driver .....	419
Comparison Table .....	421
Hive Compiler and Hive Optimizer .....	422
Hive Compiler .....	422
Hive Optimizer .....	423
Workflow Example.....	424
Summary .....	425
Hive Query Flow .....	425
User Submits Query via Hive Client .....	426
HiveServer2 Receives Query .....	426
Hive Driver – Query Lifecycle Manager.....	426
Hive Compiler – Parse, Analyze, Generate Plan.....	427
Hive Optimizer – Improve Performance .....	427

Execution Engine – Run on Hadoop/Spark .....	428
Hive Metastore Interaction.....	428
Results Return to Client .....	428
Complete Hive Query Flow Diagram .....	429
Advanced Hive Topics .....	430
Materialized Views in Hive.....	430
Hive LLAP (Live Long and Process) .....	430
Query Federation with Hive .....	431
Hive Performance Tuning: Tez vs Spark.....	431
Optimizing ORC and Parquet Storage .....	432
Working with Skewed Data in Hive.....	433
Summary Table .....	433
Apache Kafka .....	434
Introduction to Apache Kafka .....	434
What is Kafka?.....	434
History and Evolution (LinkedIn → Apache).....	434
Kafka vs Traditional Messaging Queues .....	434
Kafka Ecosystem Overview .....	435
Why Kafka is Popular .....	435
Key Features of Apache Kafka.....	436
Different Kafka Applications – Examples, Problems, and Alternatives .....	438
Real-Time Log Aggregation & Monitoring.....	438
E-Commerce Order Processing System.....	438
Banking & Financial Transactions.....	439
IoT Data Streaming (Smart Devices / Sensors) .....	440
Ride-Sharing / Food Delivery Apps (Uber, DoorDash, Swiggy, Zomato).....	440
Video Streaming Platforms (Netflix, YouTube).....	441
Data Lake Ingestion (ETL Pipelines) .....	442
Kafka Architecture .....	443
Kafka Brokers .....	443
Kafka Clusters.....	444
Topics .....	444
Partitions .....	444
Offsets .....	445
Visualizing Kafka Architecture .....	445
Summary .....	445
Kafka Partitioning.....	446
What is a Partition?.....	446
Why Partitioning is Important .....	446

How Kafka Assigns Messages to Partitions .....	446
Partitioning Rules / Considerations .....	447
Benefits of Partitioning.....	447
Partitioning Best Practices .....	448
Summary: .....	448
Kafka Producers .....	449
Asynchronous Sends .....	449
Acknowledgment Settings (acks) .....	449
Batch Processing.....	450
Compression .....	450
Producer Best Practices .....	450
Visual Overview.....	451
Summary .....	451
Kafka Consumers & Optimization .....	452
Batch Processing & Compression (Producer Side Recap).....	452
Kafka Consumers.....	453
Consumer Groups.....	453
Rebalancing .....	453
Offset Management.....	454
Summary .....	455
Kafka Replication.....	456
What is Kafka Replication?.....	456
Example Scenario.....	456
Kafka Architecture Components Recap .....	458
Example Scenario.....	458
Kafka Workflow .....	458
Zookeeper vs KRaft in Kafka metadata management .....	461
Overview .....	461
ZooKeeper-Based Kafka.....	461
KRaft-Based Kafka (Kafka Raft Mode) .....	462
Key Differences: ZooKeeper vs KRaft .....	463
Why KRaft?.....	463
Run kafka .....	464
Local Installation.....	464
Docker .....	464
Kubernetes / Cloud Native .....	465
Kafka Modes .....	465
Summary Table .....	466
Confluent Kafka .....	467

Cluster Creation.....	467
Topic Creation.....	468
Data Contracts.....	471
Build a client .....	473
Send All data.....	476
Kafka Partitioning – How Messages are Assigned to Partitions .....	477
Kafka Producer Functions Explained.....	478
Delivery Guarantee Levels .....	479
Confluent Kafka Connectors .....	480
What is Kafka Connect?.....	480
What is a Kafka Connector? .....	480
How Kafka Connect Works .....	480
Why use Kafka Connect? .....	481
Confluent Cloud Connectors .....	481
Example Workflow.....	481
Delivery Guarantees .....	482
Configuring a Connector.....	482
Advantages & Limitations .....	483
Custom Connectors .....	483

# MySQL

## Introduction to MySQL

### What is MySQL?

MySQL is an open-source Relational Database Management System (RDBMS) based on Structured Query Language (SQL). It enables users to store, manage, retrieve, and manipulate data using a structured schema defined by tables, rows, columns, and relationships.

Key Characteristics:

- Open-source (under the GNU General Public License)
- Developed in C and C++
- Cross-platform: supports Windows, Linux, macOS
- Uses client-server architecture
- Default storage engine: InnoDB

MySQL is known for its:

- Speed and reliability
- Scalability from small to large-scale applications
- Ease of use and low learning curve
- Large community and extensive documentation

### History and Evolution

MySQL History Timeline	
<b>1995</b>	Developed by Michael Widenius and David Axmark at MySQL AB (Sweden/Finland)
<b>1996</b>	First internal release
<b>2000</b>	Became fully open-source under GNU GPL
<b>2008</b>	Acquired by Sun Microsystems
<b>2010</b>	Sun Microsystems was acquired by Oracle Corporation
<b>2013+</b>	Forks like MariaDB emerged due to concerns about Oracle's control
<b>Today</b>	Continues as a major open-source database, maintained by Oracle, with active updates and community support

## Use Cases and Industry Adoption

MySQL is widely used across industries and by a broad range of applications:

### Common Use Cases

- Web Applications (WordPress, Joomla, Drupal)
- E-commerce Platforms (Magento, Shopify)
- Social Media and SaaS Platforms
- Content Management Systems
- Data Warehousing (with proper optimization)
- Mobile apps and IoT backend databases

### Popular Companies Using MySQL

- Facebook – heavily customized MySQL clusters
- Twitter – backend services
- Airbnb – user data and listing systems
- Uber – transactional systems
- Netflix – some microservices and analytics
- MySQL's flexibility makes it suitable for:
- Startups looking for cost-effective solutions
- Enterprises requiring high-availability and scalability

- Hybrid cloud and on-premises systems

## MySQL vs Other RDBMS

Feature	MySQL	PostgreSQL	Oracle DB	SQL Server
License	Open-source (GPL)	Open-source (PostgreSQL License)	Commercial	Commercial
Performance	Fast reads, great for web apps	Advanced features, complex queries	High performance, enterprise-ready	Optimized for Windows systems
ACID Compliance	Yes (with InnoDB)	Yes	Yes	Yes
Advanced Features	Limited JSON/NoSQL support	Full JSON/NoSQL, custom types	PL/SQL, full Oracle suite	Integration with Microsoft tools
Community Support	Strong	Strong	Limited (paid)	Moderate
Learning Curve	Easy	Moderate	Steep	Moderate

### Summary

- Use MySQL when you need simplicity, speed, and reliability for web or medium-sized applications.
- Choose PostgreSQL for advanced data modeling, complex queries, or data science needs.
- Oracle is preferred in large enterprises needing extensive features and support.
- SQL Server is optimal for Microsoft-centric environments.

## Installing and Setting Up MySQL

### Installation on Windows, Linux, macOS

#### Windows

1. Download the MySQL Installer from <https://dev.mysql.com>.
2. Choose:
  - **Developer Default** (includes MySQL Server, Workbench, Shell)
  - **Server only** for minimal setup
3. Run the installer:
  - Select version (typically MySQL 8.x)
  - Set root password
  - Configure server: port (default 3306), service name
  - Add MySQL to system PATH
4. Verify installation:

```
mysql -u root -p
```

#### Linux (Ubuntu/Debian)

```
sudo apt update
sudo apt install mysql-server
sudo mysql_secure_installation
```

- To start the MySQL service
- sudo systemctl start mysql



- Recommended: Install via **Homebrew**

```
brew update  
brew install mysql  
brew services start mysql  
mysql -u root
```

#### 💡 Post-install

- Use `mysql_secure_installation` to remove test DB, disable remote root access, set password strength.

## MySQL Workbench and CLI Tools

### 💻 MySQL CLI (Command Line Interface)

- Core CLI tool for managing MySQL

```
mysql -u root -p
```

- Common CLI commands:
- `SHOW DATABASES;`
- `CREATE DATABASE school;`
- `USE school;`
- `SHOW TABLES;`

### ❖ MySQL Workbench

- Official GUI tool for MySQL (available for all major platforms)
- Features:
  - Visual database design
  - SQL query editor
  - Server status and configuration
  - Data export/import
  - User management

⌚ *Tip:* Workbench is ideal for beginners and for managing complex schemas visually.

#### 🔌 Connecting to the Server

##### From CLI (local):

```
mysql -u root -p
```

##### Remote connection:

```
mysql -h <hostname_or_ip> -u <username> -p
```

## Using Workbench

- Open Workbench → New Connection
- Enter:
  - Connection name
  - Hostname: 127.0.0.1 or remote IP
  - Port: 3306 (default)
  - Username: root or other
  - Test connection → Save

##### Note:

- Ensure remote access is enabled in `mysqld.cnf`
- Open port 3306 in firewall if accessing remotely

## Configuration Basics (my.cnf / my.ini)

### Configuration Files:

- Linux/macOS: /etc/mysql/my.cnf or /etc/my.cnf
- Windows: C:\ProgramData\MySQL\MySQL Server X.Y\my.ini

### Common Settings:

```
[mysqld]
port = 3306
bind-address = 127.0.0.1
max_connections = 200
sql_mode = STRICT_ALL_TABLES
innodb_buffer_pool_size = 512M
```

Restart MySQL after config changes:

```
sudo systemctl restart mysql # Linux
brew services restart mysql # macOS
```

### Helpful Utilities:

- mysqladmin: for server status, flush, shutdown
- mysqldump: for backups

## Case Sensitivity in MySQL

MySQL's **case sensitivity** depends on **what you're working with**:

- **Identifiers** (table names, column names)
- **Data** (text content)
- **Collations** (character set comparison rules)
- **Operating System** (for file-based identifiers)

### Identifiers (Table Names, Column Names, Aliases)

#### Case Sensitivity Depends on OS:

OS	Table Names Case-Sensitive?
Linux/Unix	 Yes (case-sensitive)
Windows/macOS	 No (case-insensitive)

- MySQL stores tables as **files**, so it follows OS filesystem rules.
- **Column names and aliases** are always **case-insensitive** in SQL.

### Example on Linux:

```
SELECT * FROM Students;    --  if table is 'Students'
SELECT * FROM students;    --  error if table is 'Students'
```

### Best Practice

- Stick to **consistent lowercase naming** (snake\_case)
- Avoid relying on case to distinguish tables

## Data (Text Comparison)

Case sensitivity in data depends on the **collation** of the column.

### ✖ Collation Examples

Collation	Case-Sensitive?
<code>utf8_general_ci</code>	✗ No (CI = Case Insensitive)
<code>utf8_bin</code>	✓ Yes (Binary)
<code>utf8mb4_unicode_ci</code>	✗ No

### 🔍 Example

-- Case-insensitive

```
SELECT * FROM users WHERE name = 'Alice'; -- Matches 'alice', 'ALICE'
```

-- Case-sensitive (if collation is utf8\_bin)

```
SELECT * FROM users WHERE BINARY name = 'Alice';
```

You can also **force case-sensitive** comparison using the BINARY keyword.

## Variables, Triggers, Stored Procedures

- **User-defined variables** are **case-sensitive**

Example: `@MyVar ≠ @myvar`

- **Stored procedures, triggers, events:**

- **Names are case-sensitive on Unix/Linux**
- **Case-insensitive on Windows**

### ✓ Tips and Best Practices

Tip	Reason
<b>Use lowercase consistently</b>	Avoid OS-dependent issues
<b>Define column collations explicitly</b>	Control text comparison behavior
<b>Use BINARY if exact match needed</b>	Enforces case sensitivity in queries
<b>Avoid using case to differentiate tables</b>	Confusing and non-portable design

# MySQL Architecture Deep Dive

Understanding the internal architecture of MySQL helps optimize performance, troubleshoot issues, and write efficient queries. This section explores the core components and flow of MySQL from the moment a query is received to when it's executed.

## ❖ Client-Server Model

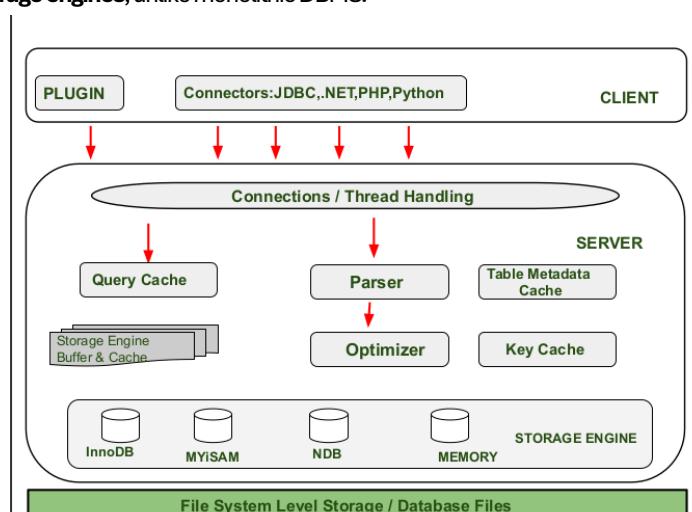
MySQL uses a **client-server architecture**, where:

- **Client:** Sends requests (SQL commands) to the server. Can be CLI tools, applications, Workbench, or scripts.
- **Server (mysqld):** The core MySQL daemon that manages databases, processes SQL, and returns results.

## Components:

1. **Clients:** Connect via network or local sockets.
2. **Connection Layer:**
  - Handles authentication
  - Manages session state and privileges
3. **SQL Layer:**
  - Parses and optimizes queries
  - Caches query results
4. **Storage Engine Layer:**
  - Manages how data is stored/retrieved
  - Interacts with physical data on disk

❖ MySQL supports **pluggable storage engines**, unlike monolithic DBMS.



Architecture of MySQL describes the relation among the different components of MySQL System. MySQL follow Client-Server Architecture. It is designed so that end user that is Clients can access the resources from Computer that is server using various networking services. The Architecture of MY SQL contain following major layer's :

- Client
- Server
- Storage Layer

## Client Layer

This layer is the topmost layer in the above diagram. The Client give request instructions to the Serve with the help of Client Layer .The Client make request through Command Prompt or through GUI screen by using valid MySQL commands and expressions .If the Expressions and commands are valid then the output is obtained on the screen. Some important services of client layer are :

- Connection Handling.
- Authentication.
- Security.

## Connection Handling :

When a client send request to the server and server will accept the request and the client is connected .. When Client is connected to

the server at that time , a client get it's own thread for it's connection. With the help of this thread all the queries from client side is executed.

#### **Authentication :**

Authentication is performed on the server side when client is connected to the MySQL server. Authentication is done with the help of username and password.

#### **Security :**

After authentication when the client gets connected successfully to MySQL server, the server will check that a particular client has the privileges to issue in certain queries against MySQL server.

## **Server Layer**

The second layer of MySQL architecture is responsible for all logical functionalities of relational database management system of MySQL. This Layer of MySQL System is also known as "**Brain of MySQL Architecture**". When the Client give request instructions to the Server and the server gives the output as soon as the instruction is matched. The various subcomponents of MySQL server are:

- **Thread Handling -**

When a client send request to the server and server will accept the request and the client is connected .. When Client is connected to the server at that time , a client get it's own thread for it's connection. This thread is provided by thread handling of Server Layer. Also the queries of client side which is executed by the thread is also handled by Thread Handling module.

- **Parser -**

A Parser is a type of Software Component that built a data structure(parse tree) of given input . Before parsing lexical analysis is done i.e. input is broken into number of tokens . After the data is available in the smaller elements parser perform Syntax Analysis , Semantics Analysis after that parse tree is generated as output.

- **Optimizer -**

As soon as the parsing is done , various types of optimization techniques are applied at Optimizer Block. These techniques may include rewriting the query, order of scanning of tables and choosing the right indexes to use etc.

- **Query Cache -**

Query Cache stores the complete result set for inputted query statement. Even before Parsing , MySQL Server consult query cache . When client write a query , if the query written by client is identical in the cache then the server simply skip the parsing, optimization and even execution, it just simply display the output from the cache.

- **Buffer and Cache -**

Cache and will buffer store the previous query or problem asked by user. When User write a query then it firstly goes to Query Cache then query cache will check that the same query or problem is available in the cache. If the same query is available then it will provide output without interfering Parser, Optimizer.

- **Table Metadata Cache -**

The metadata cache is a reserved area of memory used for tracking information on databases, indexes, or objects. The greater the number of open databases, indexes, or objects, the larger the metadata cache size.

- **Key Cache -**

A key cache is an index entry that uniquely identifies an object in a cache. By default, edge servers cache content based on the entire resource path and a query string.

## **Storage Layer**

This Storage Engine Layer of MySQL Architecture make it's unique and most preferable for developer's. Due to this Layer, MySQL layer is counted as the mostly used RDBMS and is widely used. In MySQL server, for different situations and requirement's different types of storage engines are used which are InnoDB ,MyISAM , NDB ,Memory etc. These storage engines are used as pluggable storage engine where tables created by user are plugged with them.

#### **Features of MySQL:**

1. MySQL's language is easy to use as compared to other programming language like C, C++, Java etc. By learning with some basic command we can work , create and interact with Database.

2. MySQL consist of Data Security layer which protect the data from violator. Also, passwords are encrypted in MySQL.
3. MySQL follow Client-Server Architecture where Client request Commands and instructions and Server will produce output as soon as the instruction is matched.
4. MySQL is free to use under the Community version of it. So we can download it from MySQL website and work on it freely.
5. MySQL use multithreading, which makes it Scalable. It can handle any amount of data . The default file size limit is 4 GB, but we can increase it according to our need.
6. MySQL is considered as one of the fast databases. It's fastness is determined on the basis of large number of benchmark tests.
7. MySQL is very flexible because it supports large number of embedded systems.
8. MySQL is compatible to run on various operating system such as Windows, macOS , Linux etc.
9. MySQL allow transactions to be rolled back, commit and cash recovery.
10. It has low memory leakage problem which increase its memory efficiency.
11. MySQL version 8.0 provide dual password support , one is a current password and another is secondary password. With the help of this we can create new password.
12. MySQL provide feature of Partitioning which improve performance of large databases.

## Storage Engines (InnoDB vs MyISAM)

### ◊ InnoDB (Default):

- ACID compliant (supports transactions)
- Row-level locking
- Foreign key constraints
- Crash recovery via redo logs
- Uses a buffer pool for caching
- Stores tables and indexes in a shared tablespace (ibdata) or per-table files (.ibd)

### ◊ MyISAM:

- No transaction support
- Table-level locking
- Faster for read-heavy operations
- No foreign keys
- Corrupt-prone if crashes occur

Transactions	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> No
Locking	Row-level	Table-level
Foreign Keys	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> No
Crash Recovery	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> No
Default in MySQL 8.x	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> No

⌚ Recommendation: Use InnoDB for most production applications unless there's a niche use case for MyISAM.

## Query Execution Process

1. Client sends query
2. SELECT \* FROM users WHERE id = 1;
3. Parser:
  - Checks syntax
  - Builds a parse tree
4. Optimizer:
  - Determines best execution path
  - Chooses indexes, join orders
  - Rewrites queries if necessary
5. Query Cache (if enabled):
  - Returns cached result if identical query was run before
6. Execution Engine:
  - Interacts with storage engine
  - Retrieves or modifies data
7. Result Sent Back to Client

Use EXPLAIN to analyze query execution paths and performance.

### Buffer Pool, Caches, and Threads

#### Buffer Pool (InnoDB):

- Memory area used to **cache frequently accessed data and indexes**
- Reduces disk I/O, speeding up reads/writes
- Set via innodb\_buffer\_pool\_size

innodb\_buffer\_pool\_size = 1G

#### Key Caches (MyISAM):

- Uses key\_buffer\_size for index caching

#### Query Cache (Deprecated in MySQL 8.0):

- Cached full SELECT query results
- Now replaced by **application-level or proxy caching** (e.g., Redis, Memcached)

#### Threads:

- MySQL uses a **multi-threaded architecture**:
  - One thread per client connection
  - Background threads for flushing logs, cleaning up, managing I/O
- Thread pool can be configured for high concurrency

thread\_cache\_size = 100

max\_connections = 500

### Summary

- MySQL's modular architecture separates **query processing from storage**.
- Choose storage engines based on your use case.
- Understanding **buffer pools and query execution** is essential for tuning performance.

# Database Design Principles

A well-designed database ensures **data consistency, scalability, and efficiency**. This chapter focuses on the foundations of relational modeling, normalization, data relationships, and schema optimization through indexing.

## Understanding Relational Models

### What is a Relational Database?

A **Relational Database** organizes data into **tables (relations)**, where:

- Each **row** = record (tuple)
- Each **column** = attribute (field)
- Tables are related through **keys**

### Key Concepts:

- **Primary Key (PK)**: Uniquely identifies a row
- **Foreign Key (FK)**: Links rows between tables
- **Integrity Constraints**: Ensure data validity (e.g., NOT NULL, UNIQUE)

Example:

### Students Table

student_id	name	age
101	Alice	22

### Enrollments Table

student_id	course_code
101	CS101

## Normalization (1NF, 2NF, 3NF, BCNF)

### First Normal Form (1NF) – Explained with Example

#### Definition

A table is in **First Normal Form (1NF)** if:

1. **Each column contains atomic (indivisible) values**
2. **Each record is unique**
3. **No repeating groups or arrays in any column**

#### ✗ Non-1NF Table (Bad Example):

student_id	name	phone_numbers
1	Alice	1234567890, 0987654321
2	Bob	1122334455

- **Problem:** `phone_numbers` contains multiple values (not atomic)
- This violates 1NF

### 1NF-Compliant Table (Good Example):

student_id	name	phone_number
1	Alice	1234567890
1	Alice	0987654321
2	Bob	1122334455

- Each field now holds a **single, atomic value**
- Repeating group is **removed**
- The relation satisfies **First Normal Form**

### Why it matters:

- Ensures consistent data storage
- Easier to search, filter, and update
- Lays foundation for higher normal forms

## Second Normal Form (2NF) – Explained with Example

Definition

A table is in **Second Normal Form (2NF)** if:

1. It is already in **First Normal Form (1NF)**
2. **Every non-prime attribute is fully functionally dependent on the entire primary key**, not just part of it  
→ i.e., **no partial dependency**

### Key Terms:

- **Partial Dependency**: When a non-key column depends on only **part of a composite primary key**
- **Non-prime attribute**: A column that is **not part of any candidate key**

### Non-2NF Table (Bad Example)

Let's say we have a table for student course enrollments:

student_id	course_id	student_name	course_name
1	C1	Alice	Math
1	C2	Alice	Physics
2	C1	Bob	Math

- **Primary Key**: (student\_id, course\_id) (composite key)
  - student\_name depends only on student\_id ✓
  - course\_name depends only on course\_id ✓
- 👉 These are **partial dependencies** ⇒ ✗ Not in 2NF

## 2NF-Compliant Tables (After Decomposition)

### 1. Student Table:

student_id	student_name
1	Alice
2	Bob

### 2. Course Table:

course_id	course_name
C1	Math
C2	Physics

### 3. Enrollment Table:

student_id	course_id
1	C1
1	C2
2	C1

- Now each non-key attribute depends **fully on the primary key** of its own table
- All **partial dependencies are removed**
- The design satisfies **Second Normal Form (2NF)**

#### Why 2NF Matters:

- Eliminates redundant data
- Ensures better data integrity
- Prevents anomalies during insert, update, delete

## Third Normal Form (3NF) – Explained with Example

### 🔍 Definition

A table is in **Third Normal Form (3NF)** if:

- It is already in **Second Normal Form (2NF)**
- No transitive dependency** exists  
(i.e., **non-key attributes must depend only on the primary key**, not on other non-key attributes)

### ❖ Key Terms:

- Transitive Dependency:** When  $A \rightarrow B$  and  $B \rightarrow C$ , then  $A \rightarrow C$  is transitive
- In a table, if a non-key column depends on another non-key column, that's a transitive dependency

### ✗ Non-3NF Table (Bad Example)

student_id	student_name	dept_id	dept_name
1	Alice	D1	CS
2	Bob	D2	Physics

- student\_id is the **primary key**
- dept\_id is **dependent on student\_id**
- dept\_name is **dependent on dept\_id**, not directly on student\_id → **transitive dependency**

### 3NF-Compliant Tables (After Decomposition)

#### 1. Students Table:

student_id	student_name	dept_id
1	Alice	D1
2	Bob	D2

#### 2. Departments Table:

dept_id	dept_name
D1	CS
D2	Physics

Now:

- All non-key attributes depend **only on the key**
- **No transitive dependencies**
- Table is in **Third Normal Form (3NF)**

### Why 3NF Matters:

- Prevents **data duplication**
- Reduces **update anomalies**
- Ensures **better referential integrity**

## Relationships: One-to-One, One-to-Many, Many-to-Many

### **[1] One-to-One (1:1)**

- Each row in A maps to **only one** row in B

Example:

- A user has **one** profile.

```
CREATE TABLE user (
    id INT PRIMARY KEY,
    name VARCHAR(100)
);
```

```
CREATE TABLE user_profile (
    user_id INT PRIMARY KEY,
    bio TEXT,
    FOREIGN KEY (user_id) REFERENCES user(id)
);
```

### **[1] → [2] One-to-Many (1:N)**

- A row in A maps to **multiple** rows in B

Example:

- One teacher teaches many classes.

```
CREATE TABLE teacher (
    id INT PRIMARY KEY,
    name VARCHAR(100)
);
```

```
CREATE TABLE class (
    id INT PRIMARY KEY,
    subject VARCHAR(100),
    teacher_id INT,
    FOREIGN KEY (teacher_id) REFERENCES teacher(id)
);
```

## Many-to-Many (M:N)

- Rows in A relate to **many** rows in B and vice versa
- Requires **junction table**

Example:

- Students enrolled in multiple courses.

```
CREATE TABLE student (
    id INT PRIMARY KEY,
    name VARCHAR(100)
);

CREATE TABLE course (
    id INT PRIMARY KEY,
    title VARCHAR(100)
);

CREATE TABLE enrollment (
    student_id INT,
    course_id INT,
    PRIMARY KEY (student_id, course_id),
    FOREIGN KEY (student_id) REFERENCES student(id),
    FOREIGN KEY (course_id) REFERENCES course(id)
);
```

## Indexing Strategy and Schema Optimization

### Why Indexing Matters:

Indexes improve **query performance**, especially for large datasets.

### Types of Indexes:

- **Primary Index:** Auto-created on PK
- **Secondary Index:** Manually created on other fields
- **Composite Index:** Index on multiple columns
- **Full-text Index:** For searching large text fields
- **Unique Index:** Prevents duplicate values

```
CREATE INDEX idx_name ON users(name);
```

### Indexing Best Practices:

Use Case	Indexing Tip
WHERE clauses	Index columns used frequently
JOINS	Index foreign keys
ORDER BY / GROUP BY	Index sorted/grouped columns
Composite Index	Use left-most prefix strategy
Avoid Over-indexing	Each index slows down INSERT/UPDATE

### Schema Optimization Tips:

- Use correct **data types** (e.g., INT vs BIGINT, VARCHAR vs TEXT)
- Avoid storing calculated or redundant values
- Denormalize carefully for read-heavy workloads
- Use **EXPLAIN** to evaluate query plans
- Normalize first, optimize later as needed

# Data Types and Table Structures

## Numeric Data Types

Used for storing numbers—both integers and floating-point values.

### Integer Types:

Type	Size (Bytes)	Range (Signed)	Use Case
TINYINT	1	-128 to 127	Flags, small counts
SMALLINT	2	-32,768 to 32,767	Small ranges
MEDIUMINT	3	-8M to 8M	Moderate-range IDs
INT / INTEGER	4	-2B to 2B	Most general use
BIGINT	8	Very large numbers	Large counters, timestamps

 Use UNSIGNED to store only positive values and double the positive range.

### Floating-Point Types:

Type	Description	Use Case
FLOAT(M,D)	Approx. decimal numbers (4 bytes)	Scientific calculations
DOUBLE or DOUBLE PRECISION	Higher precision (8 bytes)	Financial apps (approximate)
DECIMAL(M,D) or NUMERIC	Exact fixed-point precision	Accounting, money storage

## String Data Types

Used for text, characters, and binary data.

### Character Strings:

Type	Description	Use Case
CHAR(n)	Fixed-length string (padded)	Codes, fixed-size values
VARCHAR(n)	Variable-length string	Names, emails, titles

- n = max number of characters (not bytes)
- VARCHAR is more space-efficient for varying lengths

### Text Types (for long text):

Type	Max Length	Use Case
TINYTEXT	255 bytes	Short notes, descriptions
TEXT	64 KB	Article body, comments
MEDIUMTEXT	16 MB	Large documents
LONGTEXT	4 GB	Books, logs

### Binary Strings:

Type	Description	Use Case
BINARY(n)	Fixed-length binary	Password hashes, flags
VARBINARY(n)	Variable binary	File paths, hex values
BLOB types	Binary large object	Images, files, encrypted data

## Date and Time Data Types

Used to store calendar dates, times, timestamps, and durations.

Type	Format Example	Description
DATE	2025-06-25	Stores only date
TIME	14:30:00	Stores only time
DATETIME	2025-06-25 14:30:00	Date and time (no timezone)
TIMESTAMP	2025-06-25 14:30:00	Auto-converts to UTC, useful for logs
YEAR	2025	Stores a 4-digit year

 Use CURRENT\_TIMESTAMP as a default value for automatic date tracking.

### Tips for Choosing the Right Data Type:

- Use the **smallest type** that can hold your data (saves space).
- Use DECIMAL instead of FLOAT for **financial calculations**.
- Avoid TEXT/BLOB unless necessary—they’re stored outside the main row (performance hit).
- Use VARCHAR over CHAR unless you need fixed-width formatting.

## Creating and Altering Tables in MySQL

### Creating Tables

In MySQL, the CREATE TABLE statement is used to define a new table with specified columns, data types, and constraints.

#### Basic Syntax:

```
CREATE TABLE table_name (
    column1 datatype [constraints],
    column2 datatype [constraints],
    ...
);
```

#### Example

```
CREATE TABLE students (
    student_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    age INT,
    email VARCHAR(255) UNIQUE,
    registered_on DATE DEFAULT CURRENT_DATE
);
```

#### Key Concepts:

- AUTO\_INCREMENT: Automatically generates sequential values (for primary keys)
- PRIMARY KEY: Uniquely identifies each row
- NOT NULL: Ensures the field cannot be empty
- UNIQUE: Enforces uniqueness
- DEFAULT: Provides default value when none is supplied

#### Example with Foreign Key:

```
CREATE TABLE enrollments (
    enrollment_id INT PRIMARY KEY AUTO_INCREMENT,
    student_id INT,
    course_id INT,
    FOREIGN KEY (student_id) REFERENCES students(student_id),
    FOREIGN KEY (course_id) REFERENCES courses(course_id)
);
```

 **Best Practice:** Always index foreign keys for better performance.

## Altering Tables

The ALTER TABLE statement allows you to **modify an existing table**.

 **Add a Column:**

```
ALTER TABLE students ADD COLUMN gender VARCHAR(10);
```

 **Drop a Column:**

```
ALTER TABLE students DROP COLUMN gender;
```

 **Modify a Column Type:**

```
ALTER TABLE students MODIFY COLUMN name VARCHAR(150);
```

 **Rename a Column:**

```
ALTER TABLE students CHANGE COLUMN name full_name VARCHAR(150);
```

 **Add a Primary Key or Unique Constraint:**

```
ALTER TABLE students ADD PRIMARY KEY (student_id);
ALTER TABLE students ADD UNIQUE (email);
```

 **Add a Foreign Key:**

```
ALTER TABLE enrollments
ADD CONSTRAINT fk_student
FOREIGN KEY (student_id) REFERENCES students(student_id);
```

 **Rename a Table:**

```
RENAME TABLE students TO university_students;
```

 **Best Practices:**

- Always **backup your data** before altering structure
- Use **descriptive column names**
- Avoid altering large tables during peak traffic times
- For large changes, consider creating a new table and migrating data

# Constraints in MySQL

Constraints are rules enforced on data columns to ensure **data integrity, accuracy, and reliability** in a relational database.

## PRIMARY KEY

A **PRIMARY KEY** uniquely identifies each record in a table.

### Properties

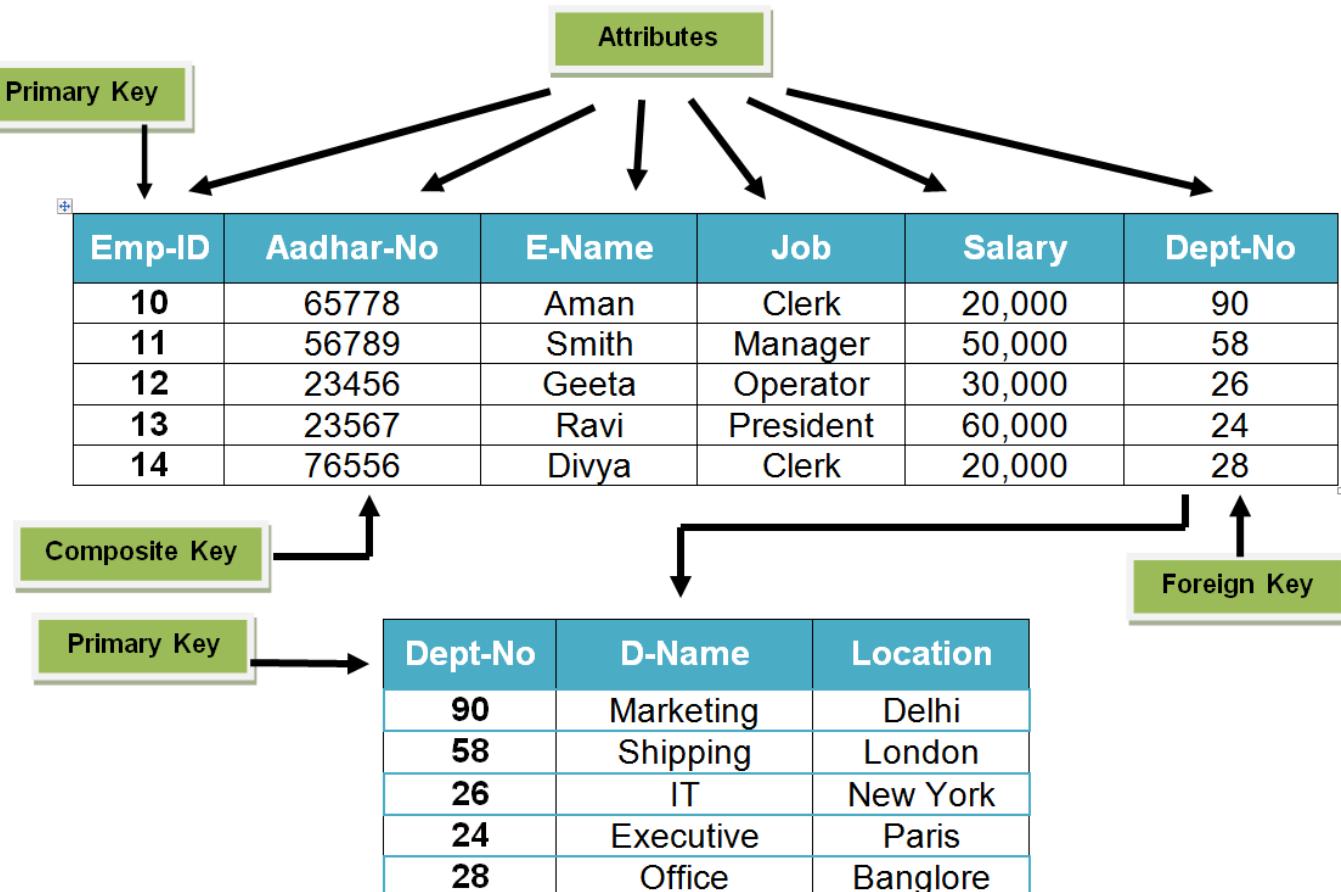
- Cannot contain NULL
- Must be **unique**
- Only one primary key per table (can be single or composite)

### Example (Single Column):

```
CREATE TABLE students (
    student_id INT PRIMARY KEY,
    name VARCHAR(100)
);
```

### Example (Composite Key – multiple primary keys):

```
CREATE TABLE enrollments (
    student_id INT,
    course_id INT,
    PRIMARY KEY (student_id, course_id)
);
```



## UNIQUE

The **UNIQUE** constraint ensures that **all values in a column are different**.

- Allows one NULL (depending on engine)
- Can be applied to multiple columns

◊ **Example:**

```
CREATE TABLE users (
    user_id INT PRIMARY KEY,
    username VARCHAR(100) UNIQUE,
    email VARCHAR(100) UNIQUE
);
```

💡 Use UNIQUE for columns like email, phone\_number, username, etc.

## 3. NOT NULL

The **NOT NULL** constraint enforces that a column **must contain a value** — it cannot be left empty.

◊ **Example**

```
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    salary DECIMAL(10,2) NOT NULL
);
```

- Prevents accidental omission of critical data

## FOREIGN KEY

A **FOREIGN KEY** is used to enforce a **relationship** between two tables.

It links a column in the child table to the **primary key** in the parent table.

Foreign key can be null.

◊ **Example**

```
CREATE TABLE departments (
    dept_id INT PRIMARY KEY,
    dept_name VARCHAR(100)
);

CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(100),
    dept_id INT,
    FOREIGN KEY (dept_id) REFERENCES departments(dept_id)
);
```

💡 With Actions

```
FOREIGN KEY (dept_id)
REFERENCES departments(dept_id)
ON DELETE CASCADE
ON UPDATE CASCADE
```

- ON DELETE CASCADE: Deletes child rows if parent is deleted
- ON UPDATE CASCADE: Updates child rows if parent key changes

## Summary Table

Constraint	Ensures...	Can be on multiple columns?
<b>PRIMARY KEY</b>	Uniqueness + not null	<input checked="" type="checkbox"/> Yes (composite)
<b>UNIQUE</b>	Uniqueness only	<input checked="" type="checkbox"/> Yes
<b>NOT NULL</b>	Value must be provided	<input checked="" type="checkbox"/> Yes
<b>FOREIGN KEY</b>	Referential integrity	<input checked="" type="checkbox"/> Yes

# DEFAULT, INDEX, and CANDIDATE KEY

## DEFAULT

The DEFAULT keyword sets a **predefined value** for a column when **no value is provided** during INSERT.

### Syntax:

```
CREATE TABLE students (
    student_id INT PRIMARY KEY,
    name VARCHAR(100),
    age INT DEFAULT 18,
    joined_date DATE DEFAULT CURRENT_DATE
);
```

#### ❖ Behavior:

```
INSERT INTO students (student_id, name) VALUES (1, 'Alice');
```

student_id	name	age	joined_date
1	Alice	18	2025-06-26

#### ❖ Use Cases

- Set a default status ('active')
- Default timestamps (CURRENT\_TIMESTAMP)
- Default numeric values (like 0 or 100)

## INDEX

An INDEX improves **query performance** by allowing MySQL to **quickly locate rows** without scanning the entire table.

### Syntax

```
CREATE INDEX idx_email ON students(email);
```

- You can also define it inside CREATE TABLE:

```
email VARCHAR(150), INDEX (email)
```

#### ❖ Types of Indexes:

Index Type	Description
PRIMARY KEY	Unique, not null, automatically indexed
UNIQUE	Prevents duplicates, automatically indexed
INDEX	Basic index for search optimization
FULLTEXT	Used for text search
SPATIAL	Used for GIS data

### Benefits:

- Faster WHERE, ORDER BY, and JOIN
- Speeds up searching on indexed columns

### Downsides:

- Slower INSERT/UPDATE (because index needs updating)
- Consumes extra storage

## CANDIDATE KEY

A **candidate key** is any column (or combination) that can uniquely identify a row.

Among candidate keys, one is chosen as the **primary key**.

### Example

```
CREATE TABLE users (
    user_id INT AUTO_INCREMENT,
    username VARCHAR(100) UNIQUE,
    email VARCHAR(150) UNIQUE,
    PRIMARY KEY (user_id)
);
```

#### ◊ In This Table

- user\_id, username, and email are **candidate keys**
- user\_id is selected as the **primary key**

#### ◊ Properties of Candidate Keys

- Must be **unique**
- Must be **not null**
- A table can have **multiple candidate keys**, but only **one primary key**

### Summary Table

Concept	Purpose	Example
DEFAULT	Assigns default value when none provided	age INT DEFAULT 18
INDEX	Improves search/query performance	CREATE INDEX idx ON (email)
CANDIDATE KEY	Uniquely identifies a row (potential PK)	username, email, user_id

# ENUMs, SETs, and JSON Fields in MySQL

These are **specialized data types** useful for enforcing controlled values (ENUM, SET) or storing structured data (JSON) in a single column.

## ENUM (Enumeration)

ENUM is a **string object** with a predefined list of **permissible values**. It stores **one value from the list**.

### ❖ Syntax:

```
CREATE TABLE users (
    status ENUM('active', 'inactive', 'banned') NOT NULL
);
```

### Example Insert

```
INSERT INTO users (status) VALUES ('active');
```

### Notes:

- Stored internally as **index values** (1 for 'active', 2 for 'inactive'...)
- Max 65,535 values (though practically you should use far fewer)
- Default sorting is **by index**, not alphabetically

### Use Cases:

- Status fields (active, pending, archived)
- Gender (male, female, other)
- Account type (free, premium, enterprise)

## SET

SET allows storing **multiple values from a predefined list** in a single column.

Values are stored as a **comma-separated string** but represented as **bitmasks** internally.

### ❖ Syntax:

```
CREATE TABLE products (
    features SET('wifi', 'gps', 'bluetooth', 'nfc')
);
```

### Example Insert:

```
INSERT INTO products (features) VALUES ('wifi,bluetooth');
```

### Notes:

- Max 64 members
- You can select **any combination** of the allowed options
- Ideal for fields where multiple options apply

### Use Cases:

- Product features
- User interests
- Tagging systems

## JSON

The JSON type stores **structured JSON-formatted data** directly in MySQL.

Supports full **querying and manipulation** of data inside JSON objects.

### ❖ Syntax:

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_info JSON
);
```

Before INSERT

order_id	customer_info
(empty)	(empty)

No data exists yet in the orders table.

#### Example Insert:

```
INSERT INTO orders (order_id, customer_info)
VALUES (1, '{"name": "Alice", "email": "alice@example.com"}');
```

After INSERT

order_id	customer_info
1	{"name": "Alice", "email": "alice@example.com"}

A new row has been inserted with a JSON object stored in the customer\_info field.

#### Querying JSON:

```
SELECT customer_info->>'$ .email' AS email
FROM orders
WHERE JSON_EXTRACT(customer_info, '$ .name') = 'Alice';
```

#### Use Cases:

- Flexible user profile fields
- Logs and events
- Storing form submissions or metadata

#### Summary Comparison

Type	Stores	Allows Multiple?	Use Case
ENUM	One predefined value	<input checked="" type="checkbox"/> No	Status, role, type fields
SET	Many predefined values	<input checked="" type="checkbox"/> Yes	Tags, features, interests
JSON	Structured key-value	<input checked="" type="checkbox"/> Yes (nested)	Metadata, dynamic fields, APIs

# CRUD Operations: INSERT (Create)

In SQL, **CRUD** stands for

Create – INSERT

Read – SELECT

Update – UPDATE

Delete – DELETE

## INSERT Statement

```
INSERT INTO table_name (column1, column2, ...)
VALUES (value1, value2, ...);
```

- You must match **column order** and **value types**
- Omit column list only if inserting into **all columns**

### Example Table

Column	Type	Description
student_id	INT (PK, AUTO_INCREMENT)	Unique ID
name	VARCHAR(100)	Student's full name
age	INT	Age
email	VARCHAR(150)	Must be unique
joined_date	DATE	Defaults to current date

```
CREATE TABLE students (
    student_id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(100),
    age INT,
    email VARCHAR(150) UNIQUE,
    joined_date DATE DEFAULT CURRENT_DATE
);
```

### 1. Insert All Columns (Explicit)

```
INSERT INTO students (name, age, email, joined_date)
VALUES ('Alice', 22, 'alice@example.com', '2025-06-25');
```

student_id	name	age	email	joined_date
1	Alice	22	alice@example.com	2025-06-25

### 2. Insert with Defaults / Auto-Increment

```
INSERT INTO students (name, age, email)
VALUES ('Bob', 20, 'bob@example.com');
```

- joined\_date will default to CURRENT\_DATE
- student\_id will auto-increment

student_id	name	age	email	joined_date
1	Alice	22	alice@example.com	2025-06-25
2	Bob	20	bob@example.com	2025-06-26

### 3. Insert Multiple Rows

```
INSERT INTO students (name, age, email)
```

## VALUES

```
('Charlie', 23, 'charlie@example.com'),  
('Dana', 21, 'dana@example.com');
```

student_id	name	age	email	joined_date
1	Alice	22	alice@example.com	2025-06-25
2	Bob	20	bob@example.com	2025-06-26
3	Charlie	23	charlie@example.com	2025-06-26
4	Dana	21	dana@example.com	2025-06-26

## 4. Insert from Another Table

Useful for data migration or transformation.

```
INSERT INTO students_archive (student_id, name, email)  
SELECT student_id, name, email FROM students WHERE age > 21;
```

## Common Errors & Fixes

Error	Cause	Fix
Duplicate entry	Violating UNIQUE or PRIMARY KEY	Ensure uniqueness
Column count doesn't match	Mismatch between columns and values	Match exactly
Incorrect date value	Bad date format	Use YYYY-MM-DD format
Cannot be NULL	Inserting NULL into a NOT NULL field	Provide a value or use default

## Best Practices

- Always **name columns** explicitly — avoids errors when table schema changes
- Use **parameterized queries** in applications to avoid SQL injection
- For bulk inserts, use **multiple rows** syntax to optimize performance

Here's a complete note on **CRUD Operations – SELECT (Read)** in MySQL, with syntax, usage, and tabulated examples:

## SELECT (Read)

The SELECT statement is used to **retrieve data** from one or more tables. It is the most commonly used command to **query** the database.

## Basic Syntax

```
SELECT column1, column2, ...  
FROM table_name  
WHERE condition  
ORDER BY column  
LIMIT n;
```

- SELECT \* retrieves **all columns**
- WHERE filters rows based on conditions
- ORDER BY sorts results
- LIMIT restricts the number of rows returned

## Example Table: students

student_id	name	age	email	joined_date
1	Alice	22	<a href="mailto:alice@example.com">alice@example.com</a>	2025-06-25
2	Bob	20	<a href="mailto:bob@example.com">bob@example.com</a>	2025-06-26
3	Charlie	23	<a href="mailto:charlie@example.com">charlie@example.com</a>	2025-06-26

### 1. Select All Columns

```
SELECT * FROM students;
```

student_id	name	age	email	joined_date
1	Alice	22	<a href="mailto:alice@example.com">alice@example.com</a>	2025-06-25
2	Bob	20	<a href="mailto:bob@example.com">bob@example.com</a>	2025-06-26
3	Charlie	23	<a href="mailto:charlie@example.com">charlie@example.com</a>	2025-06-26

### 2. Select Specific Columns

```
SELECT name, email FROM students;
```

name	email
Alice	<a href="mailto:alice@example.com">alice@example.com</a>
Bob	<a href="mailto:bob@example.com">bob@example.com</a>
Charlie	<a href="mailto:charlie@example.com">charlie@example.com</a>

### 3. Filtering Rows with WHERE

```
SELECT * FROM students WHERE age > 21;
```

student_id	name	age	email	joined_date
1	Alice	22	<a href="mailto:alice@example.com">alice@example.com</a>	2025-06-25
3	Charlie	23	<a href="mailto:charlie@example.com">charlie@example.com</a>	2025-06-26

### 4. Sorting with ORDER BY

```
SELECT * FROM students ORDER BY age DESC;
```

student_id	name	age	email	joined_date
3	Charlie	23	<a href="mailto:charlie@example.com">charlie@example.com</a>	2025-06-26
1	Alice	22	<a href="mailto:alice@example.com">alice@example.com</a>	2025-06-25
2	Bob	20	<a href="mailto:bob@example.com">bob@example.com</a>	2025-06-26

### 5. Limiting Results with LIMIT

```
SELECT * FROM students ORDER BY student_id LIMIT 2;
```

student_id	name	age	email	joined_date
1	Alice	22	<a href="mailto:alice@example.com">alice@example.com</a>	2025-06-25
2	Bob	20	<a href="mailto:bob@example.com">bob@example.com</a>	2025-06-26

## 6. Using LIKE for Pattern Matching

```
SELECT * FROM students WHERE email LIKE '%example.com';
```

student_id	name	age	email	joined_date
1	Alice	22	alice@example.com	2025-06-25
2	Bob	20	bob@example.com	2025-06-26
3	Charlie	23	charlie@example.com	2025-06-26

## 7. Aggregate Functions

- Count number of students older than 21:

```
SELECT COUNT(*) AS count_over_21 FROM students WHERE age > 21;
```

count_over_21
2

## UPDATE

The UPDATE statement is used to **modify existing records** in a table.

It is crucial to use WHERE to target specific rows—otherwise, **all rows will be updated**.

### Basic Syntax

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

### Example Table: students

student_id	name	age	email	joined_date
1	Alice	22	alice@example.com	2025-06-25
2	Bob	20	bob@example.com	2025-06-26
3	Charlie	23	charlie@example.com	2025-06-26

### 1. Update a Single Row

```
UPDATE students
SET age = 24
WHERE student_id = 3;
```

student_id	name	age	email	joined_date
3	Charlie	24	charlie@example.com	2025-06-26

### 2. Update Multiple Columns

```
UPDATE students
SET name = 'Robert', age = 21
WHERE student_id = 2;
```

student_id	name	age	email
2	Robert	21	bob@example.com

### 3. Update Multiple Rows with a Condition

```
UPDATE students
SET joined_date = '2025-07-01'
WHERE age > 22;
```

student_id	name	age	joined_date
3	Charlie	24	2025-07-01

### 4. Danger of Missing WHERE Clause

```
UPDATE students SET age = 30;
```

 This will update **age** for all rows in the table!

### 5. Use Expressions in Updates

```
UPDATE students
SET age = age + 1
WHERE name = 'Alice';
Increases Alice's age by 1.
```

### Best Practices:

Practice	Why It Matters
Always use WHERE clause	Avoid unintentional full-table updates
Test with SELECT first	Preview which rows will be affected
Use LIMIT when debugging	Prevents massive changes in test updates
Backup before bulk updates	Data integrity and recovery

## DELETE

The DELETE statement is used to **remove rows** from a table permanently.

**Be careful:** Once deleted, data **cannot be recovered** unless you have a backup or use transactions.

### Basic Syntax

```
DELETE FROM table_name
```

WHERE condition;

- WHERE filters **which rows** to delete
- Without WHERE, **all rows** will be deleted

### Example Table: students

student_id	name	age	email	joined_date
1	Alice	23	alice@example.com	2025-06-25
2	Robert	21	bob@example.com	2025-06-26
3	Charlie	24	charlie@example.com	2025-07-01

### 1. Delete a Specific Row

```
DELETE FROM students
WHERE student_id = 2;
```

student_id	name
2	Robert

## 2. Delete Multiple Rows with a Condition

```
DELETE FROM students
WHERE age > 23;
```

student_id	name
3	Charlie

## 3. Delete All Rows (Dangerous)

```
DELETE FROM students;
```

- Removes **all data** from the table
- Table **structure remains intact**

## 4. Delete All Rows + Reset Auto-Increment

```
TRUNCATE TABLE students;
```

- Faster than DELETE
- **Resets auto-increment counter**
- Cannot be rolled back in most cases

## 5. Delete Using JOIN

Delete students who are not enrolled in any course:

```
DELETE s
FROM students s
LEFT JOIN enrollments e ON s.student_id = e.student_id
WHERE e.student_id IS NULL;
```

## Best Practices:

Tip	Why It's Important
<b>Always use a WHERE clause</b>	Prevents accidental full deletion
<b>Run a SELECT first</b>	See which rows will be affected
<b>Use LIMIT for cautious testing</b>	Useful for batch deletions
<b>Backup before deleting large data</b>	Recovery in case of mistake
<b>Use TRUNCATE only when needed</b>	Resets table fast, but cannot rollback

## Safe Deletion Techniques and Advanced Use Cases

### Soft Delete (Recommended for Recoverable Data)

Instead of deleting rows permanently, **mark them as deleted** using a column like `is_deleted` or `status`.

#### Schema Example:

```
ALTER TABLE students ADD COLUMN is_deleted BOOLEAN DEFAULT FALSE;
```

#### Soft Delete (Update Instead of Delete):

```
UPDATE students SET is_deleted = TRUE WHERE student_id = 2;
```

#### Query Active Records Only:

```
SELECT * FROM students WHERE is_deleted = FALSE;
```

Soft deletes allow:

- Data recovery
- Auditability
- Better referential integrity

## ⌚ 2. DELETE with FOREIGN KEY Constraints

If a child table references a parent via a foreign key, you may get an error unless the constraint specifies what to do on delete.

### ❖ Example Setup:

```
CREATE TABLE enrollments (
    id INT AUTO_INCREMENT PRIMARY KEY,
    student_id INT,
    FOREIGN KEY (student_id) REFERENCES students(student_id)
        ON DELETE CASCADE
);
```

- ON DELETE CASCADE: Deletes related enrollments when a student is deleted.
- ON DELETE SET NULL: Sets the student\_id in the child table to NULL.

## ❖ 3. DELETE Within a Transaction (Safe Delete)

Use transactions to allow **rollback** in case something goes wrong

```
START TRANSACTION;

DELETE FROM students WHERE student_id = 3;
```

-- Optional: Rollback if needed

-- ROLLBACK;

-- Or finalize changes

COMMIT;

## ⌚ 4. DELETE with LIMIT for Batch Removal

Useful when working with millions of rows:

```
DELETE FROM logs WHERE created_at < '2024-01-01' LIMIT 1000;
```

⌚ Run in a loop to avoid locking the entire table.

## 🔍 5. DELETE Based on Subqueries

Delete students **not enrolled** in any course:

```
DELETE FROM students
WHERE student_id NOT IN (SELECT DISTINCT student_id FROM enrollments);
```

## ⌚ DELETE vs TRUNCATE vs DROP

Command	Deletes Data	Resets Auto-ID	Drops Table Structure	Rollback Possible
DELETE	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> No	<input checked="" type="checkbox"/> No	<input checked="" type="checkbox"/> Yes
TRUNCATE	<input checked="" type="checkbox"/> Yes (fast)	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> No	<input checked="" type="checkbox"/> No (usually)
DROP	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> Yes (removes table)	<input checked="" type="checkbox"/> No

# Filtering and Sorting with WHERE and ORDER BY

When using SELECT to retrieve data, you often want to:

- **Filter rows** using WHERE
- **Sort results** using ORDER BY

## WHERE Clause – Filtering Rows

The WHERE clause filters rows that meet **specific conditions**.

❖ **Basic Syntax:**

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

**Examples**

a. **Filter by Equality:**

```
SELECT * FROM students WHERE age = 22;
```

b. **Filter by Comparison:**

```
SELECT * FROM students WHERE age > 21;
```

c. **Filter with AND/OR:**

```
SELECT * FROM students
WHERE age > 21 AND joined_date = '2025-06-25';
```

d. **Filter with IN:**

```
SELECT * FROM students WHERE name IN ('Alice', 'Charlie');
```

e. **Filter with BETWEEN:**

```
SELECT * FROM students WHERE age BETWEEN 20 AND 22;
```

f. **Filter with LIKE (Pattern Match):**

```
SELECT * FROM students WHERE email LIKE '%@example.com';
```

% = wildcard for any number of characters

\_ = wildcard for a single character

## ORDER BY Clause – Sorting Results

The ORDER BY clause **sorts the output rows** based on one or more columns.

❖ **Basic Syntax:**

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column_name [ASC | DESC];
```

- ASC = Ascending (default)
- DESC = Descending

## Examples

### a. Sort by Age (Ascending):

```
SELECT * FROM students ORDER BY age;
```

### b. Sort by Name (Descending):

```
SELECT * FROM students ORDER BY name DESC;
```

### c. Sort by Multiple Columns:

```
SELECT * FROM students ORDER BY age DESC, name ASC;
```

## 3. Combine WHERE + ORDER BY

You can use both together to filter and then sort the results.

```
SELECT name, age FROM students  
WHERE age > 20  
ORDER BY age DESC;
```

## Common Use Cases Table

Query Type	Example Code
Get all students over 21	SELECT * FROM students WHERE age > 21;
Students joined after a date	WHERE joined_date > '2025-06-01'
Sort by email alphabetically	ORDER BY email ASC
Top 5 youngest students	ORDER BY age ASC LIMIT 5

# Using SQLite with Python

## What is SQLite?

- **SQLite** is a **lightweight, serverless**, self-contained SQL database engine.
- It stores the entire database in a **single .db file**.
- Ideal for **small to medium applications, testing, and local data storage**.

## Installing SQLite

### No separate installation needed:

Python comes with sqlite3 module built-in (since Python 2.5+).

```
import sqlite3
```

## Connecting to a Database

```
import sqlite3

# Connects to database file (creates if it doesn't exist)
conn = sqlite3.connect("mydata.db")

# Create a cursor object to execute SQL
cursor = conn.cursor()
```

## Creating a Table

```
cursor.execute("""
CREATE TABLE IF NOT EXISTS students (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    age INTEGER,
    email TEXT UNIQUE
)
""")
conn.commit()
```

## Inserting Data

```
cursor.execute("""
INSERT INTO students (name, age, email)
VALUES (?, ?, ?)
""", ("Alice", 22, "alice@example.com"))
conn.commit()
```

Use ? placeholders to prevent SQL injection.

## Querying Data (SELECT)

```
cursor.execute("SELECT * FROM students")
rows = cursor.fetchall()

for row in rows:
    print(row)

• Use .fetchall() for all rows

• Use .fetchone() for a single row
```

## Updating Data

```
cursor.execute("""
UPDATE students
SET age = ?
WHERE name = ?
""", (23, "Alice"))
conn.commit()
```

## Deleting Data

```
cursor.execute("DELETE FROM students WHERE name = ?", ("Alice",))
conn.commit()
```

## Using with Context Manager (with)

Automatically handles opening and closing:

```
with sqlite3.connect("mydata.db") as conn:
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM students")
    print(cursor.fetchall())
```

## Handling Exceptions

```
try:
    conn = sqlite3.connect("mydata.db")
    cursor = conn.cursor()
    # some operation
except sqlite3.Error as e:
    print("SQLite error:", e)
finally:
    conn.close()
```

## SQLite Features Summary

Feature	Description
<b>Serverless</b>	No need to install or run a DB server
<b>Lightweight</b>	Entire DB in a single file
<b>ACID compliant</b>	Supports transactions
<b>Zero configuration</b>	Runs out of the box
<b>Supported in Python</b>	Built-in via sqlite3 module

## Best Practices

- Use ? placeholders to prevent SQL injection.
- Always commit() after modifying data.
- Wrap your DB logic in try-except blocks.
- Use with for automatic resource management.

# Big Data Engineering

## Foundations of Big Data

### What is Big Data?

**Big Data** refers to **extremely large and complex datasets** that are difficult to store, process, and analyze using traditional data processing tools (like relational databases and spreadsheets).

These datasets come from:

- Social media (Facebook posts, YouTube videos)
- Sensors and IoT devices (temperature readings, GPS data)



### 5 Vs of Big Data

Big Data is commonly described using **5 Vs**. Each "V" describes a key characteristic

#### 1. Volume

- Refers to the **amount of data**.
- Companies collect **terabytes or petabytes** of data.
- *Example: Facebook stores over 300 petabytes of user data.*

#### 2. Velocity

- The **speed at which data is generated and processed**.
- Real-time systems must handle data immediately as it arrives.
- *Example: Stock trading platforms process millions of transactions per second.*

#### 3. Variety

- Data comes in **many different formats**:
  - Structured (e.g., tables)
  - Semi-structured (e.g., JSON, XML)
  - Unstructured (e.g., images, videos, text)
- *Example: A car's IoT system sends GPS (structured), camera footage (unstructured), and alerts (semi-structured).*

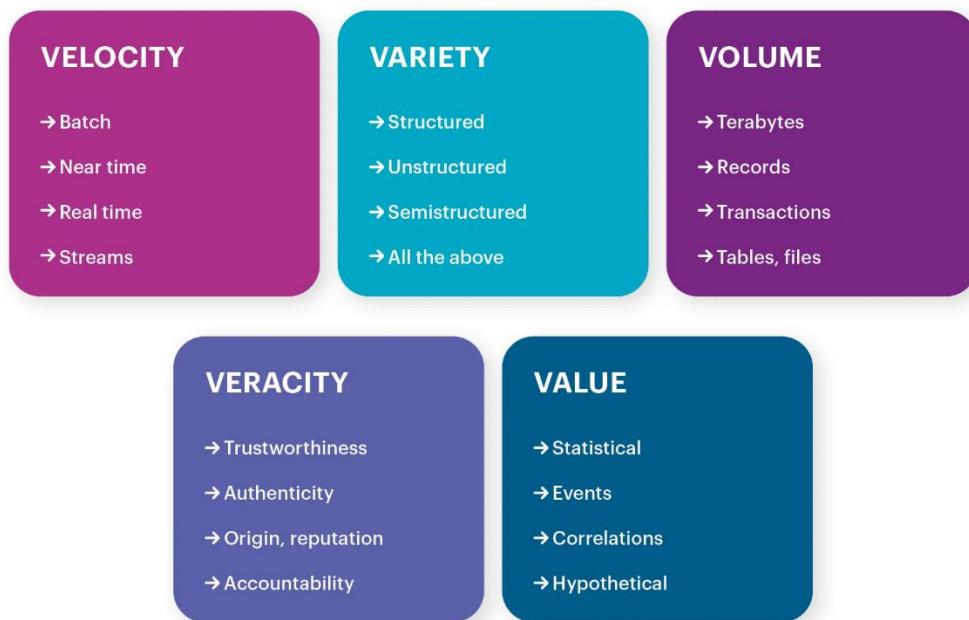
#### 4. Veracity

- Refers to the **quality and accuracy** of the data.
- Data might be incomplete, inconsistent, or noisy.
- *Example: User-entered data may contain spelling errors or false information.*

#### 5. Value

- It's not just about storing data—it's about **extracting useful insights**.
- *Example: Analyzing customer behavior data to increase sales.*

## The 5 Vs of Big Data



## Big Data vs Traditional Data

Aspect	Traditional Data	Big Data
<b>Volume</b>	Megabytes to Gigabytes	Terabytes to Petabytes and beyond
<b>Processing</b>	Batch processing (slow)	Real-time or near real-time
<b>Storage</b>	Centralized (SQL Databases)	Distributed (HDFS, S3, etc.)
<b>Format</b>	Structured	Structured, Semi-Structured, Unstructured
<b>Tools</b>	MySQL, Excel	Hadoop, Spark, Kafka, etc.
<b>Scalability</b>	Vertical (upgrade hardware)	Horizontal (add more machines)

## **Characteristics of Big Data Systems**

Big Data systems are designed to **handle massive, fast, and diverse datasets**. Key characteristics include:

### **1. Distributed Storage**

- Data is split across many machines for storage (like pieces of a puzzle).
- Tools: HDFS, Amazon S3

### **2. Parallel Processing**

- Data is processed in parallel by multiple systems, making it faster.
- Tools: Apache Spark, Hadoop MapReduce

### **3. Fault Tolerance**

- If one system fails, another takes over automatically.
- Example: Hadoop replicates data across machines.

### **4. Scalability**

- Systems can handle more data just by adding more servers (horizontal scaling).

### **5. High Throughput & Low Latency**

- Throughput: How much data can be processed per second.
- Latency: Delay between request and response.
- Big Data systems aim for **high throughput and low latency**, especially for real-time systems like fraud detection.

## **Data Lifecycle & Pipeline Overview**

Big Data goes through a **lifecycle** called a **data pipeline**. This defines how data moves from **creation to consumption**

### **1. Data Generation**

- From apps, sensors, websites, transactions, etc.

### **2. Data Ingestion**

- Getting the data into your system.
- Tools: Apache Kafka, Apache NiFi, Flume

### **3. Data Storage**

- Store in distributed file systems.
- Tools: HDFS, Amazon S3, Azure Blob

#### 4. Data Processing

- Clean, transform, and analyze.
- Batch tools: Apache Spark, Hive
- Stream tools: Flink, Storm, Spark Streaming

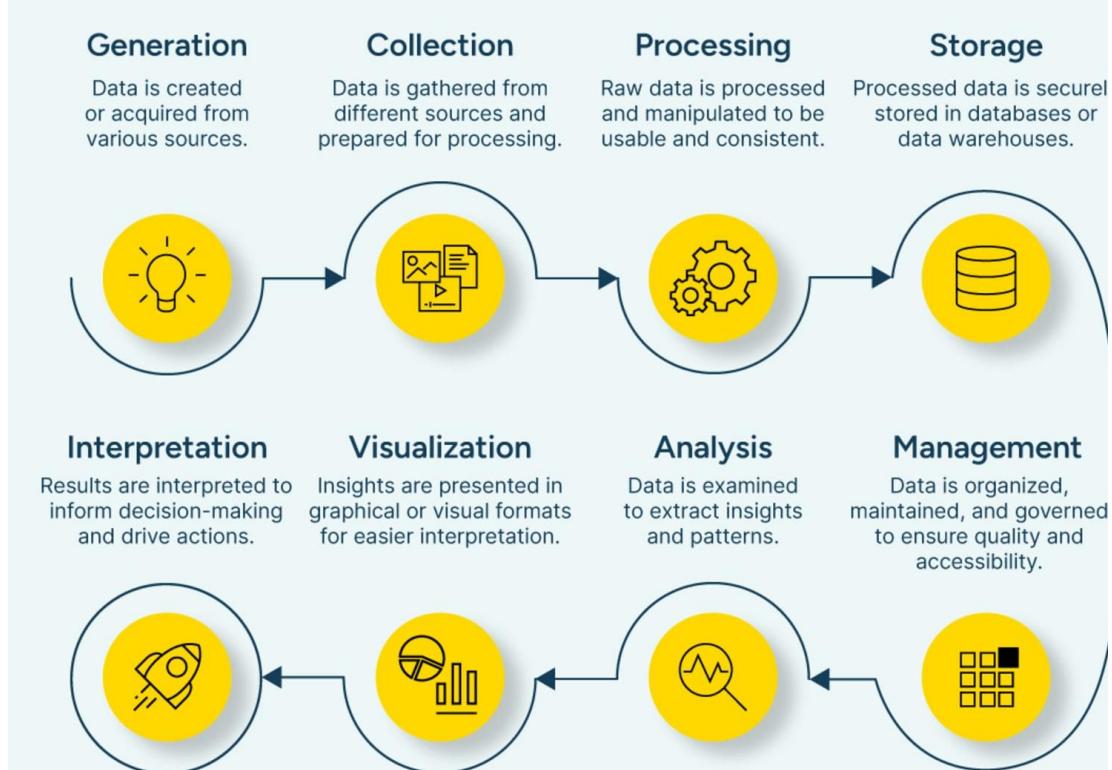
#### 5. Data Analysis & Visualization

- Analyze and present results.
- Tools: Tableau, Power BI, Superset

#### 6. Data Archival or Deletion

- Old data is either stored in cold storage or deleted as per policies.

## The Data Lifecycle



# Data Storage Systems

Big Data needs specialized systems to store massive volumes of diverse and fast-arriving data. These storage systems are **distributed, scalable, and optimized for big data analytics**.

## File Systems

### HDFS (Hadoop Distributed File System)

- A core component of the **Hadoop ecosystem**.
- Stores **large files across multiple machines**.
- Automatically splits files into blocks (default: 128MB or 256MB).
- **Redundancy**: Each block is replicated (default 3 copies) to handle failures.
- Optimized for **batch processing**, not real-time.

### GFS (Google File System)

- Developed by Google to handle **web-scale data** (like search index data).
- Inspired HDFS.
- Built for **fault tolerance, scalability, and high throughput**.
- Master-slave architecture:
  - **Master** keeps track of file metadata.
  - **Chunk servers** store the actual data in fixed-size chunks (64MB).

### Amazon S3 (Simple Storage Service)

- Cloud-based **object storage service** by AWS.
- Stores data as **objects in buckets**.
- Scalable, durable (99.99999999%), and available globally.
- Used widely in Big Data pipelines (input, output, backups).

### Azure Blob Storage

- Object storage from Microsoft Azure.
- "Blob" stands for **Binary Large Object**—great for storing unstructured data like images, videos, backups.
- Supports **hot, cool, and archive tiers** based on access frequency.

## Data Formats

Choosing the right format helps with **storage efficiency**, **query performance**, and **interoperability**.

### CSV (Comma-Separated Values)

- Simple text format: values separated by commas.
- Human-readable and easy to use.
- Not efficient for large or nested data.

#### Example:

```
name,age,city
Alice,30,New York
```

### JSON (JavaScript Object Notation)

- **Semi-structured** data format.
- Human-readable, supports nesting and hierarchies.
- Widely used in web APIs and NoSQL databases.

#### Example

```
{
  "name": "Alice",
  "age": 30,
  "skills": [ "Python", "SQL" ]
}
```

### XML (eXtensible Markup Language)

- Also semi-structured.
- Verbose, used in enterprise and legacy systems.
- Tag-based, supports hierarchy.

#### Example

```
<user>
  <name>Alice</name>
  <age>30</age>
</user>
```

### Parquet

- Columnar format, optimized for **analytical queries**.
- Great for large-scale data processing (works well with Apache Spark, Hive).
- **Compresses better** than row formats like CSV.

**Example:** If you query only the “age” column from a Parquet file, it skips the rest and loads just the age data.

## Avro

- Row-based, **binary format** used in Big Data.
- Supports schema evolution (i.e., schema changes over time).
- Smaller file size than JSON/XML.

**Example:** Used with Kafka to serialize messages efficiently.

## ORC (Optimized Row Columnar)

- Like Parquet but designed specifically for Hive.
- Very efficient for **read-heavy workloads**.
- Supports indexing, compression, and schema evolution.

**Use case:** Data warehouse queries on millions of rows in Hive.

## Protocol Buffers (Protobuf)

- Created by Google.
- Compact, **fast, binary serialization format**.
- Requires schema (defined in .proto files).
- Not human-readable, but extremely space-efficient.

**Example:** Used in systems where speed and size matter, like gRPC services.

## Data Lakes vs Data Warehouses

### Data Lake

- Stores **raw data** in its original format (structured + unstructured).
- Very **scalable and cheap** (e.g., S3, HDFS).
- Used for **exploratory analytics, machine learning**, and long-term storage.

**Example:** All raw logs, images, CSVs, and JSONs from a mobile app stored in one place.

### Data Warehouse

- Stores **processed and structured data** optimized for **analytical queries**.
- Focus on speed, performance, and reporting.
- Examples: Amazon Redshift, Google BigQuery, Snowflake.

**Use case:** Business Intelligence dashboards and KPIs.

### Key Differences:

Feature	Data Lake	Data Warehouse
<b>Data Type</b>	Raw, structured/unstructured	Structured, curated
<b>Cost</b>	Cheaper	More expensive
<b>Use Case</b>	ML, data exploration	Business reporting, analytics
<b>Storage Format</b>	Flexible (CSV, JSON, video)	Tabular (columns/tables)
<b>Performance</b>	Slower (for raw data)	Faster (optimized schema)

### ◆ Lakehouse Architecture (Hybrid)

### What is a Lakehouse?

A **Lakehouse** combines the **scalability of Data Lakes** with the **structure and performance of Data Warehouses**.

### Key Components:

- **Delta Lake** (Databricks): Brings ACID transactions to data lakes.
- **Apache Iceberg**: Supports large tables, schema evolution, and partitioning.
- **Apache Hudi**: Designed for real-time updates on large datasets.

**Use case:** An e-commerce platform storing all customer behavior logs in S3 (lake) and analyzing structured tables in Spark SQL (warehouse-style queries).

## Database vs Data Warehouse vs Data Lake

Feature	Database	Data Warehouse	Data Lake
Purpose	Store <b>real-time</b> operational data	Store structured data for <b>historical analysis</b> /reporting	Store all types of <b>raw data</b> (structured + unstructured)
Speed	High speed	Optimized for analytical queries	Variable
Data Type	Structured	Structured	Structured, Semi-structured, Unstructured
Data Format	Tables, rows, columns	Tables (optimized for queries)	Files (CSV, JSON, images, videos, logs)
Processing	OLTP (Online Transaction Processing)	OLAP (Online Analytical Processing)	ELT (Extract-Load-Transform) or raw storage
Users	App developers, transactional systems	Business analysts, data scientists	Data engineers, data scientists
Examples	MySQL, PostgreSQL, Oracle	Snowflake, Redshift, BigQuery, Teradata	Amazon S3 + Glue, Azure Data Lake, HDFS
Query Speed	Fast for small operational queries	Fast for large analytical queries	Slower unless optimized or pre-processed
Schema	Schema-on-write (predefined schema)	Schema-on-write	Schema-on-read (flexible)
Cost	Moderate	High (for compute + storage)	Low (cheap storage, high scalability)
Scalability	Limited	Moderate	Highly Scalable

### 1. Database

#### Definition:

A database is a structured system to store and manage **real-time operational data**, like customer records, orders, or banking transactions.

- Supports **CRUD operations**: Create, Read, Update, Delete.
- Optimized for fast reads/writes with transactional integrity (ACID properties).
- **Schema is fixed** and predefined.

#### Example Use Case:

A banking system uses PostgreSQL to track account balances and transactions.

## 2. Data Warehouse

### Definition:

A data warehouse is a centralized repository for **structured, processed data**, optimized for **complex queries and analysis**.

- Handles **OLAP** (analytical) operations.
- Used for **reporting, dashboards**, and business intelligence (BI).
- Ingests data from many sources via ETL processes (Extract-Transform-Load).

### Example Use Case:

A retailer stores 10 years of sales data in Snowflake and runs queries to forecast inventory trends.

## 3. Data Lake

### Definition:

A data lake is a **scalable storage system** that holds **raw data** in any format (structured, semi-structured, unstructured).

- Data is stored in its original form until needed.
- Uses **schema-on-read**: schema applied only when reading the data.
- Good for **machine learning, AI, and exploratory analytics**.

### Example Use Case:

A healthcare provider stores patient X-rays, doctor notes (PDF), and device logs in AWS S3 and later analyzes them using Apache Spark.

### Quick Analogy

System	Analogy
Database	A real-time diary you update daily.
Data Warehouse	A well-organized report book with summaries.
Data Lake	A messy box with receipts, photos, papers—you organize it only when needed.

### Real-World Integration Example

Let's say you're building an **e-commerce system**:

- **Database** (PostgreSQL): Stores user profiles, order history, payment details.
- **Data Warehouse** (Redshift): Stores monthly sales reports, profit analysis, top-selling products.
- **Data Lake** (S3): Stores clickstream logs, raw product images, chat logs, and sensor data from delivery vehicles.

# Distributed Computing & Frameworks

Big Data is too large to be processed on a single machine, so we use **distributed computing**, which breaks a job into smaller tasks and processes them across multiple machines in parallel. Frameworks like Hadoop and Spark help manage and scale this easily.

## Hadoop Ecosystem

### 1. HDFS (Hadoop Distributed File System)

- A distributed file system that stores data across many nodes.
- Handles **large files by breaking them into blocks** and replicating them for fault tolerance.

**Example:** A 1TB log file is split into 256MB blocks and stored across 20 machines with 3 copies of each block.

### 2. MapReduce

- A programming model to process big data in two stages:
  - **Map:** Break the data into key-value pairs.
  - **Reduce:** Aggregate or summarize the mapped data.

**Example:** Word Count:

```
Map: ("hello", 1), ("world", 1), ("hello", 1)
Reduce: ("hello", 2), ("world", 1)
```

### 3. YARN (Yet Another Resource Negotiator)

- Manages resources (CPU, memory) across machines in the Hadoop cluster.
- Allocates computing tasks efficiently.

**Analogy:** YARN is like a task manager that distributes work to multiple workers.

## Apache Spark

Spark is a **faster, in-memory alternative to Hadoop MapReduce** for distributed data processing.

### 1. RDDs (Resilient Distributed Datasets)

- Core Spark abstraction.
- Immutable collections of objects distributed across nodes.
- Support **fault-tolerant**, parallel operations (e.g., map, filter, reduce).

```
rdd = sc.parallelize([1, 2, 3, 4])
rdd.map(lambda x: x * 2).collect() # [2, 4, 6, 8]
```

## 2. DataFrames

- Like tables in a database or pandas in Python.
- Built on top of RDDs but more optimized.
- Support SQL-like operations (select, filter, groupBy).

```
df = spark.read.csv("data.csv", header=True)
df.select("name", "age").show()
```

## 3. Spark SQL

- Enables writing SQL queries on Spark DataFrames.
- Good for analysts who prefer SQL over programming.

```
SELECT name, COUNT(*) FROM people GROUP BY name;
```

## 4. MLlib

- Spark's built-in **Machine Learning library**.
- Supports classification, regression, clustering, etc.
- Works at scale on large datasets.

**Example:** Train a logistic regression model on millions of user records in parallel.

## 5. GraphX

- Spark library for **graph processing** (like social networks).
- Supports PageRank, connected components, and more.

**Example:** Analyzing LinkedIn connections to find mutual friends.

## Apache Flink vs Spark

Feature	Apache Spark	Apache Flink
<b>Processing Type</b>	<b>Batch-first</b> , supports streaming	<b>Stream-first</b> , also supports batch
<b>Latency</b>	Higher (micro-batching)	Very low (true real-time)
<b>Use Case</b>	ETL, ML, batch analytics	Real-time fraud detection, stock price analysis
<b>State Management</b>	Basic	Advanced, built-in checkpointing
<b>Event Time Support</b>	Moderate	Excellent (event time + watermarks)

## Summary:

- Use **Spark** when you're doing large-scale data analysis, machine learning, or batch jobs.
- Use **Flink** when you need **true real-time** stream processing.

## Python-Native Alternatives

### 1. Dask

- Parallel computing library for **Python**, designed to scale pandas, NumPy, and scikit-learn.
- Supports parallel arrays, dataframes, and ML models.
- Runs on a single machine or across a cluster.

```
import dask.dataframe as dd
df = dd.read_csv('data/*.csv')
df.groupby('city').mean().compute()
```

**Use Case:** You already use pandas but your data is too big for memory.

### 2. Ray

- Python framework for **distributed computing and machine learning**.
- Built for **low-latency, actor-based** execution.
- Used in AI/ML libraries like **RLLib**, **Modin**, and **Hugging Face Accelerate**.

```
import ray

@ray.remote
def square(x):
    return x * x

futures = [square.remote(i) for i in range(5)]
results = ray.get(futures)
```

**Use Case:** Scalable Python ML workloads (e.g., training many models in parallel).

**Summary Table**

Tool/Framework	Best For	Language Support	Real-time Capable	Batch Processing	Machine Learning
<b>Hadoop MapReduce</b>	Batch jobs on HDFS	Java	✗	✓	✗
<b>Spark</b>	In-memory analytics + ML	Scala, Python	✓ (limited)	✓	✓ (MLlib)
<b>Flink</b>	Real-time stream processing	Java, Scala	✓ (true stream)	✓	✗
<b>Dask</b>	Pythonic big data processing	Python	✗ (basic)	✓	✓
<b>Ray</b>	Distributed Python & ML	Python	✓	✓	✓

# ETL and ELT Pipelines in Data Engineering

## What is ETL?

ETL stands for:

1. **Extract** – Get data from source systems (e.g., databases, APIs, flat files).
2. **Transform** – Clean, format, join, and apply business logic to the data.
3. **Load** – Store the transformed data into a **data warehouse** or **data mart**.

ETL is best when your **data warehouse cannot handle raw data**, and transformations must happen **before loading**.

### ETL Steps (in order)

Step	Description	Example
Extract	Pull data from MySQL, APIs, CSVs	Orders from MySQL
Transform	Clean nulls, join tables, convert formats	Calculate total revenue per user
Load	Insert transformed data into Snowflake	Upload final report into analytics layer

### What is ELT?

ELT stands for:

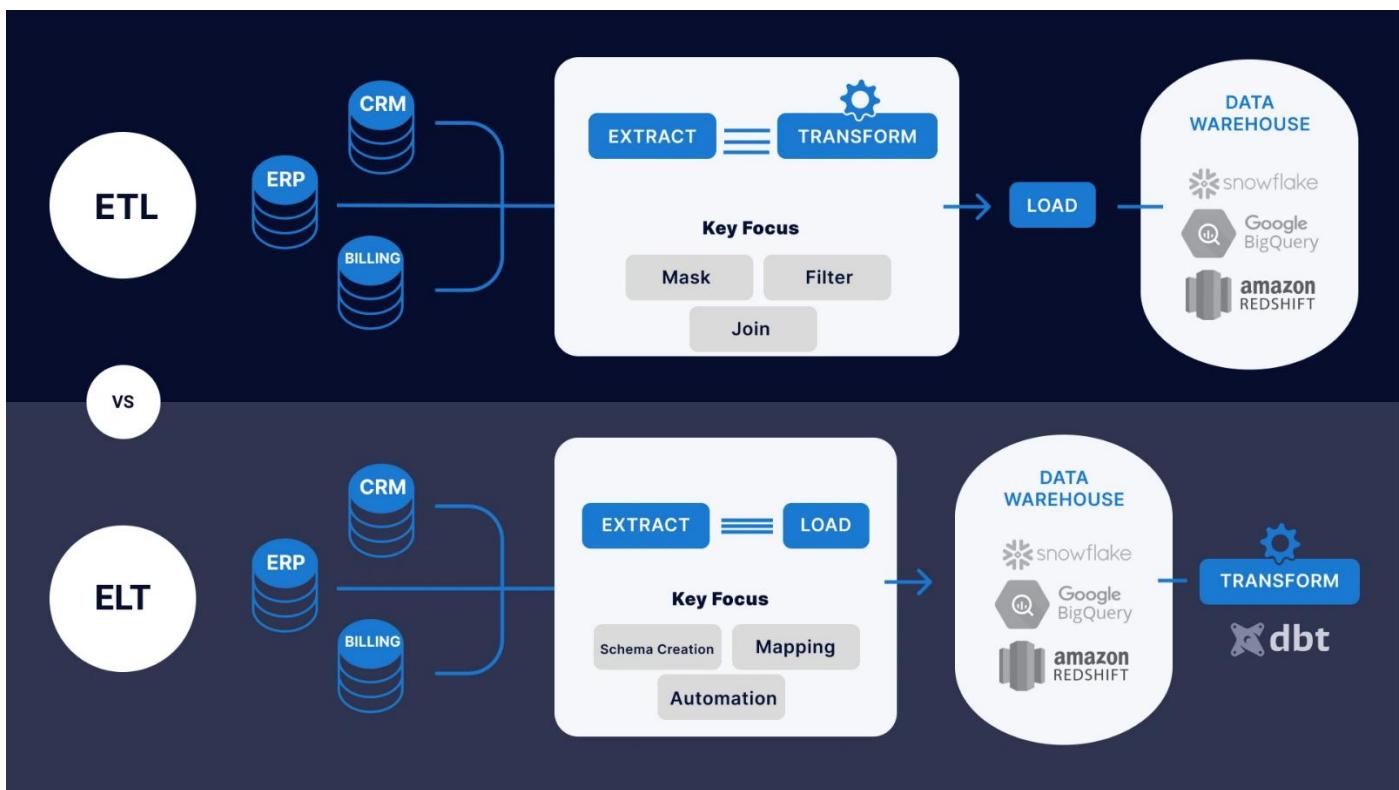
1. **Extract** – Get raw data from source systems.
2. **Load** – Store the raw data **immediately** into a data lake or warehouse.
3. **Transform** – Perform transformations **after loading**, using powerful warehouse engines (like BigQuery or Snowflake).

ELT is ideal for **modern cloud data platforms** where compute and storage are scalable.

### ELT Steps (in order)

Step	Description	Example
Extract	Pull raw logs from IoT sensors	JSON log files from 1000 devices
Load	Store directly in data lake/warehouse (S3, BigQuery)	Save logs into raw zone of S3
Transform	Run SQL to clean and analyze later	Parse and aggregate sensor values by region

## ⚖️ ETL vs ELT: Key Differences



	ETL	ELT
Feature		
<b>Transform Timing</b>	Before loading	After loading
<b>Storage First?</b>	No (Transform first)	Yes (Load first)
<b>Tooling</b>	Apache NiFi, Talend, Informatica, Airflow	dbt, BigQuery, Snowflake, Databricks
<b>Data Volume</b>	Moderate	Very Large (Petabyte-scale)
<b>Data Destination</b>	Traditional Data Warehouse	Modern Cloud Warehouses & Lakes
<b>Use Case</b>	Legacy BI reporting	Machine Learning, Real-time Analytics

## 🚀 Popular Tools for ETL/ELT

Purpose	ETL Tools	ELT Tools
<b>Orchestration</b>	Apache Airflow, Luigi	Apache Airflow, Prefect
<b>Transformation</b>	Talend, Pentaho, Informatica	dbt (data build tool), Spark SQL
<b>Loading</b>	AWS Glue, Stitch, Fivetran	Fivetran, Airbyte, BigQuery Load
<b>Storage</b>	Redshift, Snowflake	Snowflake, BigQuery, Delta Lake

## Example: Retail Data Pipeline

### ETL Approach:

- Extract orders from PostgreSQL
- Transform: Join with product data, clean nulls, convert currencies
- Load into Redshift for reporting

### ELT Approach:

- Extract raw orders, customer data, product details
- Load everything into BigQuery
- Use dbt to create cleaned, joined views inside BigQuery

### 🧠 When to Use Which?

Scenario	Use ETL	Use ELT
Legacy on-prem systems	✓	✗
Cloud-native data stack	✗	✓
Complex transformations upfront	✓	✗
Store everything, transform later	✗	✓
Tight security and compliance	✓	✓

# Stages of a Data Pipeline

A **data pipeline** is a series of steps that move and transform data from **source systems** to **target destinations** (like data warehouses, lakes, dashboards, or ML models).

## Typical Stages in a Data Pipeline

### 1. Data Generation

- **What happens:** Data is created or captured from various sources.
- **Sources include:**
  - User activity on websites or apps
  - Sensors/IoT devices
  - Application logs
  - Databases (MySQL, MongoDB)
  - APIs and third-party services
- **Example:** A user places an order in an app; the system records customer, item, and payment details.

### 2. Data Ingestion

- **What happens:** Raw data is **collected and moved** from source to the staging or raw zone.
- **Methods:**
  - **Batch ingestion** (e.g., hourly uploads)
  - **Streaming ingestion** (real-time, continuous flow)
- **Tools:** Apache Kafka, Apache NiFi, Flume, AWS Kinesis, Airbyte
- **Example:** Kafka streams website click events into a raw S3 bucket.

### 3. Data Storage

- **What happens:** Data is stored for processing, either raw or cleaned.
- **Storage Types:**
  - **Data Lakes** (S3, Azure Data Lake, HDFS) → raw, all formats
  - **Data Warehouses** (Snowflake, BigQuery) → structured, optimized
  - **Databases** (PostgreSQL, MongoDB) → for app-level use
- **Example:** Store raw log files from servers in Amazon S3 for later analysis.

## **4. Data Processing & Transformation**

- **What happens:** Data is **cleaned, formatted, joined, enriched**, and business logic is applied.
- **Types:**
  - **ETL** (Transform before loading into warehouse)
  - **ELT** (Load raw data, then transform in the warehouse)
- **Tools:** Apache Spark, Flink, dbt, Hadoop, Pandas (small scale), SQL
- **Example:** Convert timestamps to standard format, remove duplicates, join orders with customer data.

## **5. Data Validation & Quality Checks**

- **What happens:** Ensure data is accurate, consistent, complete, and trustworthy.
- **Checks include:**
  - Null values, duplicate rows
  - Schema mismatches
  - Range or integrity constraints
- **Tools:** Great Expectations, Deequ, custom SQL validations
- **Example:** Check that every order has a customer ID and a valid payment status.

## **6. Data Loading / Storage in Target System**

- **What happens:** Processed data is saved in the final storage or output layer.
- **Target systems:**
  - Data Warehouses (for BI tools)
  - Data Marts (departmental subsets)
  - Feature Stores (for ML)
- **Example:** Load transformed data into Snowflake for reporting.

## **7. Data Orchestration & Workflow Management**

- **What happens:** Automate and monitor pipeline stages (schedule, retry, alert).
- **Tools:** Apache Airflow, Prefect, Dagster, Luigi
- **Example:** Airflow runs a daily job that pulls data from APIs, transforms it, and updates dashboards.

## **8. Data Consumption / Analytics**

- **What happens:** Data is used by end users, apps, dashboards, or ML models.
- **Use cases:**
  - BI reporting (Tableau, Power BI)
  - Predictive analytics
  - Real-time dashboards
  - Personalized recommendations
- **Example:** A marketing dashboard shows daily user activity trends based on processed data.

## **9. Monitoring, Logging, and Lineage**

- **What happens:** Track pipeline performance, detect failures, and log data movement.
- **Tools:** Grafana, Prometheus, OpenLineage, Amundsen
- **Example:** You get an alert if yesterday's data ingestion failed or if data volume dropped.

# Hadoop Architecture

## What is Hadoop?

Apache Hadoop is an **open-source framework** that allows for the **distributed storage and processing** of massive datasets across clusters of commodity hardware (normal computers).

It was designed to solve the problems of:

- **Massive data volume**
- **Limited memory on single machines**
- **Slow processing of huge data using traditional systems**

Hadoop enables the storage and computation of **petabytes of data** across many machines.

## Hadoop Was Inspired by GFS

### GFS – Google File System

- **Developed by Google** to handle **web-scale data** across thousands of servers.
- Features:
  - Distributed storage with replication.
  - Master-slave architecture.
  - Chunking of large files into manageable pieces.
- Apache Hadoop's **HDFS** (Hadoop Distributed File System) was modeled after GFS.

## Core Components of Hadoop Architecture

1. **HDFS (Hadoop Distributed File System)** – For **massive storage** of data.
2. **MapReduce** – For **parallel data processing** across the cluster.
3. **YARN (Yet Another Resource Negotiator)** – Manages system resources and job scheduling.

Let's dive into each

### 1. HDFS – Hadoop Distributed File System

- **Stores huge files** across a cluster of machines.
- Files are split into fixed-size blocks (default: 128MB).
- Each block is **replicated (default: 3 times)** across different nodes for **fault tolerance**.
- **Two main nodes:**
  - **NameNode:** Stores metadata (file names, block locations)
  - **DataNode:** Stores actual data blocks

 Example: A 1GB file is split into 8 blocks (128MB each) and distributed across different machines.

## 2. MapReduce – Processing Engine

MapReduce is a **programming model** used for **processing large data sets** in a distributed manner using parallel computing.

### Two Main Phases:

Phase	Description	Example
Map	Splits data into key-value pairs	(“apple”, 1), (“banana”, 1)
Reduce	Aggregates or summarizes values by key	(“apple”, 5), (“banana”, 3)

**Real-world use case:** Word count in thousands of books stored in HDFS.

### Hadoop’s Two Main Tasks

#### 1. Massive Data Storage

- Achieved through **HDFS**, which can store petabytes of data across machines.
- It uses replication and chunking for scalability and fault tolerance.

#### 2. Faster Parallel Processing

- Handled by **MapReduce**, which splits jobs into smaller tasks and processes them in parallel.
- This reduces total computation time significantly.

### Properties of Hadoop

#### 1. Scalability

- Hadoop is **horizontally scalable** — you can add more machines (nodes) to handle more data and computation.

#### 2. Fault Tolerance

- HDFS **replicates data blocks** on multiple nodes.
- If a node fails, another replica is used automatically without data loss.
- MapReduce tasks are retried on failure.

#### 3. Distributed Processing

- Data and computation are distributed across the cluster.
- Processing is **data-local**, meaning code is moved to the data instead of moving data to code.

#### 4. Cost-Effective

- Runs on **commodity hardware** (low-cost machines).
- No need for expensive, high-end servers.

## 5. Open-Source

- Developed by the **Apache Software Foundation**.
- Free to use, widely supported by a large community.

## Real-World Use Case Example

**Company:** Facebook

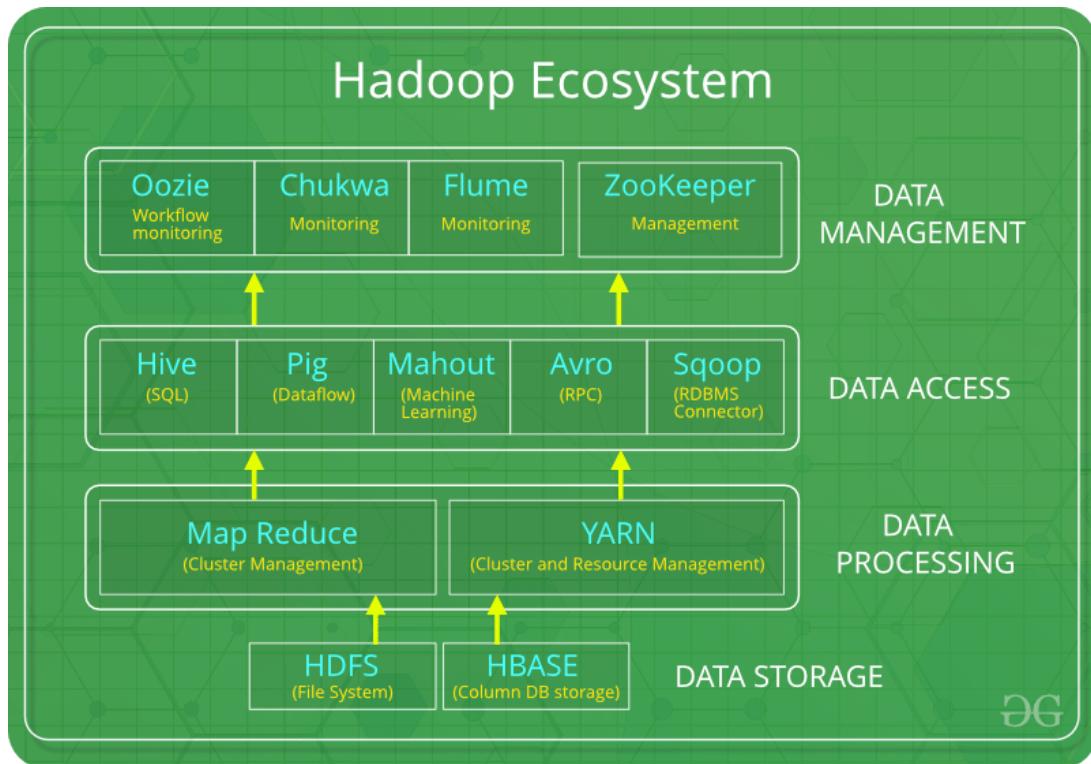
- Stores billions of posts, images, and videos in HDFS.
- Uses MapReduce to generate insights like top shared posts, trending tags, and user engagement metrics daily.

## Final Takeaway

Apache Hadoop revolutionized how we handle big data by providing:

- **Massive distributed storage** (HDFS)
- **Fast parallel processing** (MapReduce)
- All while being **scalable, fault-tolerant, and cost-effective**.

## Hadoop Components (Core Ecosystem)



### HDFS (Hadoop Distributed File System) – Storage Layer

📁 “Where data is stored.”

- A distributed file system that stores large files across a **cluster of machines**.
- Splits files into fixed-size blocks (e.g., 128MB) and **replicates** them across nodes for fault tolerance.

#### ◆ Main Nodes:

Role	Description
<b>NameNode</b>	Stores metadata (file names, block locations)
<b>DataNode</b>	Stores actual data blocks

**Example:** A 1TB file is split into chunks and stored across 50 machines with 3 copies of each chunk.

## MapReduce – Processing Layer

 “How data is processed.”

- A **programming model** for parallel processing of large datasets across the cluster.
- Breaks down tasks into **Map** (split and convert to key-value pairs) and **Reduce** (aggregate and compute).

**Example:** Counting how many times each word appears in millions of documents.

## YARN (Yet Another Resource Negotiator) – Resource Management Layer

 “Who manages what runs, where, and when.”

- Acts as Hadoop's **operating system** for allocating CPU, memory, and task execution across nodes.

### ◆ Components:

Role	Description
<b>ResourceManager</b>	Allocates system resources (CPU, memory)
<b>NodeManager</b>	Manages resources and tasks on individual machines
<b>ApplicationMaster</b>	Coordinates execution of a single application/job

**Analogy:** YARN is like an airport air traffic controller – deciding which plane (job) can land (run) and where (which machine).

## Hadoop Common – Utilities and Libraries

 “The toolkit that supports everything else.”

- Shared utilities, Java libraries, and APIs needed by HDFS, MapReduce, and YARN.
- Provides basic file and network IO, serialization, and security.

## How They Work Together

1. Data is **stored** in HDFS.
2. Jobs are **scheduled** by YARN.
3. Data is **processed** using MapReduce (or other engines like Spark).
4. Common utilities tie everything together and keep it running smoothly.

## Optional but Related Tools in the Hadoop Ecosystem

Tool	Purpose
Hive	SQL-like queries on Hadoop data
Pig	Script-based data transformations
HBase	NoSQL database on top of HDFS
Sqoop	Import/export data from RDBMS
Flume	Collect logs and streaming data
Oozie	Workflow scheduling and coordination
Zookeeper	Cluster coordination and metadata

## Summary Table

Component	Type	Function
HDFS	Storage	Store massive data files with replication
MapReduce	Processing	Batch processing in parallel
YARN	Resource Mgmt	Allocate and manage computing resources
Hadoop Common	Utilities	Core libraries and configuration

## Apache Hive?

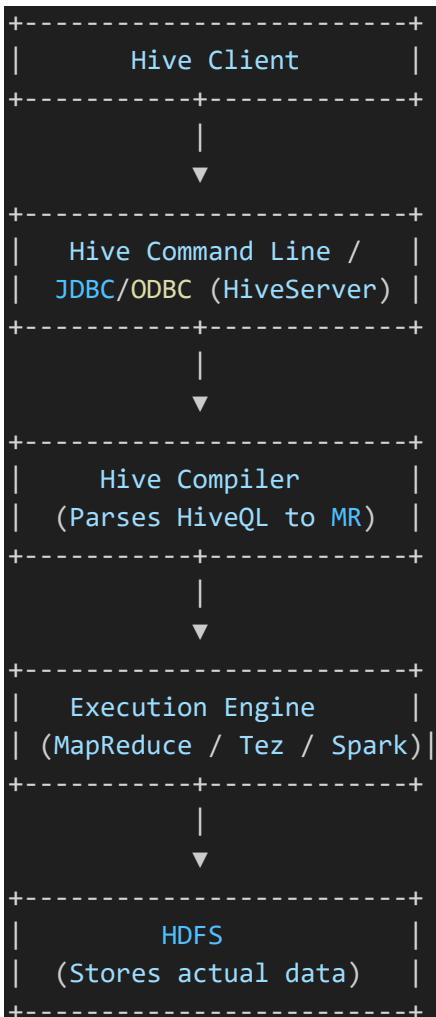
Apache Hive is a **data warehouse system** built on top of Hadoop. It provides a **SQL-like interface (HiveQL)** to query and manage large datasets stored in HDFS (Hadoop Distributed File System).

💡 In simple terms: Hive lets you use **SQL** to interact with **Big Data stored in Hadoop**, without needing to write complex MapReduce code.

### Key Features of Hive

Feature	Description
<b>SQL-like interface</b>	Query big data using HiveQL, similar to traditional SQL
<b>Built for Hadoop</b>	Translates queries into <b>MapReduce</b> , <b>Tez</b> , or <b>Spark</b> jobs
<b>Schema on Read</b>	You define the schema when you read the data (not when it's written)
<b>Supports large datasets</b>	Handles petabytes of structured data
<b>Integration-friendly</b>	Connects with BI tools, Apache Spark, HBase, and more

### Hive Architecture



## Hive Components

Component	Function
Metastore	Stores metadata (table schema, column types, location)
Driver	Parses queries and creates an execution plan
Compiler	Converts HiveQL into execution jobs
Execution Engine	Executes jobs using MapReduce, Tez, or Spark

## HiveQL – Query Language

HiveQL is very similar to standard SQL. Here are some examples:

### ❖ Create Table

```
CREATE TABLE students (
    id INT,
    name STRING,
    age INT
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ',';
```

### ❖ Load Data

```
LOAD DATA INPATH '/user/hive/input/students.csv' INTO TABLE students;
```

### ❖ Query Data

```
SELECT name, age FROM students WHERE age > 20;
```

### ❖ Aggregate

```
SELECT age, COUNT(*) FROM students GROUP BY age;
```

## ❖ Hive Table Types

Table Type	Description
Managed Table	Hive manages both data and metadata
External Table	Hive manages only metadata; data stays external

## Execution Flow

1. User submits HiveQL.
2. Hive parses and compiles the query.
3. Hive creates an execution plan (DAG).
4. Jobs are executed on Hadoop (MapReduce / Tez / Spark).
5. Results are returned to the user.

## When to Use Hive

### When you:

- Have massive structured datasets on HDFS
- Want to use SQL instead of Java/MapReduce
- Need integration with BI/reporting tools
- Want to query data in S3, HDFS, or Hive tables

### Avoid if:

- You need **real-time** querying (Hive is batch-based)
- You work with **unstructured or semi-structured** data (JSON, XML)

## Real-World Use Cases

Industry	Use Case
E-commerce	Product search analysis, customer behavior
Banking	Transaction summaries and fraud detection
Telecom	Call record analysis
Healthcare	Patient visit trends, drug usage reports

## Hive vs SQL Databases

Feature	Hive	Traditional SQL (e.g., MySQL)
Query Engine	Batch (MapReduce/Tez/Spark)	Real-time engine
Data Size	Petabyte-scale	GB/TB-scale
Schema Type	Schema on read	Schema on write
Speed	Slower (batch)	Faster (indexed, OLTP)
Best Use	Big Data Analytics	Transactional apps (OLTP)

## Summary

- Hive is a **powerful Big Data SQL engine** on top of Hadoop.
- Allows querying massive datasets using familiar SQL.
- Uses **HDFS for storage and MapReduce/Spark for computation**.
- Ideal for **batch analytics, data summarization, and ETL**.

## Apache Pig

### What is Apache Pig?

Apache Pig is a **high-level platform** for processing large-scale data in **Hadoop**.

It uses a **scripting language called Pig Latin** that makes it easier to write data transformations without directly writing complex MapReduce jobs.

 Pig is like a "data flow language" that converts scripts into **MapReduce jobs** and runs them on **HDFS**.

### Why Use Pig?

- Writing **MapReduce in Java is hard and verbose**.
- Pig makes it simple to perform operations like filtering, grouping, joining, and sorting.
- Especially useful for **data engineers** doing ETL (Extract, Transform, Load) jobs.

### Key Features of Apache Pig

Feature	Description
Pig Latin Language	Easy-to-read scripting for data processing tasks
Runs on Hadoop	Executes scripts as MapReduce jobs on HDFS
Extensible	Allows UDFs (User Defined Functions) in Java, Python, etc.
Schema Flexibility	Can handle structured, semi-structured, and unstructured data
Handles complex data flows	More efficient than SQL for multi-step data pipelines

## Apache Pig Architecture

```
User (Pig Latin Script)
      ↓
    Pig Compiler
      ↓
Logical Plan → Physical Plan
      ↓
MapReduce Jobs
      ↓
Hadoop (HDFS)
```

## Workflow

1. Write a Pig script using **Pig Latin**
2. Pig converts it into a **logical plan**
3. Then it converts to a **physical execution plan**
4. The plan is executed as **MapReduce jobs** on Hadoop

## Pig Latin – The Language

Pig Latin is a procedural language where each step is explicitly defined.

### Example: Word Count

```
-- Load data from HDFS
lines = LOAD '/input/data.txt' AS (line:chararray);

-- Split lines into words
words = FOREACH lines GENERATE FLATTEN(TOKENIZE(line)) AS word;

-- Group and count words
grouped = GROUP words BY word;
word_count = FOREACH grouped GENERATE group AS word, COUNT(words) AS count;

-- Store result back into HDFS
STORE word_count INTO '/output/result';
```

### Example: CSV File Processing

```
students = LOAD 'students.csv' USING PigStorage(',') AS (id:int, name:chararray,
marks:int);
top_students = FILTER students BY marks > 80;
DUMP top_students;
```

## When to Use Apache Pig

### Ideal for

- ETL jobs (cleansing, transforming, aggregating data)
- Processing logs, clickstreams, and large flat files
- Multi-step batch data pipelines

### Not ideal for:

- Real-time or interactive querying
- Use cases where SQL is more intuitive (use Hive instead)

## Pig vs Hive

Feature	Apache Pig	Apache Hive
Language	Pig Latin (procedural)	HiveQL (declarative SQL)
Use Style	Data flow scripts	Query-based (like databases)
Learning Curve	Slightly steeper	Easier for SQL users
Best For	Complex ETL pipelines	Ad-hoc queries and BI
Execution Engine	MapReduce	MapReduce, Tez, or Spark

## Pig Execution Modes

Mode	Description
Local Mode	Runs on a single machine (small data/testing)
MapReduce Mode	Runs on a Hadoop cluster via MapReduce

Run using:

```
pig -x local      # Local mode  
pig -x mapreduce # Hadoop cluster mode
```

## Real-World Use Cases

Industry	Use Case
E-commerce	Clean and transform clickstream logs
Banking	Parse transaction logs and summarize patterns
Advertising	Aggregate impressions, clicks, and conversions
Telecom	Analyze call records and user behavior

## Summary

- **Apache Pig** simplifies large-scale data processing on Hadoop using a scripting language.
- Ideal for **complex data flows and ETL pipelines**.
- Converts scripts into **MapReduce jobs**, allowing scalability.
- Works well for engineers familiar with programming over SQL.

## Apache Sqoop

### What is Apache Sqoop?

**Apache Sqoop** (SQL-to-Hadoop) is a **data transfer tool** that enables **efficient movement of data between relational databases and the Hadoop ecosystem**.

💡 In simple terms: Sqoop helps **import data from MySQL, PostgreSQL, Oracle, etc., into Hadoop**, and **export it back** after processing.

### ⚙️ Why Use Sqoop?

- Traditional RDBMS systems hold important data (sales, customers, etc.)
- Hadoop is used for **large-scale storage and analytics**
- Sqoop bridges the gap between **structured RDBMS** and **Big Data processing platforms**

## Key Features

Feature	Description
<b>Bidirectional transfer</b>	Import from RDBMS to HDFS, Hive, HBase; Export from HDFS to RDBMS
<b>Parallel processing</b>	Uses MapReduce to speed up data transfer in parallel chunks
<b>Supported databases</b>	MySQL, PostgreSQL, Oracle, SQL Server, DB2, etc.
<b>Flexible formats</b>	Supports CSV, Avro, Parquet, and SequenceFile formats
<b>Integration</b>	Works with HDFS, Hive, HBase, and Hadoop ecosystems

## How Sqoop Works

1. You write a **Sqoop command** to specify
  - o Source (e.g., MySQL)
  - o Destination (e.g., HDFS)
  - o Table or query to pull data from
2. Sqoop
  - o Connects to the RDBMS
  - o Splits the data into **multiple parallel tasks**
  - o **Runs MapReduce jobs** to extract and load the data efficiently

## Common Sqoop Commands

### Import from MySQL to HDFS

```
sqoop import \
--connect jdbc:mysql://localhost/school \
--username root --password yourpassword \
--table students \
--target-dir /user/hadoop/students_data \
--m 4
```

- ◆ --m 4: Imports data using 4 parallel mappers.

### Export from HDFS to MySQL

```
sqoop export \
--connect jdbc:mysql://localhost/school \
--username root --password yourpassword \
--table processed_students \
--export-dir /user/hadoop/cleaned_data \
--input-fields-separated-by ','
```

### Import Data Directly into Hive

```
sqoop import \
--connect jdbc:mysql://localhost/school \
--username root --password yourpassword \
--table students \
--hive-import \
--hive-table hive_students \
--m 1
```

## Import Using SQL Query

```
sqoop import \
--connect jdbc:mysql://localhost/school \
--username root --password yourpassword \
--query 'SELECT id, name FROM students WHERE $CONDITIONS' \
--split-by id \
--target-dir /user/hadoop/query_students \
--m 2
```

## Real-World Use Cases

Use Case	Description
Retail	Import sales orders from MySQL → analyze with Hive
Finance	Export processed fraud data back to Oracle DB
Marketing	Load campaign performance data into Hadoop
Healthcare	Migrate patient logs from SQL Server to HDFS

## Limitations of Sqoop

- Batch-only (no real-time streaming)
- Requires JDBC drivers and connectivity
- Replaced in some modern pipelines by tools like **Apache Nifi**, **Airbyte**, or **Kafka Connect** in real-time environments

## Summary Table

Feature	Sqoop Does...
 Import	SQL → HDFS, Hive, HBase
 Export	HDFS → SQL
 Fast	Parallel import/export using MapReduce
 File Format	Text, CSV, Avro, Parquet
 Source	MySQL, PostgreSQL, Oracle, etc.

# Apache Oozie

## What is Apache Oozie?

Apache Oozie is a **workflow scheduler system** for managing Hadoop jobs.

It allows you to **define, schedule, and coordinate complex data workflows** involving multiple Hadoop ecosystem components like:

- MapReduce
- Hive
- Pig
- Sqoop
- Spark
- Shell scripts

 Think of Oozie as the “**Airflow for Hadoop**” — it automates the execution of Big Data jobs in a defined sequence.

## 💡 Why Use Oozie?

In large data pipelines, jobs depend on each other. For example:

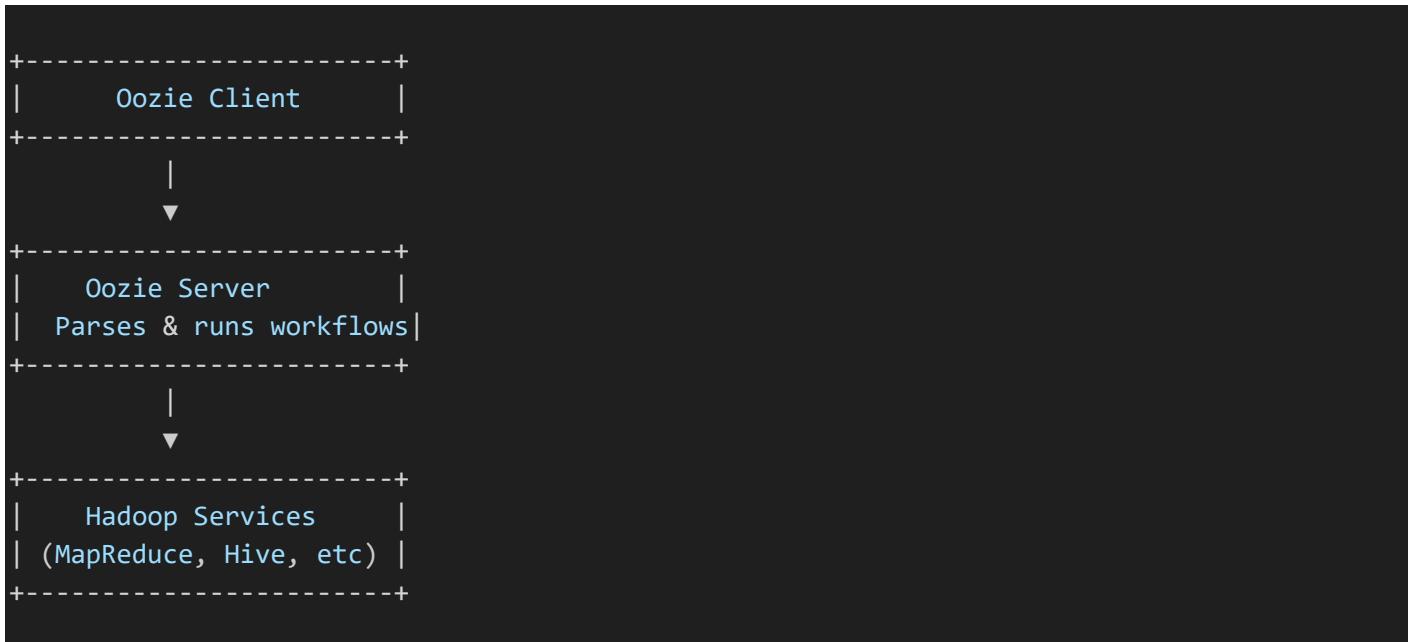
1. Extract data using **Sqoop**
2. Clean it using **Pig**
3. Load it into **Hive**
4. Generate a **report using Spark**

Oozie manages this flow: **who runs what, when, and under what conditions.**

## Key Features of Apache Oozie

Feature	Description
<b>Workflow orchestration</b>	Manage multi-step Hadoop tasks in sequence
<b>Time and event triggers</b>	Supports scheduling based on time (cron) or data availability
<b>Retries &amp; recovery</b>	Can retry failed jobs automatically
<b>Extensible</b>	Can include custom shell scripts or Java actions
<b>Integrated with Hadoop</b>	Native support for MapReduce, Hive, Pig, etc.

## Oozie Architecture Overview



## Components of Oozie

Component	Description
Workflow	A <b>Directed Acyclic Graph (DAG)</b> of actions (e.g., MapReduce → Hive → Email)
Coordinator	Triggers workflows based on <b>time, events, or data availability</b>
Bundle	Groups multiple coordinators together for large pipeline management
Actions	Tasks in a workflow (Hive, Pig, Java, Shell, Sqoop, etc.)

## Workflow Example (XML-based)

Here's a simple example of an Oozie workflow file (workflow.xml) that runs a Pig job

```
<workflow-app name="pig-wf" xmlns="uri:oozie:workflow:0.5">
    <start to="pig-node"/>

    <action name="pig-node">
        <pig>
            <script>script.pig</script>
        </pig>
        <ok to="end"/>
        <error to="fail"/>
    </action>

    <kill name="fail">
        <message>Job failed</message>
    </kill>

    <end name="end"/>
</workflow-app>
```

Oozie reads this file and knows how to execute the steps.

## Scheduling with Coordinator

A coordinator can schedule a job to run **every hour**, **daily**, or **based on data arrival**.

```
<coordinator-app name="coord-example" frequency="60"
  start="2025-06-01T00:00Z" end="2025-06-30T00:00Z"
  timezone="UTC" xmlns="uri:oozie:coordinator:0.4">

  <action>
    <workflow>
      <app-path>${workflowPath}</app-path>
    </workflow>
  </action>
</coordinator-app>
```

## Real-World Use Cases

Use Case	Description
<b>ETL Pipelines</b>	Automate multi-step batch jobs
<b>Daily Reporting</b>	Run Hive/Spark queries at midnight
<b>Data Ingestion with Validation</b>	Schedule Sqoop → Pig → Hive sequence
<b>Retry Logic for Failed Jobs</b>	Retry failed steps automatically
<b>Email Alerts</b>	Notify users if workflows fail

## 💡 Oozie vs Other Workflow Tools

Feature	Oozie	Apache Airflow
<b>Language</b>	XML	Python
<b>Target Platform</b>	Hadoop (HDFS, MR, Pig, etc)	General-purpose
<b>Job Triggers</b>	Time, event, data	Time-based, sensor-based
<b>Learning Curve</b>	Moderate (XML heavy)	Easier (Python-based)
<b>Native Hadoop Support</b>	✓	✗ (uses operators/plugins)

## Summary

- **Apache Oozie** is a powerful workflow scheduler for managing Hadoop jobs.
- Supports **MapReduce, Hive, Pig, Sqoop, Spark**, and shell tasks.
- Uses **XML-based workflows** and **time/data-based coordinators**.
- Ideal for **batch pipelines, ETL flows, and data warehouse operations** in Hadoop.

## Apache HBase

### What is Apache HBase?

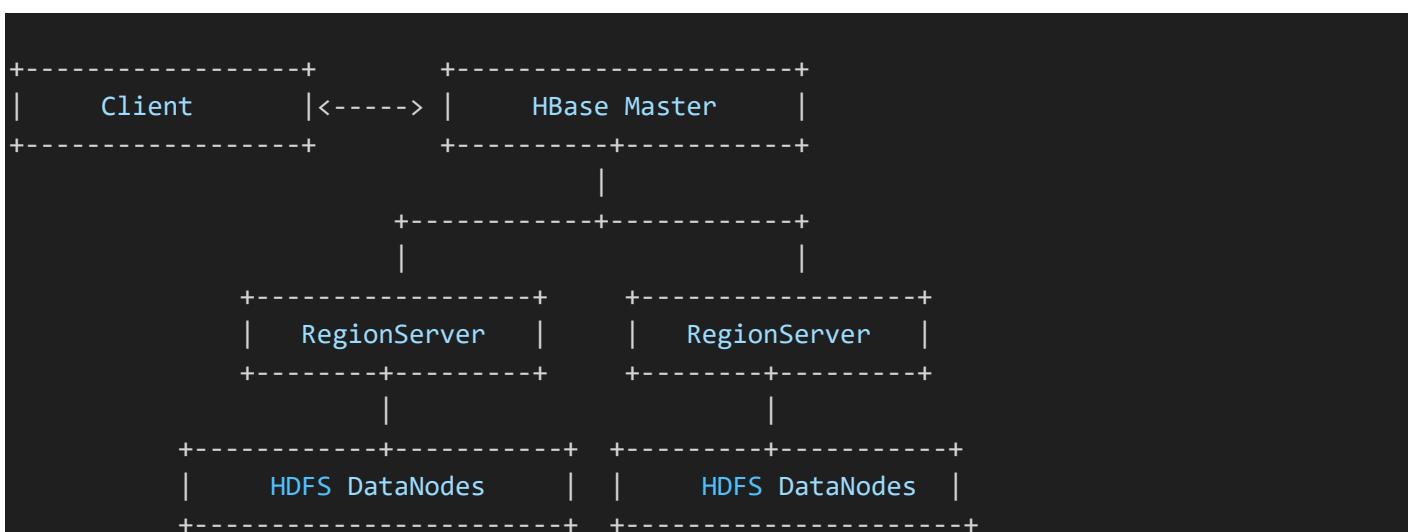
**Apache HBase** is a **distributed, scalable, NoSQL database** built on top of **Hadoop HDFS**. It is modeled after **Google's Bigtable** and is designed to **store and retrieve large quantities of sparse data** (datasets where most values are empty/null).

 Think of HBase as a **non-relational, column-oriented database** that supports **random read/write access** to big data.

### Key Features of HBase

Feature	Description
<b>Column-oriented</b>	Stores data by columns, not rows (efficient for analytics)
<b>NoSQL</b>	Doesn't use SQL tables—uses rows, column families, and cells
<b>Scalable</b>	Scales horizontally to billions of rows and millions of columns
<b>Real-time Access</b>	Supports real-time reads/writes—not batch like HDFS
<b>Tightly integrated with Hadoop</b>	Uses HDFS as its storage engine

### HBase Architecture Overview



## Key Components

Component	Role
<b>HBase Master</b>	Coordinates RegionServers and metadata (but not involved in I/O)
<b>RegionServer</b>	Handles reads/writes and stores <b>regions</b> (subsets of data)
<b>Regions</b>	Contain rows stored in <b>HFiles</b> on HDFS
<b>Zookeeper</b>	Manages coordination and configuration across cluster
<b>HDFS</b>	Underlying storage for actual data blocks

## HBase Data Model

Unlike relational databases, HBase organizes data as:

<Table>

```
Row Key →  
Column Family →  
Column Qualifier →  
Timestamp →  
Value
```

### Example

Row Key	Column Family: Info	Column: Name	Value: "Alice"	Timestamp
user1	info	name	Alice	2025-06-27

## Concepts in HBase

Concept	Description
<b>Row key</b>	Unique ID for each row (sorted lexicographically)
<b>Column family</b>	Logical group of columns (must be predefined)
<b>Column qualifier</b>	Specific column within a family (flexible)
<b>Cell</b>	Intersection of row, column, and timestamp
<b>Versioning</b>	Each cell can store multiple versions by timestamp

## HBase vs HDFS

Feature	HBase	HDFS
Data Access	Random real-time access (Get/Put)	Sequential batch access (read/write files)
Data Structure	Table (rows, columns, versions)	Flat files
Use Case	Fast lookup, updates	Long-term storage, batch processing
Example	User profile lookup	Store logs, images, videos

## HBase vs RDBMS

Feature	HBase	Traditional RDBMS (e.g., MySQL)
Schema	Flexible, dynamic columns	Fixed schema
Joins	Not supported (denormalized data)	Supported
Transactions	No full ACID, but atomic row-level	Full ACID
Query Language	Java API, REST, or shell (no SQL)	SQL
Ideal Use	Sparse, large, high-speed data	Structured, transactional data

## Real-World Use Cases

Industry	Use Case
Social Media	Store user profiles, timelines, likes
Telecom	Call records, logs, SMS metadata
Banking	Fraud detection logs, audit trails
IoT	Store sensor data with timestamps
E-commerce	Product catalog with real-time inventory updates

## 💡 Common HBase Commands

### Create a Table

```
create 'students', 'info'
```

### Insert Data

```
put 'students', '001', 'info:name', 'Alice'  
put 'students', '001', 'info:age', '23'
```

## Get Data

```
get 'students', '001'
```

## Scan All Rows

```
scan 'students'
```

## Summary

- Apache HBase is a **NoSQL columnar database** built for **real-time reads/writes** on **big data**.
- Runs on top of **HDFS**, supports **billions of rows**, and stores **multi-versioned** data efficiently.
- Great for **low-latency lookups**, unlike Hive or HDFS which are batch-oriented.

# Apache Flume

## What is Apache Flume?

Apache Flume is a **distributed, reliable, and available service** for **efficiently collecting, aggregating, and transporting large volumes of log data** from many sources (like web servers) to centralized stores (like **HDFS** or **HBase**).

💡 In simple terms: Flume is used to **stream logs or event data** from systems like **Apache web servers, application logs, or Twitter feeds** into **Hadoop for storage or analysis**.

## Why Use Apache Flume?

Reason	Description
Real-time log ingestion	Handles continuous streams of log or event data
Scalability	Easily scale up to handle high-volume sources
Reliability	Guarantees delivery even during node failure
Integration	Supports HDFS, HBase, Kafka, and other sinks
Custom sources	Allows custom sources and interceptors

## Core Components of Flume

A **Flume agent** is the fundamental unit. Each agent has:

Component	Function
<b>Source</b>	Accepts data from external sources (e.g., web server, syslog, Kafka)
<b>Channel</b>	Acts as a buffer (queue) between source and sink (e.g., memory or file)
<b>Sink</b>	Delivers data to final destination (e.g., HDFS, HBase, Elasticsearch)

## Flow

Source → Channel → Sink

### 💡 Apache Flume Architecture



### ✓ Example Use Case

Logs generated by a web server are collected by a **source**, buffered in a **channel**, and written to HDFS by a **sink**.

## Flume Sources, Channels, and Sinks

### 📍 Sources

- Avro
- Spooling Directory
- Syslog
- Kafka
- Netcat
- HTTP
- Twitter (via Flume plugin)

## Channels

- **Memory Channel** (faster, but not durable)
- **File Channel** (persistent to disk, fault-tolerant)

## Sinks

- HDFS
- HBase
- Kafka
- Elasticsearch
- Custom sinks

## Sample Flume Configuration File

```
# Agent name
agent1.sources = src
agent1.channels = ch
agent1.sinks = snk

# Source config
agent1.sources.src.type = netcat
agent1.sources.src.bind = localhost
agent1.sources.src.port = 44444

# Channel config
agent1.channels.ch.type = memory

# Sink config
agent1.sinks.snk.type = hdfs
agent1.sinks.snk.hdfs.path = hdfs://localhost:9000/logs

# Bind source and sink to channel
agent1.sources.src.channels = ch
agent1.sinks.snk.channel = ch
```

This config reads data from a Netcat connection and writes it into HDFS.

## Real-World Use Cases

Industry	Use Case
E-commerce	Stream clickstream data to Hadoop
Social Media	Capture user activity logs
Telecom	Ingest call logs from distributed towers
Finance	Collect transaction and audit logs
DevOps	Centralize server/app logs for analysis

## Flume vs Kafka

Feature	Flume	Kafka
Focus	Log collection and ingestion	Distributed messaging platform
Storage	Short-term buffering (channel)	Persistent messaging (topics)
Use Case	Log shipping to HDFS/HBase	Real-time data pipelines, stream processing
Reliability	FileChannel for durability	Highly durable by design
Integration	Hadoop native	Broader, with stream processing

## Summary

- Apache Flume is a **log/event ingestion tool** designed to move large amounts of data **into Hadoop-based systems**.
- It supports **real-time streaming, buffering, and reliable delivery**.
- Ideal for use cases involving **server logs, social data, and system events**.

## What is Apache ZooKeeper?

Apache ZooKeeper is a **centralized service** for **maintaining configuration information, naming, synchronization, and group services in distributed systems.**

💡 Think of ZooKeeper as the "**coordinator**" or "**referee**" that ensures all nodes in a distributed cluster **know what's happening, who is the leader, and what state they're in.**

## Why is ZooKeeper Needed?

Distributed systems have many independent nodes. They need to:

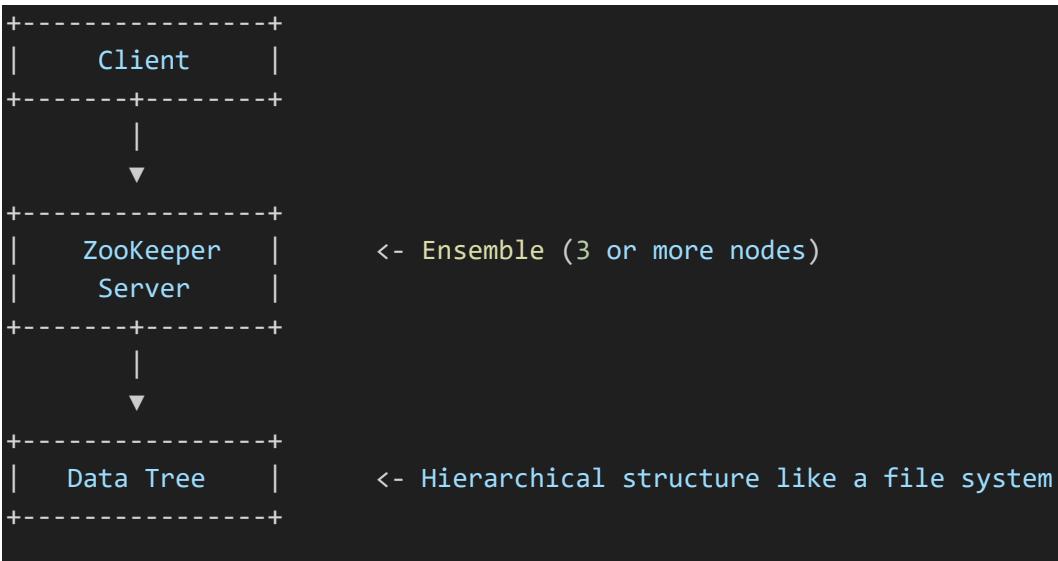
- **Coordinate** tasks (e.g., who is the leader?)
- **Maintain consistent configuration**
- **Detect node failures**
- **Avoid conflicts**

ZooKeeper **solves these problems** by acting as a **highly available, consistent, fault-tolerant coordination system.**

## What ZooKeeper Does

Feature	Purpose
<b>Configuration Management</b>	Store and share configuration between cluster nodes
<b>Naming Service</b>	Assign and track unique IDs to services or nodes
<b>Distributed Synchronization</b>	Coordinate actions like leader election or locking
<b>Cluster Membership</b>	Track live or dead nodes
<b>Leader Election</b>	Automatically elect a leader when needed

## ZooKeeper Architecture



## Components

Component	Role
ZNode	Basic data unit (like a file or directory) in ZooKeeper
Client	Any application that connects to ZooKeeper
Server	A node in the ZooKeeper ensemble
Leader	The main node responsible for handling write requests
Follower	Nodes that handle read requests and vote during elections
Ensemble	A group of ZooKeeper servers working together

## ZNode Hierarchy (Tree-like Structure)

Just like a filesystem

```
/  
├── config  
├── services  
│   └── kafka-broker-1  
└── locks  
    └── job-123-lock
```

Each **ZNode**

- Can store **small amounts of data**
- Can have **children nodes**
- Supports **watchers** (event listeners)

## ZooKeeper in Action: Common Use Cases

### 1. Configuration Management

- Shared config file stored in ZooKeeper (e.g., Kafka broker configs)

### 2. Leader Election

- Automatically choose a new leader node if the current one fails

### 3. Distributed Locking

- Prevent multiple nodes from accessing the same resource at the same time

### 4. Service Discovery

- Keep track of available services or machines

## Examples of ZooKeeper Usage

System	How ZooKeeper Helps
Apache Kafka	Manages broker metadata, leader elections
Apache HBase	Keeps track of region servers
Apache Hadoop YARN	Coordinates ResourceManager and high availability
Apache Storm	Coordinates workers and tasks

## How ZooKeeper Ensures Consistency

ZooKeeper uses a protocol called **Zab (ZooKeeper Atomic Broadcast)** to ensure:

- **Linearizable writes**
- **Sequential consistency**
- **Atomicity** of operations
- **Durability** even if some servers crash

 A write is successful only if **more than half of the servers (a quorum) acknowledge it.**

## Simple Command-Line Example

```
Start CLI:  
zkCli.sh -server localhost:2181  
Create a node:  
create /hello "world"  
Read a node:  
get /hello  
Set data:  
set /hello "zookeeper"
```

## Summary

- Apache ZooKeeper is a **central coordination service** for distributed systems.
- It provides **naming, configuration, synchronization, and leader election**.
- Used in major systems like **Kafka, Hadoop, HBase, and Storm**.
- Ensures **high availability and strong consistency**.

# Hadoop Terminologies

## File System

A **file system** is a method and data structure used by an operating system to **store, organize, and access data on storage devices** (like hard drives or SSDs). In Big Data, the type of file system you use can dramatically affect data processing, scalability, and fault tolerance.

### ◆ a. Standalone File System

A **standalone file system** is a traditional file system **used on a single machine**.

- **Examples:** NTFS (Windows), ext4 (Linux), APFS (macOS)
- **How it works:** All files and directories are stored and accessed **locally** on a hard drive.
- **Limitations in Big Data:**
  - No scalability across machines
  - Vulnerable to **single point of failure**
  - Not suitable for **massive datasets**

Not used in distributed environments like Hadoop.

### ◆ b. Distributed File System

A **Distributed File System (DFS)** stores files **across multiple machines (nodes)** in a cluster, providing a **single virtual file system interface**.

- **Examples:** HDFS (Hadoop), GFS (Google File System), Amazon S3 (object storage with DFS-like traits)
- **Key Properties:**
  - **Scalable:** Handles petabytes of data
  - **Fault-tolerant:** Replicates data across nodes
  - **Accessible in parallel:** Multiple machines can read/write simultaneously

## ❖ Relevance

Aspect	Effect in Big Data
<b>Scalability</b>	Add more machines to store and process more data
<b>Fault Tolerance</b>	Continues working even if some nodes fail
<b>Performance</b>	Data is stored near compute nodes (locality advantage)

## Block

In both local and distributed file systems, **data is stored in blocks**. A **block** is the smallest unit of data storage.

- ◆ In **HDFS**, the default block size is **128 MB** (can be customized).
- ◆ In traditional systems like NTFS or ext4, blocks are smaller, e.g., **4 KB**.

### Why Blocks Matter

- Large blocks reduce overhead in **metadata tracking**.
- Each block can be **replicated across multiple nodes** for fault tolerance.

### Relevance

Concept	Effect
<b>Block replication</b>	Ensures data availability even when nodes fail
<b>Parallelism</b>	Blocks can be processed in parallel by multiple nodes
<b>Storage efficiency</b>	Fewer large blocks = less metadata overhead

## Cluster

A **cluster** is a group of **interconnected computers (nodes)** that work together as a single system.

- 💡 In Hadoop, a cluster runs the entire ecosystem: **HDFS, YARN, Hive, Spark, etc.**
- All nodes in the cluster **share workload and storage**.

### Relevance

Feature	Effect
<b>Distributed computing</b>	Each node handles a part of the workload (MapReduce, Spark)
<b>High availability</b>	If one node fails, others take over
<b>Elastic scaling</b>	You can add/remove nodes as needed

## Node

A **node** is a **single machine/server** within a cluster.

### Types of Nodes:

- **Master Node:**
  - Controls the cluster
  - In HDFS: **NameNode**
  - In YARN: **ResourceManager**
- **Worker/Slave Node:**
  - Performs actual data storage or computation
  - In HDFS: **DataNode**
  - In YARN: **NodeManager**

### ❖ Relevance

Node Type	Role in Big Data Cluster
<b>NameNode</b>	Manages file system metadata (file names, block info)
<b>DataNode</b>	Stores actual data blocks
<b>ResourceManager</b>	Allocates compute resources for jobs
<b>NodeManager</b>	Executes individual processing tasks

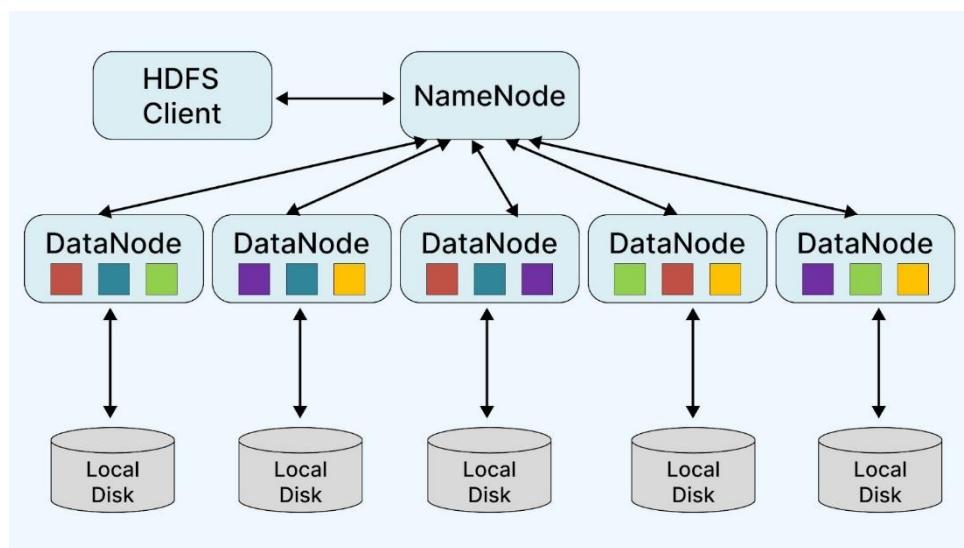
### Example Workflow in Hadoop

Let's say you're storing and processing a file in Hadoop:

1. You upload a file (e.g., 300 MB) → HDFS splits it into **3 blocks** (128 MB + 128 MB + 44 MB).
2. These blocks are **replicated** across **multiple nodes** for fault tolerance.
3. A **MapReduce** job is submitted; it is divided and distributed across **worker nodes** in the **cluster**.
4. Each node **processes its block locally**, improving speed and minimizing network load.
5. Results are **combined** and written back to HDFS.

## Summary Table

Term	Definition	Key Relevance in Big Data
File System	Organizes data storage	DFS (e.g., HDFS) enables scalable, fault-tolerant storage
Block	Smallest unit of storage	Supports replication and parallel processing
Cluster	Group of machines	Allows distributed storage and computing
Node	Individual machine in a cluster	Performs specific storage or processing tasks



## Process

### ✓ Definition

A **process** is a **running instance** of a program. When you run a program or script, the OS creates a **process** to execute it.

### ✖ Technical Characteristics:

- Has its own memory and CPU context
- Managed by the operating system
- Can run in the foreground or background

### ▀ Example in Big Data:

- Running a **MapReduce job** triggers multiple **processes**:
  - A process runs the **Mapper**
  - A process runs the **Reducer**
- Spark starts **executors** as processes to perform computations

## Daemon Process

### Definition

A **daemon** is a special type of background process that **runs continuously** and performs system-level or service-related tasks.

 Daemons are **always-on processes** that **wait for tasks**, unlike normal processes that run and exit.

### Naming

Daemon process names often end with d, like:

- sshd → handles SSH connections
- httpd → handles HTTP web requests

### Example in Hadoop:

Daemon	Role
NameNode daemon	Manages file system metadata
DataNode daemon	Stores and serves actual data blocks
ResourceManager daemon	Manages cluster resource allocation
NodeManager daemon	Executes tasks assigned by ResourceManager

## Metadata

### Definition:

Metadata is **data about data**. It describes **information like where data is stored, its size, permissions, and structure**—but **not the actual content**.

### Example in HDFS

When you upload a file to Hadoop:

- **Metadata** includes:
  - File name
  - File size
  - Number of blocks
  - Block locations (which DataNodes store them)
  - Permissions and ownership

The **NameNode** stores this metadata in memory and on disk.

### Why It's Important

- Metadata allows quick **file location lookup**
- Enables efficient **data management**
- Loss of metadata = total system failure (which is why NameNode is critical)

## Replication

### Definition

**Replication** means making **copies of data blocks** and **storing them on multiple nodes** for **fault tolerance** and **high availability**.

### Example in HDFS

- Default replication factor: **3**
- If you store a 128 MB block, it is copied to **3 different DataNodes**
  - If one node fails, the other 2 have the backup

### Key Effects:

Benefit	Description
<b>Fault Tolerance</b>	Data remains accessible even if nodes fail
<b>Data Availability</b>	Ensures continuous read access to data
<b>Load Balancing</b>	Multiple nodes can serve the same data

### Trade-off

More replication = more fault tolerance but **more storage usage**.

## Summary Table

Term	Description	Role in Big Data Systems
<b>Process</b>	A running instance of a program	Executes jobs like MapReduce, Spark tasks, etc.
<b>Daemon</b>	Background service process	Manages critical services in Hadoop (NameNode, etc.)
<b>Metadata</b>	Information about data	Tracks files, blocks, and locations in HDFS
<b>Replication</b>	Duplicates of data blocks	Provides fault tolerance and load balancing in HDFS

# HDFS Architecture

## Main Components

Component	Role
NameNode	Master node that <b>manages metadata</b> (file names, block info)
DataNode	Worker nodes that <b>store actual data blocks</b>
Secondary NameNode	Takes <b>periodic checkpoints</b> of the NameNode (not backup!)
Client	The application or user that reads/writes data

## How It Works: Step-by-Step

### 1. Storing a File in HDFS

Let's say you upload a 300 MB file.

1. Client contacts NameNode to ask:

- o "How do I store this file?"

2. NameNode:

- o Splits file into **blocks** (default block size = 128 MB)
- o File → Block1 (128MB), Block2 (128MB), Block3 (44MB)
- o Assigns each block to 3 different DataNodes (replication factor = 3)

3. Client writes data directly to DataNodes, block by block.

### 2. Reading a File from HDFS

1. Client contacts NameNode to ask:

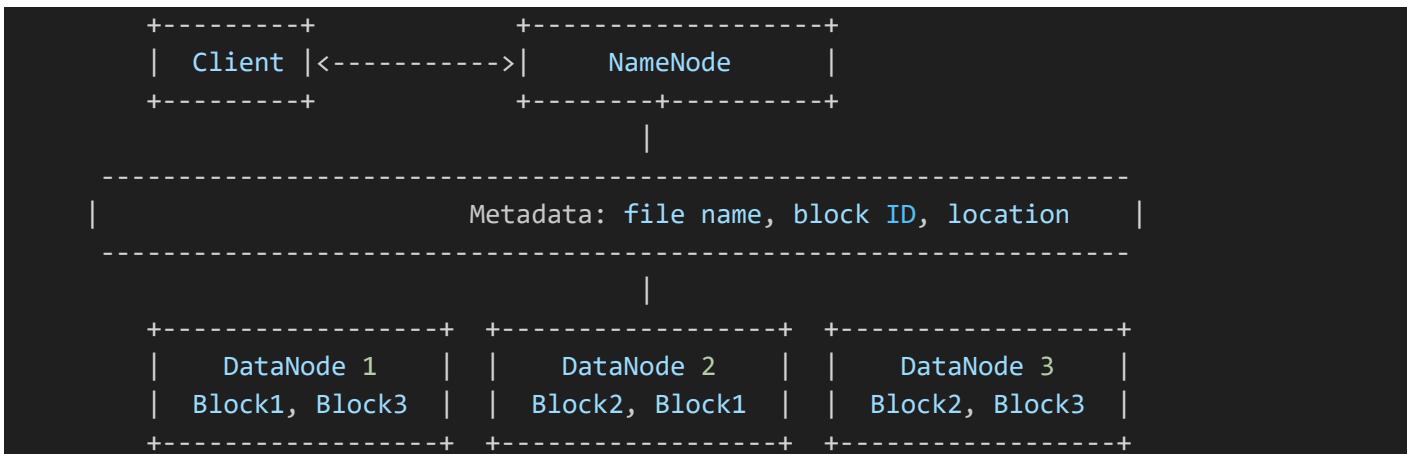
- o "Where are the blocks of this file?"

2. NameNode returns block locations

3. Client reads blocks from DataNodes directly

- o If a DataNode fails, client reads from another replica

## HDFS Architecture Diagram



💡 **NameNode** = Metadata only

💾 **DataNode** = Actual data blocks

## Replication in HDFS

By default, every block is **replicated 3 times** (replication factor = 3) and stored on different nodes.

- Ensures **fault tolerance**
- If 1 or 2 nodes go down, data is still available

## Technical Terms and Their Role

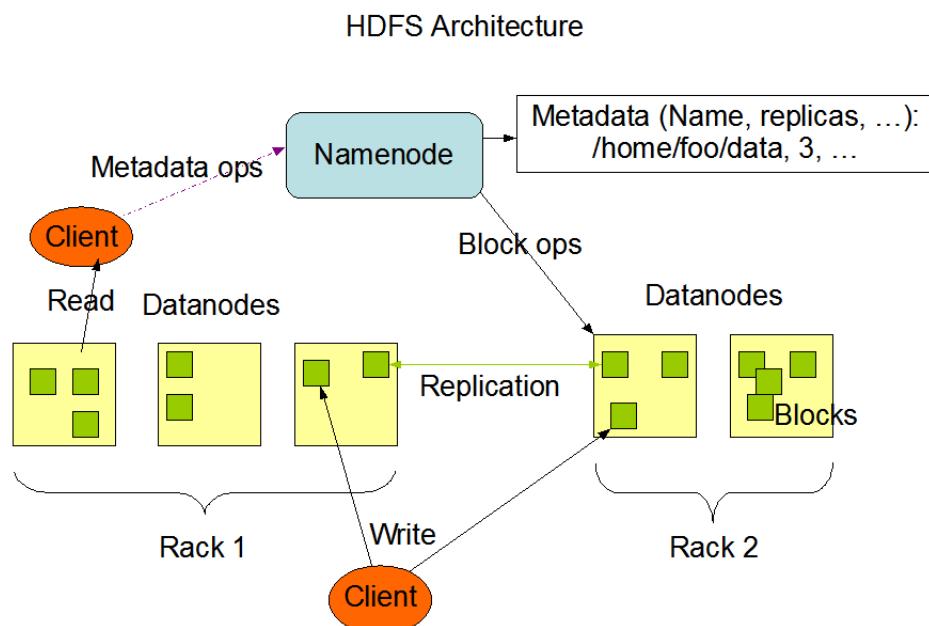
Term	Description	Role in HDFS
<b>Block</b>	Unit of data storage (default 128MB)	Splits large files into manageable chunks
<b>Replication</b>	Copying each block to multiple DataNodes	Ensures high availability and fault tolerance
<b>NameNode</b>	Stores only metadata (no actual data)	Knows which DataNode has what data
<b>DataNode</b>	Stores actual data blocks	Executes read/write as per NameNode instructions
<b>Secondary NameNode</b>	Periodically pulls metadata from NameNode to create checkpoints	Helps with recovery in case of NameNode failure

## Properties of HDFS

Property	Description
Fault Tolerant	Replication keeps data safe during failures
Scalable	Can grow to petabytes of data by adding more nodes
High Throughput	Optimized for batch processing of large datasets
Write Once, Read Many	Designed for append-only access (no random writes)
Data Locality	Moves computation close to where data is stored

## ⚠️ What Happens if NameNode Fails?

- ✗ HDFS cannot function without the NameNode (metadata is lost)
- ✓ That's why checkpointing via **Secondary NameNode** is vital
- ✓ In modern Hadoop, **High Availability (HA)** with multiple NameNodes is used



## Racks

### What is a Rack (in Distributed Systems)?

In the context of **Big Data** and **distributed computing**, a **rack** refers to a **physical collection of servers (nodes)** that are placed together in the same **hardware cabinet** within a **data center**. It's an important concept for understanding **data placement, fault tolerance, and network topology** in systems like **Hadoop (HDFS)**.

A **rack** is a set of nodes (machines) that are **physically connected through the same rack switch**, which in turn connects to a **higher-level network switch** that communicates with other racks.

\*\*\*\* Replications across different racks

## Why Racks Matter in HDFS

HDFS is **rack-aware**. It knows where each node is physically located in the data center hierarchy.

### Example:

Assume each rack holds 20 machines.

[Rack 1] — Nodes 1 to 20

[Rack 2] — Nodes 21 to 40

When storing a file with **replication factor = 3**, HDFS:

1. Stores Block 1 on a node in **Rack 1**
2. Stores Block 2 on another node in **Rack 1**
3. Stores Block 3 on a node in **Rack 2**

This way:

- Two replicas are **close together** (same rack) for **fast access**
- One replica is **far away** (different rack) for **fault tolerance**

## HDFS Rack-Aware Block Placement Policy

### Without rack awareness:

- All three replicas might be on the same rack
  - ➡ risk of total data loss if that rack fails.

### With rack awareness:

- Ensures **replica distribution across racks**

Block	Replica 1 (default)	Replica 2	Replica 3
Block A	Node A (Rack 1)	Node B (Rack 1)	Node C (Rack 2)

## Relevance in Big Data Systems

Concept	Importance
<b>Fault tolerance</b>	Survive rack-level failures
<b>Network efficiency</b>	Local rack traffic is faster and cheaper
<b>Scalability</b>	Data placement can scale with data center growth

## NameNode vs DataNode

Feature	NameNode	DataNode
<b>Role</b>	Master	Worker
<b>Stores</b>	Metadata (file names, block IDs, block locations)	Actual data blocks
<b>Type of Node</b>	Master node in HDFS architecture	Slave/worker node
<b>Data Stored</b>	File system tree, namespace, block mappings	Real file contents in block format
<b>Frequency of Updates</b>	Updated when files are added, deleted, or changed	Sends periodic block reports and heartbeats
<b>Interaction With</b>	Clients, DataNodes	NameNode
<b>Memory Use</b>	High (keeps all metadata in RAM)	Moderate (stores block files on local disk)
<b>Failure Impact</b>	Critical: HDFS cannot function without it	Tolerable: HDFS re-replicates data from other nodes
<b>Backups</b>	Requires Secondary NameNode or HA setup	No backups needed; relies on replication
<b>Runs Daemon</b>	NameNode daemon	DataNode daemon

### NameNode in Detail

- Stores the **HDFS namespace**: directory structure, permissions, etc.
  - Maps **files → blocks → DataNodes**
  - Does **not store actual data**, only metadata
  - Handles
    - File creation
    - File deletion
    - Block placement decisions
- If NameNode fails (without HA), the whole system becomes **unusable**.

## DataNode in Detail

- Actually **stores data blocks** on local disks
- Sends **heartbeats** every 3 seconds to the NameNode to signal it's alive
- Sends **block reports** regularly with info on what blocks it stores
- Can serve **read/write requests** directly to/from the client

 If one DataNode fails, **replicated data** is used from other nodes.

## Temporary vs Permanent Node Failure in Distributed Systems

Aspect	Temporary Node Failure	Permanent Node Failure
 <b>Definition</b>	Node goes offline briefly (e.g., network glitch, reboot)	Node is lost permanently (e.g., hardware failure, decommissioned)
 <b>Duration</b>	Short-term (minutes or hours)	Long-term or forever (above 10.5 Mins)
 <b>Data Availability</b>	Data still accessible via other replicas	Data served from replicas; system re-replicates the missing copy [Completely Reset the data and create missing replicas in other nodes maintaining the replication factor]
 <b>Handled by</b>	HDFS detects via missed heartbeats and marks as "dead"	HDFS considers it lost and triggers <b>replication restoration</b>
 <b>Recovery Action</b>	No manual intervention needed; node rejoins automatically	Requires <b>manual replacement</b> or <b>cluster rebalancing</b>
 <b>Replication Impact</b>	No block replication unless node is down too long	Replication happens immediately to meet the configured factor
 <b>Impact on Performance</b>	Minimal (unless many nodes fail at once)	Can cause temporary storage or compute resource shortage
 <b>Cluster State</b>	Remains healthy; marked as "temporarily dead"	Node removed from cluster metadata
 <b>Risk Level</b>	Low (transient)	High (may lead to data loss if not handled properly)

## HDFS Configuration Property Changes and Their Effects

Property	Change	Positive Effects <input checked="" type="checkbox"/>	Negative Effects <input type="checkbox"/>
<b>dfs.blocksize</b> <b>(default: 128 MB)</b>	▲ Increase (e.g., 128MB → 256MB)	- Fewer blocks = less NameNode memory usage- Better performance for large files- Reduced disk seeks	- Wastes space for small files- Reduces parallelism in MapReduce
	▼ Decrease (e.g., 128MB → 64MB)	- Better for small files- More parallelism	- Higher NameNode memory usage- More I/O overhead
<b>dfs.replication</b> <b>(default: 3)</b>	▲ Increase (e.g., 3 → 5)	- Higher fault tolerance- Better read availability	- More disk usage- More network traffic
	▼ Decrease (e.g., 3 → 2)	- Saves storage space	- Less fault tolerance- Reduced read performance
<b>dfs.heartbeat.interval</b> <b>(default: 3s)</b>	▼ Decrease	- Faster DataNode failure detection	- Slight increase in network traffic
	▲ Increase	- Reduced network load	- Slower failure detection
<b>dfs.namenode.handler.count</b>	▲ Increase	- Handles more client requests simultaneously	- Higher CPU/memory usage
<b>dfs.datanode.max.transfer.threads</b>	▲ Increase	- Better data transfer throughput	- More resource consumption on DataNode

## NameNode Failure in HDFS

The **NameNode** is the **central master node** in HDFS (Hadoop Distributed File System). It is responsible for managing **metadata**—that is, information **about** the files and blocks stored in the system (but not the actual data).

💡 Because the NameNode is critical to the operation of HDFS, its failure has **severe consequences**.

### What Happens If the NameNode Fails?

#### Impact of NameNode Failure

Consequence	Description
🚫 HDFS becomes inaccessible	No read/write operations can be performed
✗ Cluster halts	All jobs fail or are unable to proceed
⬇️ No access to metadata	File-to-block mapping and block locations are lost temporarily
⚠️ Not a data loss event	Actual data blocks still exist on DataNodes, but no map to access them

Think of it as having all your documents (data) in drawers (DataNodes), but losing the **index or table of contents** (NameNode).

### Why the NameNode Is a Single Point of Failure (SPOF)

Before Hadoop 2.x, **HDFS had only one NameNode**. If it crashed:

- You had to manually restore from the last saved **fsimage** and **edit logs**
- System downtime could be **hours or days**

#### Solutions for NameNode Failure

##### ✓ 1. Secondary NameNode (Misleading Name!)

- **Not a real-time backup**
- Periodically merges the **fsimage** and **edit logs** from the active NameNode
- Creates **checkpoints**
- Helps in **manual recovery** but cannot take over automatically

##### ✓ 2. Backup Node (Hadoop 1.x)

- Maintains an in-memory copy of metadata
- Receives real-time updates from NameNode
- Can **minimize downtime** during failover, but still not fully automatic

### 3. High Availability (HA) – Hadoop 2.x and beyond

#### HA Mode introduces two NameNodes:

- One **Active** (handling requests)
- One **Standby** (ready to take over immediately)

Component	Role
Active NameNode	Handles file operations and metadata
Standby NameNode	Maintains a synced state and takes over during failure

- Uses a **shared storage** or **JournalNodes** for metadata logs
- Automatic failover with **Zookeeper**

#### Recovery Steps Without HA (Legacy)

1. Restart NameNode manually
2. Load last fsimage (file system snapshot)
3. Apply recent changes from edit logs
4. Reconnect clients

Downtime: Significant

Risk: High if fsimage or logs are corrupted

#### Best Practices to Prevent Disaster

Practice	Purpose
Enable HA Mode	Eliminate SPOF
Use Zookeeper + JournalNodes	Ensure automatic failover
Regular checkpoints	For quick recovery from crash
Monitor with alerts/logs	Detect early signs of NameNode issues
Backup fsimage and logs	Prepare for worst-case recovery

## NameNode vs Secondary NameNode vs Standby NameNode

Feature	NameNode	Secondary NameNode	Standby NameNode
<b>Role</b>	Master node; manages metadata & client requests	Checkpoints metadata; merges edits with fsimage	Replica node; takes over if Active NameNode fails
<b>Workflow</b>	Handles all file system operations	Periodically pulls fsimage and edits to create checkpoint	Syncs state from Active via shared storage or JournalNodes
<b>Hot Backup</b>	No backup role	No live failover or backup capability	Yes; hot standby that can auto-failover
<b>Handles Client Requests?</b>	Yes	No	(Only after failover)
<b>Includes</b>	Namespace tree, block mapping, in-memory metadata	Fetches fsimage and edits from NameNode	Full metadata replication, ZKFC for failover
<b>Cost</b>	Medium (1 master node + memory intensive)	Low (used only for checkpointing)	High (requires JournalNodes + HA setup)
<b>Sync Frequency</b>	Continuous (live metadata management)	Periodic (manual or scheduled)	Near-real-time sync via shared edit logs
<b>Failure Recovery</b>	System halts on failure (SPOF)	Helps manual recovery (no automatic takeover)	Automatically takes over during failure
<b>Dependency</b>	Needs DataNodes and stable disk/memory	Needs only access to NameNode's logs & fsimage	Depends on shared edit logs + Zookeeper
<b>Best Use Case</b>	Default single-master deployments	Legacy/low-risk clusters	High-availability critical environments

### Quick Definitions

Term	Description
<b>fsimage</b>	Snapshot of the file system metadata
<b>edit logs</b>	Log of recent changes (appended until merged with fsimage)
<b>Checkpointing</b>	Combining fsimage + edit logs into a new fsimage
<b>ZKFC</b>	Zookeeper Failover Controller (handles automatic failover)
<b>JournalNode</b>	Shared log store for HA (used by both Active and Standby NameNodes)

## Summary

- **NameNode** = Brain of HDFS; critical, handles all operations.
- **Secondary NameNode** = Misleading name! It does **not** provide failover.
- **Standby NameNode** = Part of **HA setup**; provides true **hot failover** and redundancy.

## HDFS High Availability (HA) Architecture

In a traditional HDFS setup, the **NameNode is a single point of failure** (SPOF). If the **NameNode fails**, the entire HDFS cluster becomes unavailable—even though the data on **DataNodes** is intact.

To solve this, **HDFS High Availability (HA)** architecture was introduced in **Hadoop 2.x**.

### What is HDFS HA?

HDFS High Availability allows an HDFS cluster to **run with two NameNodes**:

- One **Active**
- One **Standby**

They **share the same metadata** using a **shared storage mechanism** (like JournalNodes), ensuring that the **Standby can take over automatically** if the Active fails.

## HDFS HA Architecture Components

Component	Role
Active NameNode	Handles all client requests (read/write, metadata)
Standby NameNode	Maintains an up-to-date copy of the metadata from shared logs
JournalNodes (JNs)	A quorum of services that store edit logs reliably
Zookeeper	Coordinates failover and leader election
Zookeeper Failover Controller (ZKFC)	Daemon on both NameNodes for health monitoring and automatic failover

## Workflow: Normal Operation

### 1. Active NameNode:

- Receives file system requests (e.g., create, delete files)
- Writes **edit logs** to a quorum of **JournalNodes (usually 3 or 5)**

### 2. JournalNodes:

- Store the edit logs
- Synchronize changes across all JournalNodes

### 3. Standby NameNode:

- Continuously reads edit logs from JournalNodes
- Applies the changes to its own namespace
- Stays in perfect sync with the Active

## What Happens on NameNode Failure?

## Automatic Failover Workflow

Step	Description
1	ZKFC detects that the Active NameNode is unresponsive using health checks
2	It sends a signal to Zookeeper to begin failover
3	Zookeeper performs leader election and promotes the Standby NameNode to Active
4	New Active NameNode continues from the last applied edit log (no data loss)
5	Clients are redirected to the new Active NameNode

 Failover happens in seconds – No manual intervention needed!

## Recovery After Failover

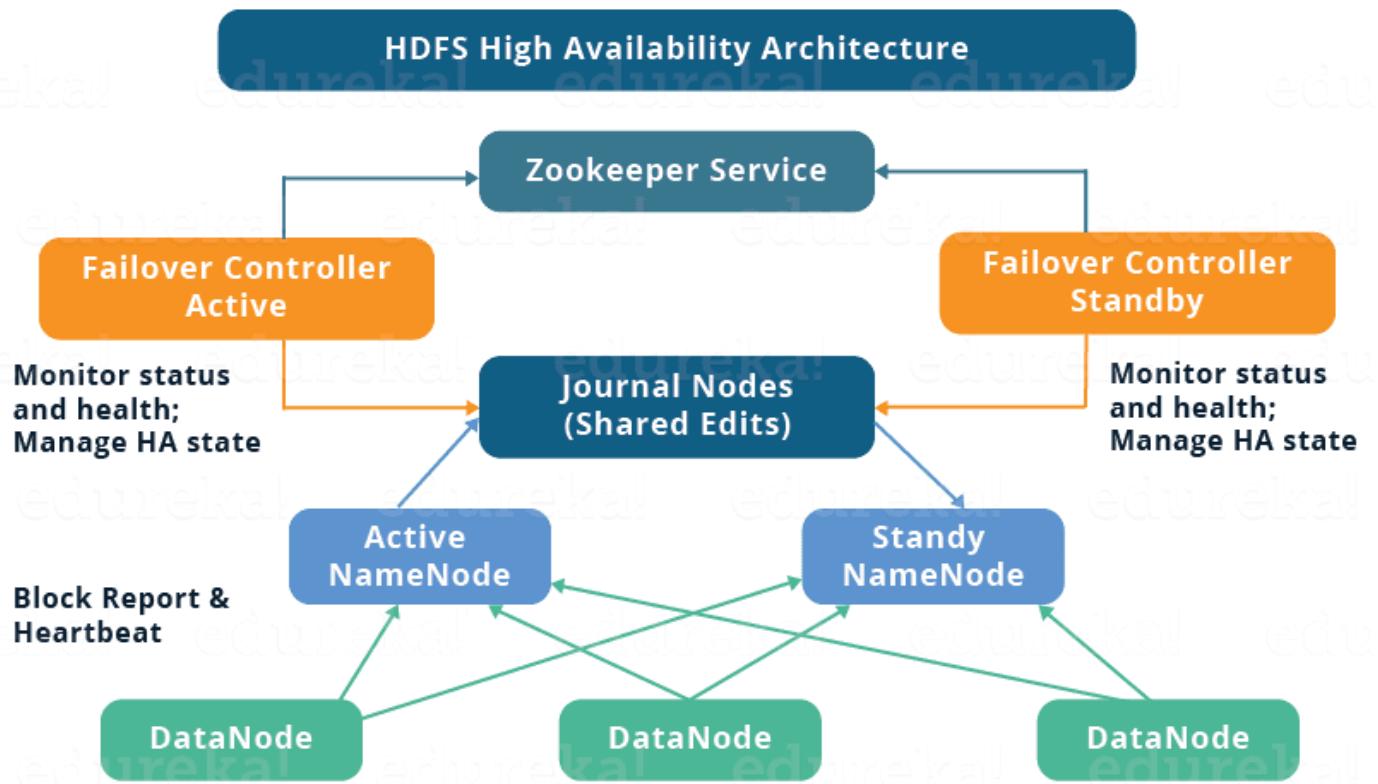
When the previously failed NameNode comes back online:

- It joins as Standby
- Starts syncing edit logs from JournalNodes again
- Can be promoted again if needed

## Key Benefits of HDFS HA

Benefit	Description
<input checked="" type="checkbox"/> No single point of failure	The cluster stays online even if one NameNode crashes
<input checked="" type="checkbox"/> Zero data loss	Edit logs are shared and applied in near-real-time
<input checked="" type="checkbox"/> Automatic failover	Zookeeper + ZKFC handle failure detection and recovery
<input checked="" type="checkbox"/> Production-ready	Reliable for enterprise-grade systems

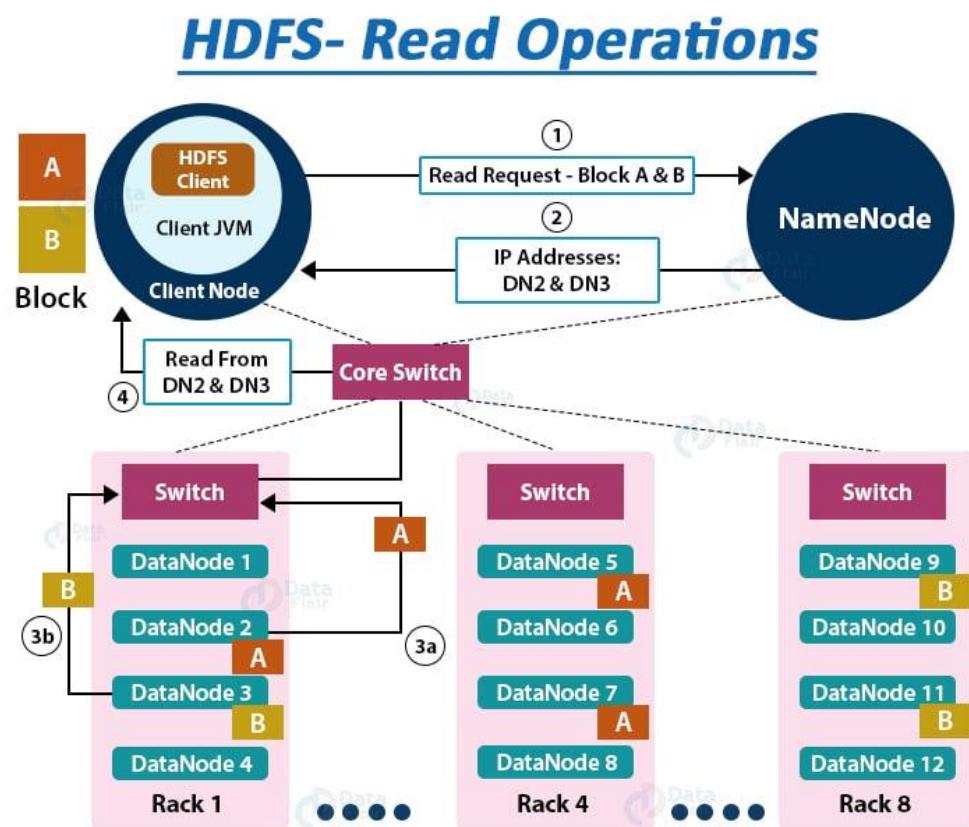
## HDFS HA Architecture Diagram



# Read and Write in HDFS

## HDFS Write Flow

When a client writes a file to HDFS, the process involves the **NameNode**, **DataNodes**, and internal replication.



## Step-by-Step Write Flow

Step	Action
1	Client contacts NameNode to request to write a new file.
2	NameNode checks metadata (filename, permissions, etc.) and breaks file into blocks (default: 128MB).
3	NameNode chooses DataNodes to store each block and its replicas (based on rack awareness and availability).
4	NameNode sends back block ID and target DataNodes (e.g., DN1 → DN2 → DN3).
5	Client sends data to the first DataNode (DN1). DN1 begins pipelining the data to DN2, which then sends it to DN3.
6	Each block is broken into packets. As packets flow down the pipeline, ACKs flow back to the client.
7	After all blocks are written and acknowledged, client notifies NameNode the file is closed.
✓	File is now successfully written and available in HDFS with all replicas stored.

## Fault Tolerance in HDFS (During Write)

HDFS is designed to **tolerate and recover from failures** in real-time.

### Key Mechanisms

Type of Failure	How HDFS Handles It
<b>DataNode failure</b>	Block is re-replicated to a new DataNode by NameNode
<b>Client crashes mid-write</b>	File is marked as incomplete and later removed by NameNode
<b>One replica fails during write</b>	Pipeline reconfigures with remaining DataNodes
<b>NameNode failure</b>	In HA mode, Standby NameNode takes over automatically
<b>Corrupt block detected</b>	Detected via checksum; replaced from another replica

## Important Concepts in Fault Tolerance

Concept	Description
<b>Replication Factor</b>	Default is 3; allows data to survive multiple node failures
<b>Checksums</b>	Each data block has checksums to detect corruption
<b>Heartbeats</b>	DataNodes send heartbeats to NameNode every 3s to show they're alive
<b>Block Reports</b>	DataNodes periodically report which blocks they have
<b>Rack Awareness</b>	Ensures replicas are placed across different racks for higher availability

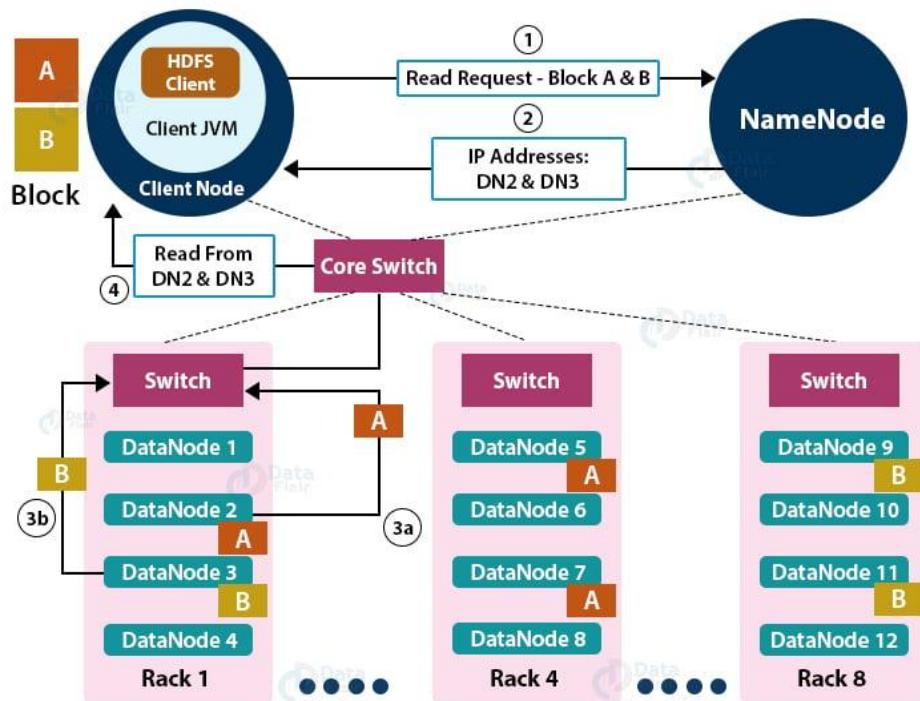
## Summary Table

Element	Purpose
<b>NameNode</b>	Assigns block locations; tracks metadata
<b>DataNode</b>	Stores actual data blocks; participates in pipeline
<b>Client</b>	Writes data block by block, receives ACKs
<b>Replication</b>	Maintains 3 copies by default across nodes/racks
<b>Pipelining</b>	Efficient parallel block writing and acknowledgment
<b>Fault Recovery</b>	Automatic re-replication and failover mechanisms

## HDFS Read Flow

When a client wants to read a file from HDFS, the following happens

### HDFS- Read Operations



#### Step-by-Step HDFS Read Process:

Step	Action
1	Client contacts the NameNode and requests metadata for the file (block locations).
2	NameNode responds with block IDs and DataNode addresses where each block (and its replicas) is stored.
3	The client chooses the nearest or least-loaded DataNode for each block.
4	Client connects directly to that DataNode and begins reading the block data.
5	If the file spans multiple blocks, the client continues to read block by block from their respective DataNodes.
<input checked="" type="checkbox"/>	Once all blocks are read, the client assembles the complete file locally.

## Fault Tolerance in HDFS Read

### What if a DataNode fails during read?

HDFS is highly fault-tolerant thanks to **replication** and **failover mechanisms**.

Scenario	What Happens
 <b>DataNode fails</b>	Client <b>automatically switches</b> to another replica (same block, different node)
 <b>Block is corrupted (bad checksum)</b>	Block is <b>re-read from a different replica</b> with valid checksum
 <b>Multiple failures (rare)</b>	As long as <b>one replica is alive</b> , the block is accessible
 <b>No available replica</b>	NameNode <b>marks file as corrupt</b> and raises an alert

### Key Mechanisms Ensuring Fault Tolerance:

Mechanism	Purpose
 <b>Replication</b>	Default: 3 copies of each block, on separate nodes (and racks)
 <b>Checksum verification</b>	Each block has a checksum; invalid data triggers re-read
 <b>Client failover logic</b>	Built-in logic tries next replica on failure
 <b>Heartbeat system</b>	NameNode constantly monitors DataNode health
 <b>Rack awareness</b>	Blocks are spread across racks to survive rack-level failures

### Summary Table – HDFS Read Flow vs Fault Handling

Component	Read Flow Role	Fault Tolerance Role
<b>Client</b>	Reads blocks from chosen DataNodes	Switches to another replica on failure
<b>NameNode</b>	Sends metadata & block locations	Removes failed DataNodes from responses
<b>DataNode</b>	Serves actual block data	If fails, another node serves the same block
<b>Replication</b>	Provides multiple copies	Ensures availability even if nodes go down
<b>Checksum</b>	Verifies block integrity	Detects corruption and retries automatically

 **Simple Analogy:**

Reading from HDFS is like having 3 photocopies of a book page in different drawers.

If one drawer is jammed (failed node), you grab a copy from the next one.

# Linux Commands

## File Management Commands

Command	Explanation	I/O
<b>ls</b>	Lists files and directories	Output: File list
<b>cd [dir]</b>	Changes current directory	Input: Directory name
<b>pwd</b>	Prints current working directory	Output: Absolute path
<b>mkdir [dir]</b>	Creates a new directory	Input: Directory name
<b>rmdir [dir]</b>	Deletes an empty directory	Input: Directory name
<b>rm [file]</b>	Removes a file	Input: File name
<b>rm -r [dir]</b>	Recursively removes directory	Input: Directory name
<b>rm -f [file]</b>	Force delete file (no prompt)	Input: File name
<b>mv [src] [dest]</b>	Moves or renames files	Input: Source and destination paths
<b>cp [src] [dest]</b>	Copies a file	Input: Source and destination paths
<b>cp -r [src] [dest]</b>	Recursively copies directories	Input: Directory paths
<b>touch [file]</b>	Creates an empty file or updates timestamp	Input: File name
<b>cat [file]</b>	Displays file contents	Output: File content
<b>more, less</b>	Views large file content one page at a time	Input: File name
<b>head, tail</b>	Displays first/last N lines of file	Output: Partial content
<b>stat [file]</b>	Displays file metadata	Output: Size, permissions, timestamps
<b>file [file]</b>	Determines file type	Output: Description of content
<b>find . -name "*.txt"</b>	Searches for files by name	Output: Matching files
<b>basename [path]</b>	Extracts filename from path	Output: File name only
<b>dirname [path]</b>	Extracts directory from path	Output: Directory only
<b>diff [file1] [file2]</b>	Compares two files line-by-line	Output: Differences
<b>cmp [file1] [file2]</b>	Binary comparison	Output: Byte difference
<b>du -sh [dir]</b>	Shows directory size summary	Output: Size in human-readable format
<b>df -h</b>	Shows disk space usage	Output: Filesystem capacity
<b>ln -s [target] [link]</b>	Creates symbolic (soft) link	Input: Target and link path
<b>ln [target] [link]</b>	Creates hard link	Input: Files must be on same filesystem

## File Types

Type Symbol	Description
-	Regular file
d	Directory
l	Symbolic link
b	Block device
c	Character device
s	Socket
p	Named pipe (FIFO)

## Linux File Permissions Commands Table

Command	Explanation	I/O
<code>ls -l</code>	Lists files with permissions	Output: -rwxr-xr-- style
<code>chmod [mode] [file]</code>	Changes file permissions	Input: Mode (e.g., 755, u+x)
<code>chown [owner] [file]</code>	Changes file owner	Input: New owner; affects ownership
<code>chown [owner]:[group] [file]</code>	Changes owner and group	Input: User and group
<code>chgrp [group] [file]</code>	Changes group ownership only	Input: Group name
<code>umask</code>	Shows or sets default permission mask	Input/Output: Octal mask (e.g., 0022)
<code>stat [file]</code>	Displays detailed file metadata	Output: Permissions, owner, timestamps
<code>getfacl [file]</code>	Shows ACL (Access Control List)	Output: Extended permissions
<code>setfacl -m u:username:rw file</code>	Sets ACL for a user	Input: User and permission
<code>find . -type f -perm 777</code>	Finds files with specific permissions	Output: Matching files
<code>chmod +x script.sh</code>	Adds execute permission to a file	Input: Mode; Output: Permission changed
<code>chmod -R 755 directory/</code>	Recursively changes permissions	Input: Mode and path

## Permission Bits Breakdown

Symbol	Meaning
r	Read (4)
w	Write (2)
x	Execute (1)
-	No permission

## File Mode Format Example

-rwxr-xr--

Entity	Meaning
-	File type (- = regular file, d = directory)
rwx	Owner: read, write, execute (7)
r-x	Group: read, execute (5)
r--	Others: read only (4)

## Numeric (Octal) Permissions

Octal	Permission	Symbolic
7	Read + Write + Execute	rwx
6	Read + Write	rw-
5	Read + Execute	r-x
4	Read only	r--
0	No permission	---

E.g., chmod 755 file = rwxr-xr-x

## File Renaming in Linux

Command / Tool	Explanation	I/O
<b>mv oldname newname</b>	Renames a single file or directory	Input: Old name, new name
<b>rename 's/old/new/' *.txt</b>	Batch rename using Perl-style regex	Input: Pattern & files; Output: Renamed files
<b>rename -n 's/\.jpeg\$/\.jpg/' *.jpeg</b>	Dry run to preview changes	Input: Pattern match
<b>mmv '*.txt' '#1.md'</b>	Mass renaming using pattern matching	Input: Wildcards; Output: Renamed set
<b>for f in *.txt; do mv "\$f" "\${f%.txt}.md"; done</b>	Bash loop to rename all .txt to .md	Input: Shell script
<b>find . -name "*.JPG" -exec rename 's/\.JPG\$/\.jpg/' {} +</b>	Recursive renaming	Input: find + rename

# Linux Networking Commands Table

Command	Explanation	I/O
<b>ifconfig</b>	Displays or configures network interfaces (legacy)	Output: IPs, MAC, interfaces
<b>ip addr</b>	Shows IP addresses (modern replacement for ifconfig)	Output: Interface config
<b>ip link</b>	Views and manages network links	Input/Output: Interface status
<b>ip route</b>	Displays routing table	Output: Routes and gateways
<b>ping [host]</b>	Tests connectivity to a host	Input: Host/IP; Output: Response time
<b>traceroute [host]</b>	Shows route taken to reach a host	Input: Host/IP; Output: Hops
<b>nslookup [domain]</b>	Queries DNS records for a domain	Input: Domain name; Output: IP info
<b>dig [domain]</b>	Detailed DNS lookup	Input: Domain; Output: DNS records
<b>host [domain]</b>	Simple DNS lookup	Input: Domain; Output: IPs
<b>netstat -tuln</b>	Shows active ports and listening services	Output: Network socket status
<b>ss -tuln</b>	Modern netstat replacement (faster, more detailed)	Output: Open ports, services
<b>nmap [host]</b>	Port scanner and network mapper	Input: Target IP; Output: Open ports/services
<b>telnet [host] [port]</b>	Tests if a port is open on a host	Input: Host + port; Output: Connection test
<b>nc [host] [port] (Netcat)</b>	General-purpose network tool for port testing/data transfer	Input: Host, port, message
<b>curl [url]</b>	Sends HTTP requests	Input: URL; Output: HTML or data
<b>wget [url]</b>	Downloads content from URL	Input: URL; Output: Saved file
<b>scp source target</b>	Securely copy files between systems	Input: Path; Output: File copied
<b>ssh user@host</b>	Connects to remote machine over SSH	Input: Hostname/IP; Output: Shell access
<b>ftp, sftp</b>	Transfers files via FTP/SFTP	I/O: Credentials and file transfer
<b>iptables</b>	Configures firewall rules	Input: Rules; Output: Traffic filtering
<b>ufw</b>	Simplified firewall tool (for Ubuntu)	Input: Allow/deny commands
<b>nmcli</b>	NetworkManager CLI to manage connections	Input: Interface config
<b>hostname</b>	Shows or sets system's hostname	I/O: Hostname
<b>arp -a</b>	Displays ARP cache	Output: IP-to-MAC mappings
<b>route</b>	Displays/modifies IP routing table (deprecated)	Output: Routes
<b>tcpdump</b>	Captures and analyzes network traffic	Output: Packet details
<b>netcat or nc</b>	Can open TCP/UDP connections, scan ports, transfer files	Input: IP/port
<b>whois</b>	Fetches domain registration info	Input: Domain; Output: Registry data

# Mastering HDFS Commands

## Setting Up Hadoop for HDFS

### What Is HDFS Setup and Why Do We Need It?

HDFS stands for **Hadoop Distributed File System**—it's like a big storage system spread across many computers.

Before we can use HDFS to store or read files, we need to:

- Install Java (because Hadoop runs on Java)
- Set up Hadoop
- Configure a few XML files
- Start HDFS services (like the NameNode and DataNode)

This chapter walks you through each step so you can run Hadoop on your own computer in **pseudo-distributed mode** (i.e., a single machine mimicking a cluster).

### Installing Prerequisites

#### Step 1: Install Java

Hadoop is built in Java, so it **requires Java to run**.

Open terminal and run:

```
sudo apt update  
sudo apt install openjdk-11-jdk  
Check if Java is installed:  
java -version
```

**Expected output:**

```
openjdk version "11.0.x"
```

#### Step 2: Create a Separate Hadoop User (Optional but Good Practice)

Creating a separate user for Hadoop improves security and organization.

```
sudo adduser hadoop  
sudo usermod -aG sudo hadoop  
su - hadoop
```

#### Step 3: Set Up Passwordless SSH

SSH is needed because Hadoop uses it internally to manage its own processes—even on the same machine.

Generate and authorize SSH key:

```
ssh-keygen -t rsa  
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

Now test:

```
ssh localhost  
If it logs in without asking for a password, it worked!
```

## Download and Configure Hadoop

### Step 4: Download and Extract Hadoop

Use wget to get the latest Hadoop release (change the version if needed)

```
wget https://downloads.apache.org/hadoop/common/hadoop-3.3.6/hadoop-3.3.6.tar.gz
```

Extract the file and rename it

```
tar -xzvf hadoop-3.3.6.tar.gz  
mv hadoop-3.3.6 ~/hadoop
```

## Configure Hadoop Environment

### Step 5: Set Environment Variables

These settings help your system know where Hadoop is installed.

Edit your .bashrc file:

```
nano ~/.bashrc
```

Add these lines at the bottom:

```
export HADOOP_HOME=$HOME/hadoop  
export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64  
export PATH=$PATH:$HADOOP_HOME/bin:$HADOOP_HOME/sbin
```

Then apply it:

```
source ~/.bashrc  
Now hadoop version should show output.
```

## Configure Hadoop XML Files

Navigate to config directory:

```
cd $HADOOP_HOME/etc/hadoop
```

### Configure core-site.xml

This tells Hadoop where the NameNode (master) is running.

```
<configuration>
  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://localhost:9000</value>
  </property>
</configuration>
```

### Configure hdfs-site.xml

This file manages how the file system behaves.

```
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>1</value> <!-- 1 means only one copy of each file -->
  </property>

  <property>
    <name>dfs.namenode.name.dir</name>
    <value>file:///home/hadoop/hdfs/namenode</value>
  </property>

  <property>
    <name>dfs.datanode.data.dir</name>
    <value>file:///home/hadoop/hdfs/datanode</value>
  </property>
</configuration>
```

Create these directories manually:

```
mkdir -p ~/hdfs/namenode
mkdir -p ~/hdfs/datanode
```

## **Format the NameNode (First Time Only)**

This step initializes the HDFS file system metadata.

```
hdfs namenode -format
```

### **Expected output:**

Storage directory /home/hadoop/hdfs/namenode has been successfully formatted.

## **Starting and Stopping HDFS**

### **Start HDFS**

```
start-dfs.sh
```

This launches:

- NameNode (master that stores file metadata)
- DataNode (worker that stores the actual data)

Verify with:

```
jps
```

### **Expected processes:**

```
NameNode  
DataNode  
SecondaryNameNode
```

### **Stop HDFS:**

```
stop-dfs.sh
```

## Using the HDFS Shell

Once services are running, you can interact with HDFS using hdfs dfs:

### Example Commands:

```
hdfs dfs -mkdir /user  
hdfs dfs -mkdir /user/hadoop  
hdfs dfs -put myfile.txt /user/hadoop/  
hdfs dfs -ls /user/hadoop  
hdfs dfs -cat /user/hadoop/myfile.txt
```

### Output Explanation

- -mkdir → creates directories in HDFS
- -put → uploads local file to HDFS
- -ls → lists files/directories in HDFS
- -cat → displays file contents

## Basic HDFS Commands

### What Are HDFS Shell Commands?

HDFS provides two main shell commands:

- hdfs dfs (recommended for Hadoop 2.x and above)
- hadoop fs (older, still supported)

 Both are functionally the same, but hdfs dfs is more specific to the HDFS file system.

### General Syntax

```
hdfs dfs -<command> <arguments>
```

OR

 **Tip:** Always use hdfs dfs unless working with other file systems (like S3 or local FS).

## -ls: List Files and Directories

### What It Does:

Lists the contents of a directory in HDFS (like ls in Linux).

### Syntax:

```
hdfs dfs -ls <HDFS-path>
```

### Example:

```
hdfs dfs -ls /user/hadoop
```

### Output:

Found 1 items

```
-rw-r--r-- 1 hadoop supergroup 2048 2025-07-02 /user/hadoop/file.txt
```

- -rw-r--r-- → File permissions: owner can read/write, group and others can read.
- 1 → Number of replicas (usually 1 in HDFS listing).
- hadoop → File owner.
- supergroup → Group owner.
- 2048 → File size in bytes.
- 2025-07-02 → Last modification date.
- /user/hadoop/file.txt → File path in HDFS.

## -mkdir: Create Directories

### What It Does:

Creates new directories in HDFS.

### Syntax:

```
hdfs dfs -mkdir <HDFS-path>
```

### Example

```
hdfs dfs -mkdir /user/hadoop/input
```

### Notes:

- Use -p to create parent directories if they don't exist

```
hdfs dfs -mkdir -p /user/hadoop/data/logs
```

## -put: Upload Files to HDFS

### What It Does:

Sends a file from the **local file system** to HDFS.

### Syntax:

```
hdfs dfs -put <local-path> <HDFS-path>
```

### Example:

```
hdfs dfs -put sample.txt /user/hadoop/
```

It uploads the file sample.txt from your local machine to the Hadoop Distributed File System (HDFS).

### Tip:

- Use this to move data from your PC to the Hadoop cluster.

## -get: Download Files from HDFS

### What It Does

Fetches a file from **HDFS** to the **local machine**.

### Syntax:

```
hdfs dfs -get <HDFS-path> <local-destination>
```

### Example:

```
hdfs dfs -get /user/hadoop/sample.txt ~/Downloads/
```

It **downloads** the file sample.txt from the **HDFS path** /user/hadoop/ to your **local Downloads folder** (~/Downloads/).

## -cat: View Contents of an HDFS File

### What It Does

Displays the contents of a file stored in HDFS (like cat in Linux).

### Syntax:

```
hdfs dfs -cat <HDFS-file>
```

### Example

```
hdfs dfs -cat /user/hadoop/sample.txt
```

It **displays the contents** of the file sample.txt located in **HDFS** directly in the **terminal**.

## -touchz: Create Empty Files

### What It Does

Creates a new **zero-byte file** in HDFS (useful for signaling or testing).

### Syntax

```
hdfs dfs -touchz <HDFS-file>
```

### Example:

```
hdfs dfs -touchz /user/hadoop/ready.txt
```

It creates an empty file named ready.txt at the specified HDFS path /user/hadoop/.

### Use Case:

- Used as a flag or trigger file in data pipelines.

## -rm: Delete Files or Directories

### What It Does:

Removes files or folders from HDFS.

### Syntax:

```
hdfs dfs -rm <HDFS-file>
```

### Example:

```
hdfs dfs -rm /user/hadoop/sample.txt
```

### To delete a directory recursively:

```
hdfs dfs -rm -r /user/hadoop/old_data/
```

## Summary Table

Command	Description	Example Command
<b>-ls</b>	List files/directories in HDFS	hdfs dfs -ls /
<b>-mkdir</b>	Make new HDFS directory	hdfs dfs -mkdir /data
<b>-put</b>	Upload file to HDFS	hdfs dfs -put file.txt /data/
<b>-get</b>	Download file from HDFS	hdfs dfs -get /data/file.txt .
<b>-cat</b>	Show file contents	hdfs dfs -cat /data/file.txt
<b>-touchz</b>	Create an empty file	hdfs dfs -touchz /done.txt
<b>-rm</b>	Delete file or folder	hdfs dfs -rm -r /old/

# Intermediate File Operations in HDFS

## -cp: Copy Files within HDFS

### What It Does:

Copies files from one HDFS location to another. It works like the Linux cp command, **but within HDFS only.**

### Syntax:

```
hdfs dfs -cp <source-path> <destination-path>
```

### Example:

```
hdfs dfs -cp /user/hadoop/file.txt /user/hadoop/backup/
```

It copies the file file.txt from its current HDFS location /user/hadoop/ to the HDFS directory /user/hadoop/backup/.

 Note: The destination can be a file or directory.

## -mv: Move or Rename Files

### What It Does:

Moves or renames files in HDFS. Similar to mv in Linux.

### Syntax:

```
hdfs dfs -mv <source-path> <destination-path>
```

### Example (move to folder):

```
hdfs dfs -mv /user/hadoop/file.txt /user/hadoop/archive/
```

### Example (rename file):

```
hdfs dfs -mv /user/hadoop/file.txt /user/hadoop/file_backup.txt
```

It **moves or renames** the file file.txt in HDFS to a new location or name file\_backup.txt within the same directory.

## -du: Show Disk Usage of Files/Directories

### What It Does

Shows how much space each file or directory occupies in HDFS.

### Syntax

```
hdfs dfs -du <path>
```

### Example

```
hdfs dfs -du /user/hadoop/
```

### Output:

2048 /user/hadoop/file.txt → file.txt is 2048 bytes

## -dus: Show Summary Disk Usage

### What It Does:

Displays the total size of all files in a given directory, like a summary.

### Syntax:

```
hdfs dfs -dus <path>
```

### Example:

```
hdfs dfs -dus /user/hadoop/
```

### Output:

```
6144 /user/hadoop/
```

 Note: This is useful when you just want the total size, not a breakdown per file.

## -stat: Show File Status Information

### What It Does:

Prints detailed info (like size, modification time, permission) of a file.

### Syntax:

```
hdfs dfs -stat <format> <path>
```

### Example:

```
hdfs dfs -stat "%n %b %y" /user/hadoop/file.txt
```

### Format Codes:

- %n: File name
- %b: File size in bytes
- %y: Modification date

 Note: Good for scripting and automation.

## -tail: Show Last Part of a File

### What It Does:

Displays the **last 1 KB** of a file's contents—like Linux tail.

### Syntax:

```
hdfs dfs -tail <file-path>
```

### Example:

```
hdfs dfs -tail /user/hadoop/logs/app.log
```

 Use Case: Checking the latest log entries without reading the full file.

**It displays the last part (usually last 1 KB)** of the file app.log stored in HDFS.

## -text: Convert File to Readable Format

### What It Does:

Reads a file and outputs it in **human-readable** format. Useful for compressed or sequence files.

### Syntax:

```
hdfs dfs -text <file-path>
```

### Example:

```
hdfs dfs -text /user/hadoop/data.seq
```

 Note: Use this for SequenceFiles or Avro/Parquet formats if they're readable.

### Summary Table

Command	Description	Example
<b>-cp</b>	Copy files/folders in HDFS	-cp /source /dest
<b>-mv</b>	Move or rename files	-mv /a /b
<b>-du</b>	Show file/folder size	-du /folder/
<b>-dus</b>	Total size summary of a directory	-dus /data/
<b>-stat</b>	Show file details (size, time)	-stat "%n %b %y" /file
<b>-tail</b>	Show end of file (last 1KB)	-tail /log.txt
<b>-text</b>	Read file in human-readable format	-text /compressed.seq

# Directory Management in HDFS

## Recursive Directory Operations

### 🧠 What It Means

Sometimes you need to create or delete **multiple levels of directories at once**, not just one.

#### ✓ Create a directory (with parents if needed)

```
hdfs dfs -mkdir -p /user/hadoop/data/input/logs
```

- -p ensures that **parent directories** are created if they don't already exist.

#### ✓ Delete directory recursively

```
hdfs dfs -rm -r /user/hadoop/data
```

- -r flag allows you to **delete folders and all contents inside them**.

## -count: Count Files, Directories, and Space Used

### 🧠 What It Does:

Reports

- Number of directories
- Number of files
- Total space used (in bytes)

#### ✓ Syntax:

```
hdfs dfs -count <HDFS-path>
```

### 📌 Example:

```
hdfs dfs -count /user/hadoop/
```

### 📤 Output

```
5      12    1048576 /user/hadoop/
```

Column	Meaning
1st (DIR_COUNT)	Number of sub-directories
2nd (FILE_COUNT)	Number of files
3rd (SPACE_USED)	Space used (bytes)

## Quota Management

### What Are Quotas?

Quotas **limit how many files or how much space** a user or directory can use in HDFS. This helps prevent abuse or overuse of storage resources.

There are two types:

- **Namespace Quota:** Limits the number of files/directories.
- **Space Quota:** Limits the total storage used.

### Set Namespace Quota

```
hdfs dfsadmin -setQuota <number> <path>
```

 Example:

```
hdfs dfsadmin -setQuota 100 /user/hadoop
```

 Allows **only 100 files or directories** under /user/hadoop.

### Set Space Quota (in bytes):

```
hdfs dfsadmin -setSpaceQuota <number> <path>
```

 Example:

```
hdfs dfsadmin -setSpaceQuota 104857600 /user/hadoop
```

 Sets **100 MB** space limit.

### View Quotas:

```
hdfs dfs -count -q <HDFS-path>
```

 Example:

```
hdfs dfs -count -q /user/hadoop
```

### Remove Quotas:

```
hdfs dfsadmin -clrQuota /user/hadoop
```

```
hdfs dfsadmin -clrSpaceQuota /user/hadoop
```

## Setting Permissions on Directories

 HDFS uses a permission model like Linux:

- r (read), w (write), x (execute/search)

 Change Permissions

```
hdfs dfs -chmod <permissions> <path>
```

 Example

```
hdfs dfs -chmod 755 /user/hadoop/data
```

- 755 = Owner can read/write/execute; others can read and execute.

 Change Owner

```
hdfs dfs -chown <owner>[:group] <path>
```

 Example:

```
hdfs dfs -chown hadoop:supergroup /user/hadoop/data
```

 Change Group:

```
hdfs dfs -chgrp <group> <path>
```

 Example:

```
hdfs dfs -chgrp analytics /user/hadoop/data
```

 View Permissions:

```
hdfs dfs -ls /user/hadoop/
```

 Output:

```
drwxr-xr-x - hadoop supergroup 0 2025-07-02 /user/hadoop/data
```

Permission	Meaning
d	Directory
rwx	Owner permissions
r-x	Group permissions
r-x	Other users

## Summary Table

Task	Command Example
Create nested dirs	-mkdir -p /path/dir
Delete directory	-rm -r /path
Count files/dirs	-count /path
Set namespace quota	dfsadmin -setQuota 100 /path
Set space quota	dfsadmin -setSpaceQuota 1G /path
Clear quota	dfsadmin -clrQuota /path
Change permissions	-chmod 755 /path
Change owner	-chown user:group /path
Check permissions	-ls /path

# Permissions and Access Control in HDFS

## HDFS File Permission Model

HDFS uses a traditional **UNIX-style permission model** consisting of:

- **Owner** (user who owns the file/directory)
- **Group** (a group of users)
- **Others** (everyone else)

Each entity (owner, group, others) can be granted three types of permissions:

Symbol	Meaning	Description
r	Read	Read the file or list directory
w	Write	Modify file or create/delete contents
x	Execute	Run a file or enter a directory

**Example:**

```
-rw-r--r-- 1 hadoop supergroup 1234 2025-07-02 /user/hadoop/file.txt
```

This means:

- Owner (hadoop) can read/write
- Group and Others can only read

## -chmod: Change Permissions

### Syntax

```
hdfs dfs -chmod <mode> <path>
```

### Example

```
hdfs dfs -chmod 755 /user/hadoop/data
```

This gives

- Owner: read, write, execute (7)
- Group: read, execute (5)
- Others: read, execute (5)

### Symbolic Format

```
hdfs dfs -chmod g+w /user/hadoop/file.txt # Adds write permission to group
```

## -chown: Change Ownership

### Syntax:

```
hdfs dfs -chown <owner>[:group] <path>
```

### Example:

```
hdfs dfs -chown analytics:hadoop /user/analytics/data
```

- Changes file owner to analytics and group to hadoop

**Note:** Only superusers can change ownership.

## -chgrp: Change Group Ownership

### Syntax:

```
hdfs dfs -chgrp <group> <path>
```

### Example:

```
hdfs dfs -chgrp devops /user/hadoop/logs
```

You can only change group if you're part of that group (or are superuser).

## Working with Superusers in HDFS

### What Is a Superuser?

A superuser in HDFS

- Can **access any file**
- Can **change permissions**, ownership, and quotas
- Is configured in core-site.xml under the property hadoop.proxyuser.<username>

### Example Configuration:

```
<property>
  <name>hadoop.proxyuser.hdfs.groups</name>
  <value>*</value>
</property>
<property>
  <name>hadoop.proxyuser.hdfs.hosts</name>
  <value>*</value>
</property>
```

- This allows user hdfs to act as proxy (superuser) for any group or host.

### Superuser Command Example

```
sudo -u hdfs hdfs dfs -chown user1:usergroup /user/user1/data
```

## ACLs: Access Control Lists in HDFS

### Why Use ACLs?

Permissions in the rwx model are limited to:

- Owner
- Group
- Others

ACLs allow **more flexible** permission settings for **multiple users and groups**.

### Enable ACLs in Configuration (`hdfs-site.xml`):

```
<property>
  <name>dfs.namenode.acls.enabled</name>
  <value>true</value>
</property>
```

### Set ACLs:

```
hdfs dfs -setfacl -m user:alice:rw- /user/shared/file.txt
```

→ Grants user alice **read and write** access.

### View ACLs:

```
hdfs dfs -getfacl /user/shared/file.txt
```

### Remove ACL:

```
hdfs dfs -setfacl -b /user/shared/file.txt
```

→ Removes all ACL entries (back to normal permissions).

### Set ACLs Recursively:

```
hdfs dfs -setfacl -R -m user:bob:r-- /user/data
```

## Summary Table

Command	Purpose	Example
<b>-chmod</b>	Change file/directory permissions	<code>-chmod 755 /dir</code>
<b>-chown</b>	Change owner and/or group	<code>-chown user:group /file</code>
<b>-chgrp</b>	Change group only	<code>-chgrp dev /file</code>
<b>-setfacl</b>	Add custom user/group access	<code>-setfacl -m user:alice:rw- /file</code>
<b>-getfacl</b>	View ACLs	<code>-getfacl /file</code>
<b>-setfacl -b</b>	Remove all ACLs	<code>-setfacl -b /file</code>

## Advanced File Operations in HDFS

### **-appendToFile: Append to an Existing HDFS File**

#### What It Does

Adds new content from a **local file** to the **end of an existing HDFS file**.

#### Syntax

```
hdfs dfs -appendToFile <local-file> <HDFS-file>
```

#### Example

```
hdfs dfs -appendToFile log_update.txt /user/hadoop/logs.txt
```

#### Note

- The destination HDFS file **must already exist**.
- Appending works best with uncompressed text files.

### **-setrep: Set Replication Factor for Files**

#### What It Does

Changes the **number of replicas** (copies) stored across the HDFS cluster for fault tolerance.

#### Syntax

```
hdfs dfs -setrep [-R] <replication-factor> <path>
```

#### Example

```
hdfs dfs -setrep 3 /user/hadoop/data.txt
```

- -R applies it recursively to all files in a directory.

## Why change this?

- Increase for high availability.
- Decrease to save space.

## -getmerge: Merge Multiple Files into One (Local)

### What It Does:

Combines multiple HDFS files into **one file on the local machine**.

### Syntax

```
hdfs dfs -getmerge <HDFS-dir> <local-file>
```

### Example

```
hdfs dfs -getmerge /user/hadoop/output/ merged.txt
```

### Useful for

- Merging part files (part-0000x) from MapReduce output into one readable file.

## -copyToLocal and -copyFromLocal

### Copy From Local (like put)

```
hdfs dfs -copyFromLocal <local-path> <HDFS-path>
```

### Example

```
hdfs dfs -copyFromLocal file.csv /user/hadoop/input/
```

### Copy To Local (like get)

```
hdfs dfs -copyToLocal <HDFS-path> <local-path>
```

### Example

```
hdfs dfs -copyToLocal /user/hadoop/input/file.csv ~/Downloads/
```

 These work the same as -put and -get but can be more **reliable** when used inside **shell scripts**.

## Working with Compressed Files in HDFS

### What It Means

You can **store, read, or write** compressed files (e.g., .gz, .bz2, .zip) in HDFS. HDFS doesn't care about compression type—it stores binary files as-is.

### Upload a .gz file

```
hdfs dfs -put compressed_logs.gz /user/hadoop/logs/
```

### View contents (if readable format)

```
hdfs dfs -text /user/hadoop/logs/compressed_logs.gz
```

 Use -text to stream readable compressed formats like .gz, not .zip.

## Listing Files Using Wildcards (Globs / Regex)

### What Are Globals?

HDFS supports **wildcard patterns** to match multiple files (similar to shell globs).

### Common Patterns:

- \* — Matches any number of characters
- ? — Matches a single character
- [abc] — Matches a, b, or c
- {x,y} — Matches x or y

### Example: List all .txt files

```
hdfs dfs -ls /user/hadoop/*.txt
```

### Example: Match any file starting with log\_

```
hdfs dfs -ls /user/hadoop/log_*
```

### Example: Match log files from Jan or Feb

```
hdfs dfs -ls /user/hadoop/{jan,feb}_log.txt
```

## Summary Table

Command	Description	Example
<b>-appendToFile</b>	Add local content to end of HDFS file	-appendToFile local.txt /log.txt
<b>-setrep</b>	Set file replication factor	-setrep 2 /file.txt
<b>-getmerge</b>	Merge HDFS files to one local file	-getmerge /output/ merged.txt
<b>-copyFromLocal</b>	Copy local file to HDFS	-copyFromLocal file.txt /data/
<b>-copyToLocal</b>	Copy HDFS file to local	-copyToLocal /data/file.txt ./
<b>-text</b>	View readable compressed file	-text /logs.gz
<b>-ls with globs</b>	Match multiple files with patterns	-ls /logs/*.txt

## File System Metadata Operations in HDFS

Metadata is **data about data**. In HDFS, file system metadata includes information like file size, modification time, block placement, checksums, and storage policies. Understanding metadata helps ensure file integrity, optimize performance, and support backup/recovery.

### -checksum: Verifying File Integrity

#### What It Does

Displays a **checksum** for an HDFS file—used to **detect corruption** or verify integrity after transfers.

#### Syntax

```
hdfs dfs -checksum <file-path>
```

#### Example

```
hdfs dfs -checksum /user/hadoop/data.csv
```

#### Output

Checksum type: CRC32C

Bytes per CRC: 512

Checksum: 00000200000000000000000044b71b6c

 Use Case: Check if a file uploaded or downloaded matches the source.

## File Modification Time and Status (-stat)

### Syntax

```
hdfs dfs -stat "%n %b %y" <file-path>
```

### Example

```
hdfs dfs -stat "%n %b %y" /user/hadoop/data.csv
```

### Output

data.csv 1048576 2025-07-02 10:15:00

Format	Meaning
%n	File name
%b	File size (bytes)
%y	Last modification date

 You can automate log checks or validate that files were updated as expected.

## File Block Locations

HDFS breaks files into blocks (default 128MB) and distributes them across the cluster. You can check where each block is stored using **Hadoop API or CLI tools**.

### Option 1: Web UI

Go to:

```
http://<namenode-host>:9870
```

Then navigate:

Utilities → Browse the file system → Click file → Block Info

### Option 2: Java API (programmatic access)

```
FileStatus fileStatus = fs.getFileStatus(new Path("/user/data.txt"));
BlockLocation[] locations = fs.getFileBlockLocations(fileStatus, 0, fileStatus.getLen());
```

 This helps in **debugging data skew, locality issues**, or performance bottlenecks.

## File Locality & Storage Policies

### 🧠 What Is Locality?

Locality refers to whether a file's block is stored on the **same node where computation is running**. HDFS tries to schedule jobs where data is already present.

You can **set policies** to store files on specific storage types like:

- SSD (ALL\_SSD)
- DISK (HOT)
- ARCHIVE (COLD)

### ✓ Check Storage Policy:

```
hdfs storagepolicies -getStoragePolicy <path>
```

📌 Example:

```
hdfs storagepolicies -getStoragePolicy /user/hadoop/data.csv
```

### ✓ Set Storage Policy:

```
hdfs storagepolicies -setStoragePolicy -path <path> -policy <policy-name>
```

📌 Example:

```
hdfs storagepolicies -setStoragePolicy -path /user/hadoop/archive/ -policy COLD
```

## Snapshots and Rollbacks

### 🧠 What Are Snapshots?

HDFS snapshots are **read-only point-in-time copies** of directories. They are useful for **backup, recovery, and rollback**.

### ✓ Enable Snapshots

```
hdfs dfsadmin -allowSnapshot <dir>
```

📌 Example

```
hdfs dfsadmin -allowSnapshot /user/hadoop/projects
```

### ✓ Create Snapshot:

```
hdfs dfs -createSnapshot <dir> [snapshot-name]
```

📌 Example:

```
hdfs dfs -createSnapshot /user/hadoop/projects pre-migration
```

## List Snapshots

```
hdfs dfs -ls /user/hadoop/projects/.snapshot
```

## Rollback (Delete current state and restore snapshot):

```
hdfs dfs -renameSnapshot <dir> <snapshot-old> <snapshot-new>
```

Or manually:

```
hdfs dfs -deleteSnapshot <dir> <snapshot-name>
```

 Be careful: Snapshots are not version-controlled—they **capture current state** only.

## Summary Table

Task	Command / Tool	Example
<b>Check file checksum</b>	hdfs dfs -checksum	-checksum /file.txt
<b>View file modification</b>	hdfs dfs -stat	-stat "%n %y" /file
<b>View block location</b>	Web UI or API	NameNode UI
<b>Get storage policy</b>	-getStoragePolicy	-getStoragePolicy /file
<b>Set storage policy</b>	-setStoragePolicy -policy COLD	/archive/
<b>Enable snapshot</b>	dfsadmin -allowSnapshot <dir>	/projects/
<b>Create snapshot</b>	-createSnapshot <dir> snapshot1	/dir/
<b>Rollback or delete snap</b>	-deleteSnapshot <dir> snapshot1	N/A

# HDFS Data Recovery and Troubleshooting

HDFS is designed for fault tolerance, but administrators still need to know how to recover lost or corrupted data, manage logs, and handle common errors. This chapter covers everything you need to know to diagnose and recover from HDFS issues.

## Handling Corrupted Blocks

### What Are Corrupted Blocks?

A corrupted block occurs when HDFS cannot verify a block's checksum or fails to replicate it properly.

#### Detect Corruption:

```
hdfs fsck / -files -blocks -locations
```

 Output example:

```
/user/hadoop/data.csv: CORRUPT block blk_107865
```

#### Fix: Trigger Block Replication

If a block is missing or corrupt

- HDFS will try to auto-replicate from good copies.
- You can also trigger manual replication or copy the file again.

#### Remove Corrupt File:

If no good replica exists:

```
hdfs dfs -rm /user/hadoop/data.csv
```

*Tip:* Always keep backups via **snapshots** or external archives.

## Recovering Deleted Files with Trash

### What Is Trash in HDFS?

Trash is a "recycle bin" for deleted files. Instead of being deleted permanently, files are moved to:

```
/user/<username>/._Trash/
```

#### Enable Trash

Add to core-site.xml:

```
<property>
  <name>fs.trash.interval</name>
  <value>60</value> <!-- Minutes -->
</property>
```

 This retains deleted files for 60 minutes.

### **Restore File from Trash:**

```
hdfs dfs -mv /user/hadoop/.Trash/Current/path/to/file /user/hadoop/
```

### **Manually Move File to Trash:**

```
hdfs dfs -rm -skipTrash /file.txt # This bypasses trash (use with caution!)
```

## **Enable Snapshots for Recovery**

Snapshots can protect data beyond trash retention time.

### **Enable Snapshots:**

```
hdfs dfsadmin -allowSnapshot /user/hadoop/projects
```

### **Create Snapshot:**

```
hdfs dfs -createSnapshot /user/hadoop/projects backup1
```

### **Restore from Snapshot:**

Copy the snapshot file back:

```
hdfs dfs -cp /user/hadoop/projects/.snapshot/backup1/file.txt /user/hadoop/projects/
```

 Snapshots are versioned backups of entire directories.

## **Log Files and Audit Logs**

### **Where to Find HDFS Logs**

Logs are essential for debugging HDFS errors. You can find them in:

Component	Log Location
<b>NameNode</b>	\$HADOOP_HOME/logs/hadoop-* Namenode-* .log
<b>DataNode</b>	\$HADOOP_HOME/logs/hadoop-* Datanode-* .log
<b>JournalNode</b>	\$HADOOP_HOME/logs/hadoop-* Journalnode-* .log

Use

```
tail -f $HADOOP_HOME/logs/hadoop-hdfs-namenode-* .log
```

 Use grep to filter by ERROR, WARN, or specific file names.

## **Enable HDFS Audit Logs**

Audit logs track **who accessed what** and are useful for security and compliance.

Set in log4j.properties:

```
log4j.logger.org.apache.hadoop.hdfs.server.namenode.FSNamesystem.audit=INFO,DRFA
```

## Common HDFS Errors and Fixes

Error Message	Possible Cause	Fix
<b>Permission denied</b>	Insufficient permissions	Use -chmod, -chown, or ACLs
<b>No such file or directory</b>	Path typo or deleted file	Use -ls to verify path
<b>Cannot append to a compressed file</b>	HDFS doesn't support append for compressed files	Use a different format
<b>Corrupt block detected</b>	DataNode failure or bad disk	Use fsck, remove corrupt files
<b>Quota exceeded</b>	File count or size exceeded quota limit	Increase quota or clean files
<b>SafeModeException</b>	NameNode is in safe mode	Wait or run hdfs dfsadmin -safemode leave
<b>Connection refused</b>	NameNode or DataNode not running	Check and restart HDFS daemons

### Summary Table

Task	Command / Tool
<b>Detect corrupt blocks</b>	hdfs fsck / -files -blocks
<b>Enable Trash</b>	fs.trash.interval in core-site.xml
<b>Restore from Trash</b>	hdfs dfs -mv .Trash/Current/...
<b>Enable Snapshots</b>	dfsadmin -allowSnapshot
<b>Create Snapshot</b>	hdfs dfs -createSnapshot
<b>View logs</b>	tail -f \$HADOOP_HOME/logs/...
<b>Fix common errors</b>	Check permissions, replication, or restart services

# File System Administration in HDFS

## Safe Mode Operations (-safemode)

### 🧠 What Is Safe Mode?

Safe Mode is a **read-only state** of the HDFS NameNode. HDFS enters Safe Mode during:

- Startup
- Major maintenance
- Manual intervention

### ✅ Common Safemode Commands:

#### 📌 Check if HDFS is in Safe Mode:

```
hdfs dfsadmin -safemode get
```

#### 📌 Enter Safe Mode manually:

```
hdfs dfsadmin -safemode enter
```

#### 📌 Leave Safe Mode:

```
hdfs dfsadmin -safemode leave
```

💡 *Tip:* While in safe mode, you cannot write/delete files—only read operations are allowed.

## Balancer: Rebalancing Data Across DataNodes

### 🧠 Why Use the Balancer?

Over time, some DataNodes may fill up more than others. The **Balancer** re-distributes HDFS blocks to maintain **uniform storage usage**.

### ✅ Run the Balancer:

```
hdfs balancer
```

💡 *Note:* This operation is **non-disruptive**—it runs in the background and stops automatically when the balance threshold is met.

## Reporting with dfsadmin -report

### What It Does:

Provides a detailed report of the HDFS cluster status, including:

- Total and remaining space
- Number of live/dead DataNodes
- Block information

### Syntax:

```
hdfs dfsadmin -report
```

### Sample Output:

```
Configured Capacity: 500 GB
DFS Used: 150 GB
Non DFS Used: 20 GB
DFS Remaining: 330 GB
Live datanodes (3):
  Datanode1: 100 GB used
  Datanode2: 20 GB used
```

 Useful for monitoring and debugging storage issues.

## Checking File System Health (fsck)

### What Is fsck?

The **File System Check (fsck)** utility scans the HDFS namespace and finds:

- Corrupt blocks
- Missing replicas
- Under-replicated files

### Basic Usage:

```
hdfs fsck /
```

### Example with detailed info:

```
hdfs fsck / -files -blocks -locations
```

### Example for a specific file:

```
hdfs fsck /user/hadoop/data.csv
```

### Fixing Problems:

- Delete corrupt files (hdfs dfs -rm)
- Re-replicate under-replicated blocks
- Restart DataNodes if dead

## Upgrading and Downgrading HDFS

### When Needed:

- When moving to a new Hadoop version
- When reverting back due to compatibility issues

### Upgrade HDFS (Basic Steps):

1. Stop all Hadoop services:

```
stop-dfs.sh
```

2. Backup NameNode metadata and config:

```
cp -r $HADOOP_HOME/dfs/name /backup/
```

3. Start NameNode in upgrade mode:

```
hdfs namenode -upgrade
```

4. Start the cluster normally:

```
start-dfs.sh
```

5. Verify upgrade:

```
hdfs dfsadmin -upgradeProgress status
```

6. Finalize upgrade (non-reversible):

```
hdfs dfsadmin -finalizeUpgrade
```

### Downgrade (if needed)

1. Stop cluster
2. Restore backup metadata
3. Start with old version and metadata

 Never finalize an upgrade until you've tested the new version thoroughly!

### Summary Table

Task	Command Example
Enter/exit Safe Mode	dfsadmin -safemode enter/leave
Run balancer	hdfs balancer
View cluster report	dfsadmin -report
Check file health	fsck / -files -blocks
Upgrade HDFS	hdfs namenode -upgrade
Finalize upgrade	dfsadmin -finalizeUpgrade

# Performance and Optimization in HDFS

## Optimizing Replication Factor

### What It Is

The replication factor determines how many copies of each HDFS block are stored across DataNodes. By default, this is 3.

### Benefits of Adjusting

- **Lower replication** (e.g., 1–2) saves space on non-critical data.
- **Higher replication** (e.g., 4–5) increases fault tolerance and read performance.

### Set Globally (in hdfs-site.xml)

```
<property>
  <name>dfs.replication</name>
  <value>3</value>
</property>
```

### Set Per File:

```
hdfs dfs -setrep 2 /user/hadoop/input/file.txt
```

 *Tip:* Don't set replication too low for important files—data loss risk increases.

## Data Locality Awareness

### What It Means:

HDFS tries to schedule processing **on the same node** where the data resides to avoid network overhead.

### How It Works:

- HDFS exposes block locations to YARN and MapReduce.
- Job schedulers assign tasks where the data already exists.

### How to Improve Locality:

- Keep enough **task slots** on each DataNode.
  - Avoid overloading a few nodes (balance load).
  - Run the **HDFS balancer** regularly to spread data evenly.
-  Data locality improves speed and reduces bandwidth usage.

## HDFS Caching (In-Memory Data Caching)

### What It Does:

Caches frequently accessed files in **RAM** across DataNodes for **faster reads**.

### Enable Caching for a File:

```
hdfs cacheadmin -addDirective -path /user/hadoop/input/file.txt -pool default
```

### List Cached Directives:

```
hdfs cacheadmin -listDirectives
```

### Remove Cache:

```
hdfs cacheadmin -removeDirective -id <directive-id>
```

 Ideal for small files or data used repeatedly in analytic workloads.

## Rack Awareness Configuration

### What Is Rack Awareness?

Rack awareness ensures that:

- Replicas are placed across **different physical racks**
- Fault tolerance improves (survive rack failure)
- Network traffic is optimized

### Enable in hdfs-site.xml:

```
<property>
  <name>net.topology.script.file.name</name>
  <value>/etc/hadoop/topology.sh</value>
</property>
```

- This script maps hostnames to rack IDs.

### Example Output from topology.sh:

```
node1.hadoop.com /rack1
node2.hadoop.com /rack2
```

 Proper rack awareness ensures **at least one replica is outside the failed rack**.

## HDFS Federation

### What Is HDFS Federation?

Federation allows **multiple independent NameNodes** to manage different portions of the file system. All share the same set of DataNodes.

### Benefits

- Removes single NameNode bottleneck
- Scales horizontally
- Better isolation between departments or workloads

### Federation Setup Overview

1. Configure multiple namenodes and nameservices in hdfs-site.xml.
2. Mount NameNode namespaces using ViewFS.
3. DataNodes register with all NameNodes.

### Example ViewFS Mount:

```
<property>
  <name>fs.viewfs.mounttable.cluster1.link./dept1</name>
  <value>hdfs://namenode1:8020/dept1</value>
</property>
```

 Federation is useful for **very large clusters** with high concurrency.

### Summary Table

Feature	Purpose	Example/Command
Replication tuning	Balance fault tolerance vs. storage	-setrep 2 /file
Data locality	Faster job execution	Scheduler assigns task to DataNode
In-memory caching	Speed up hot data access	cacheadmin -addDirective
Rack awareness	Prevent data loss across racks	Use topology.sh
HDFS Federation	Scale with multiple NameNodes	Use ViewFS

# HDFS in a Cloud-Native World

As data systems move to the cloud, HDFS continues to evolve. Although cloud object stores like Amazon S3, Google Cloud Storage (GCS), and Azure Data Lake Storage (ADLS) are more popular, HDFS still plays a major role, especially in managed big data services and hybrid setups.

This chapter explains how HDFS works in the cloud, integrates with cloud-native storage, and how to run HDFS-like functionality using Kubernetes.

## HDFS on Azure HDInsight

### What is HDInsight?

Azure HDInsight is a **managed Hadoop service** that supports HDFS, Spark, Hive, and other Hadoop tools on the cloud.

### Features:

- Built-in HDFS
- Auto-scaling and high availability
- Integrated with **Azure Data Lake Storage (ADLS)**

### HDFS Access on HDInsight:

- Each Hadoop cluster includes an internal HDFS.
- You can also mount **ADLS or Blob Storage** to act like HDFS:

```
wasbs://<container>@<account>.blob.core.windows.net/<path>  
abfss://<filesystem>@<account>.dfs.core.windows.net/<path>
```

 *Tip:* Use ADLS Gen2 for hierarchical namespace support (similar to HDFS structure).

## HDFS on Google Cloud Dataproc

### What is Dataproc?

Google Cloud Dataproc is a **managed Spark/Hadoop service** on GCP that includes:

- HDFS
- YARN
- Hive, Presto, etc.

### Dataproc Storage Options:

- Internal HDFS (default)
- Native integration with **Google Cloud Storage (GCS)**

### Access GCS as HDFS:

gs://your-bucket/path

- Use gcs-connector to read/write like HDFS
- GCS supports hadoop fs and hdfs dfs commands via the connector

 Dataproc jobs can run **directly on GCS**, skipping HDFS entirely for many use cases.

## Integrating HDFS with Cloud Storage (GCS, ADLS)

### Why Integrate?

- Object stores (GCS, S3, ADLS) offer **low-cost, scalable** storage.
- HDFS is faster for **temporary, local processing**, but object stores are better for long-term storage.

### Hadoop Support for Object Stores:

Cloud	File System Prefix	Connector Name
GCP	gs://	gcs-connector
Azure	abfss://, wasbs://	azure-data-lake-store
AWS	s3a://	hadoop-aws

### Example: Read from ADLS using Hadoop

```
hdfs dfs -ls abfss://data@youraccount.dfs.core.windows.net/
```

 These connectors are configured in core-site.xml with authentication credentials or IAM roles.

## 4. Using HDFS with Kubernetes

### Why Use HDFS with Kubernetes?

- Kubernetes is the cloud-native standard for container orchestration.
- While HDFS was not designed for containers, **cloud-native adaptations** allow it to work.

### Options to Run HDFS on Kubernetes:

#### a) HDFS Helm Chart

You can deploy HDFS on K8s using community Helm charts:

```
helm install my-hdfs stable/hdfs
```

## b) HDFS Operator (via Apache Hadoop on K8s)

Experimental support exists for running NameNode and DataNodes as **stateful sets** in Kubernetes clusters.

## c) Alluxio (HDFS API Compatible)

Alluxio acts as an **in-memory cache** that speaks HDFS API and runs easily on Kubernetes:

```
hdfs dfs -ls alluxio://<hostname>:19998/path
```

### Volumes and Persistent Storage:

Use **PersistentVolumes** (PVs) to simulate DataNode storage:

```
kind: PersistentVolume
apiVersion: v1
spec:
  storageClassName: hdfs-storage
  capacity:
    storage: 100Gi
```

 Cloud-native file systems like **SeaweedFS**, **MinIO**, or **CephFS** are often better alternatives to HDFS in K8s.

### Summary Table

Topic	Description / Tool	Example
HDFS on Azure	HDInsight with ADLS support	abfss:// paths
HDFS on GCP	Dataproc with GCS connector	gs://bucket/
Cloud object store integration	Use GCS, ADLS, S3 via Hadoop connectors	hadoop fs -ls abfss://...
HDFS on Kubernetes	Run HDFS via Helm or StatefulSets	helm install hdfs
HDFS Alternatives on K8s	Use Alluxio or other HDFS-compatible layers	alluxio://

# Map Reduce

## Data Locality and Code Locality

### 💡 What Is Data Locality?

**Data locality** refers to the principle of moving computation closer to where the data resides, instead of transferring large datasets across a network to the computation.

### 🔍 Why is Data Locality Important?

- Reduces **network traffic**
- Improves **performance and throughput**
- Optimizes resource usage
- Critical for **big data processing frameworks** like Hadoop MapReduce, Spark, and Flink

### 💡 Data Locality in Hadoop MapReduce

Hadoop stores data in **HDFS**, which is distributed across multiple **DataNodes**. When a **MapReduce job** runs, the **Map tasks** ideally run on the **same node** where the data block is stored. This is called **data-local processing**.

### 🧠 Example:

Imagine you have a 1TB log file stored across 10 DataNodes. If you need to process that file:

- Without data locality: The entire data might be pulled over the network.
- With data locality: Tasks run **on the nodes** holding the relevant data blocks.

### 📁 Types of Data Locality

Type	Description
<b>Data-Local</b>	Task runs on the node containing the data block.  Ideal case.
<b>Rack-Local</b>	Task runs on a different node but <b>same rack</b> as the data.
<b>Off-Rack/Remote</b>	Task runs on a different <b>rack or data center</b> .  Least efficient.

Hadoop **tries for data-local** execution first. If no suitable node is available (e.g., busy or down), it falls back to rack-local or off-rack execution.

## What is Code Locality?

**Code locality** means **deploying or pushing the processing logic (MapReduce code)** to the machine or node where the data is stored.

### In Hadoop:

- You don't **move the data** to a central processing location.
- You **push the MapReduce code (jar files, logic)** to each DataNode and run it **locally on the data**.

### Why It Works: Move Code, Not Data

- Code is usually a few MBs; data might be **terabytes**
- Transferring code is cheap; transferring data is expensive

### Example:

Instead of:

10 GB log file → sent to central processor

We do:

MapReduce JAR (10 MB) → sent to DataNode holding the log file block

### Role in MapReduce Framework

Stage	Locality Role
Map Phase	Maximizes data locality by placing tasks on data-residing nodes
Shuffle/Reduce	Locality is less effective (data usually moves across the network)

### Challenges to Data Locality

- Cluster congestion (slots on data-local nodes may be busy)
- Skewed data distribution
- High-priority jobs may override locality

Hadoop has a **delay scheduling** mechanism to wait a short time for data-local tasks to become available before sacrificing locality.

## Related Concepts

Concept	Description
TaskTracker (Hadoop v1)	Tries to assign map tasks to local data blocks
YARN (Hadoop v2+)	ResourceManager tries to maintain data locality via NodeManager
Delay Scheduling	Waits before assigning non-local tasks in hopes of regaining locality
Speculative Execution	Sometimes launches duplicate tasks to handle stragglers, possibly breaking locality

## Summary Table

Aspect	Data Locality	Code Locality
Definition	Computation is placed near data	Code is sent to where data resides
Goal	Reduce network I/O, improve speed	Minimize data transfer overhead
Used in	Hadoop, Spark, Flink	Hadoop, distributed databases
Efficiency	Very high for large data processing	High when code is lightweight

## Final Thought

**"In Big Data, moving data is expensive; moving code is cheap."**

Data locality and code locality are key design principles that make distributed processing scalable, fault-tolerant, and performant.

# Map Reduce Architecture

## What is MapReduce?

**MapReduce** is a **programming model** and **processing engine** used to process and generate large datasets with a **distributed and parallel algorithm** on a **cluster of computers**.

It was originally developed by **Google**, and later adopted by **Apache Hadoop**.

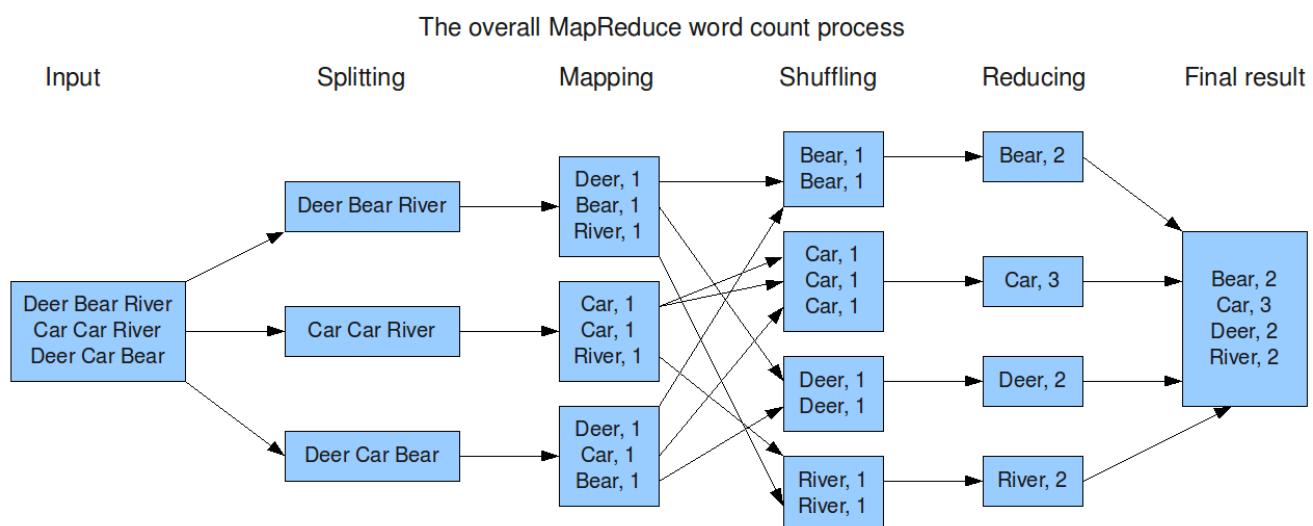
## MapReduce Architecture Overview

MapReduce consists of two main phases:

1. **Map Phase** – Processes and transforms the input data.
2. **Reduce Phase** – Aggregates, summarizes, or combines the output from the map phase.

It also has intermediate phases like **Shuffle and Sort**, which connect the Map and Reduce tasks.

## High-Level Flow



## Components in MapReduce Architecture

Component	Description
<b>Client</b>	Submits the MapReduce job to the system
<b>JobTracker / ResourceManager</b>	Coordinates the job execution across the cluster
<b>TaskTracker / NodeManager</b>	Executes map and reduce tasks on individual nodes
<b>HDFS</b>	Stores the input data and final output
<b>Map Function</b>	Converts input into key-value pairs
<b>Reduce Function</b>	Merges, summarizes, or processes grouped key-value pairs

## Step-by-Step Execution of a MapReduce Job

### 1 Job Submission

- A **client** program submits the MapReduce job (JAR file, input, output, config) to the **JobTracker** (Hadoop v1) or **ResourceManager** (YARN, Hadoop v2).

### 2 Input Splitting

- The input data is **split into chunks** (default 128 MB) by the **InputFormat** class.
- Each chunk is assigned to a **Map Task**.

► These splits are also known as **InputSplits**.

### 3 Map Phase

- Each **Map Task** reads a chunk of data and applies the **Map function**, which processes the data and emits intermediate **key-value pairs**.

Example:

Input: "Hello world hello"

Map Output: (hello, 1), (world, 1), (hello, 1)

### 4 Shuffle and Sort Phase

- Hadoop automatically **groups all values by key**.
- Intermediate data is **shuffled** across the network so that **all data for one key** ends up at the **same reducer**.
- Data is also **sorted** by key.

► This phase connects the output of the Map phase to the input of the Reduce phase.

### 5 Reduce Phase

- Each **Reduce Task** processes a group of keys and applies the **Reduce function** to aggregate results.

Example:

Reduce Input: (hello, [1, 1]), (world, [1])

Reduce Output: (hello, 2), (world, 1)

## 6 Output Format

- Final output is written to **HDFS** or any configured output storage.

## 7 Key Features of MapReduce

Feature	Explanation
<b>Scalability</b>	Can run on clusters with thousands of nodes
<b>Fault Tolerance</b>	Automatically retries failed tasks on other nodes
<b>Data Locality</b>	Moves code to the data to reduce network overhead
<b>Parallelism</b>	Map tasks and Reduce tasks run in parallel on different nodes

## 8 MapReduce in Hadoop Ecosystem

Layer	Technology
<b>Storage</b>	HDFS
<b>Processing</b>	MapReduce Engine
<b>Coordination</b>	YARN (Hadoop 2.x)
<b>Resource Mgmt</b>	ResourceManager + NodeManagers

## 9 Visual Representation (Text)

### Limitations of MapReduce

Limitation	Why It's a Problem
<b>Disk-based</b>	Reads/writes to disk between stages
<b>Not ideal for real-time</b>	Best for batch jobs only
<b>Complex programming</b>	Requires writing Map and Reduce logic in Java/Scala
<b>No built-in DAG support</b>	Cannot easily define complex data pipelines

Tools like **Apache Spark** and **Apache Flink** were designed to address these.

\*\*\* 1 CPU core can process 1 mapper job on 1 block/input split of data

\*\*\* We cannot control number of mapper tasks. Mapper tasks are equals to block count and limited by CPU core count.

## Summary Table

Stage	Purpose
Map	Transform and generate key-value pairs
Shuffle	Reorganize data by key across network
Sort	Sort intermediate data by key
Reduce	Aggregate data and produce final results

## Map Phase – The Data Transformer

### Purpose

- Breaks down the input data into smaller pieces.
- Transforms raw data into **intermediate key-value pairs** that are easier to group and aggregate later.

### Why It's Important

- It's where **pre-processing** and **filtering** happen.
- Prepares data for distributed aggregation by tagging relevant fields as keys.

### Example (Word Count):

Input: "hello world hello"

Map Output: (hello, 1), (world, 1), (hello, 1)

Without mapping, you can't distribute work across the cluster or identify what needs to be grouped.

## ◆ 2. Shuffle & Sort Phase – The Organizer

### Purpose:

- **Groups all values by key** across the cluster.
- Ensures all data associated with a key goes to the **same Reducer**.
- Sorts keys for efficient reducing.

### Why It's Important:

- Enables **distributed aggregation**.
- **Reorganizes data** by key, even though it was originally scattered across many nodes.
- A key step in converting parallel processing into coherent grouped results.

## Example:

From multiple Mappers:

(hello, 1), (hello, 1), (world, 1), (hello, 1)

After Shuffle:

hello → [1, 1, 1]

world → [1]

Without shuffle, a reducer wouldn't know **which values belong together**, making aggregation impossible.

## 3. Reduce Phase – The Aggregator

### Purpose:

- Processes grouped key-value pairs to generate the **final output**.
- Performs **aggregation, summarization, or calculations**.

### Why It's Important:

- Converts intermediate results into **meaningful final results**.
- Final step in the transformation process that produces human- or system-readable output.

### Example (continued):

hello → [1, 1, 1] → (hello, 3)

world → [1] → (world, 1)

Without reducing, all you have is grouped data; no totals, insights, or processed output.

## Summary Table: Importance of Each Phase

Phase	Role	Why It's Crucial
Map	Data transformation	Breaks raw data into meaningful pieces
Shuffle	Data redistribution	Groups same-key data across the cluster
Reduce	Aggregation/summary	Generates the final output from groups

## Real-World Analogy

Imagine you're organizing a national student test scoring system:

- **Map** = Each school calculates scores for its students.
- **Shuffle** = All scores for a subject are collected and grouped by subject.
- **Reduce** = Final national average per subject is calculated.

# Zero Reducer

## What is a Zero Reducer in MapReduce?

A **Zero Reducer** means a **MapReduce job that doesn't require a Reduce phase**—only the **Map phase is executed**, and its output is written directly to the final output (typically in HDFS or S3).

### When Do You Use Zero Reducers?

When your task:

- **Does not require aggregation, grouping, or summarization**
- Is more about **filtering, transforming, or formatting** the input data

### Examples of Zero Reducer Jobs

Use Case	What Happens in Map Phase	Why No Reduce Needed?
<b>Log filtering</b>	Drop all lines except ERROR lines	No grouping needed
<b>Data format transformation</b>	Convert JSON to CSV or XML	Each record handled independently
<b>Copying or moving data</b>	Read from one file, write to another	Just a pass-through
<b>Tokenizing text</b>	Split sentences into words	No aggregation required
<b>File scanning (e.g., malware tags)</b>	Scan records for a signature or pattern	Each line is independent

### How to Configure a Zero Reducer in Hadoop

#### CLI:

```
hadoop jar yourjob.jar YourMapOnlyJob -D mapreduce.job.reduces=0 input_path output_path
```

#### In Java Code:

```
job.setNumReduceTasks(0);
```

## Important Notes

Point	Explanation
Faster Execution	Skipping the Reduce phase can <b>significantly reduce job time</b> .
Less Network Overhead	No shuffle phase means <b>no intermediate data movement</b> .
No Sorting	Outputs are <b>not sorted by key</b> unless handled in the Map phase.
Good for Preprocessing	Often used in <b>ETL pipelines</b> as the first stage.

## Summary

Attribute	Zero Reducer Jobs
Reduce Tasks	0
Use Case	Filtering, transforming, copying
Performance	High (since no shuffling)
Limitations	No grouping, aggregation, sorting

## Real-World Analogy:

Like a factory line where items are cleaned and repackaged one by one, but **not grouped, compared, or combined** — that's a zero reducer.

# Input Splits

## What are Input Splits in Hadoop MapReduce?

**Input Splits** are the **logical representation of input data** chunks that determine how a MapReduce job is divided into **individual tasks**. Each input split is processed by **one map task**.

### Why Input Splits Matter

- They **enable parallelism** in Hadoop by breaking large files into manageable chunks.
- Each **Map Task** processes one split.
- The **efficiency and scalability** of a MapReduce job largely depend on how well data is split.

### How It Works

1. A large file (e.g., 2 GB) is stored in **blocks** in HDFS (default 128 MB per block).
2. When a MapReduce job runs, Hadoop's **InputFormat** (like TextInputFormat) divides the input into **Input Splits**.
3. These splits are **logical** – they reference **byte ranges**, not actual files.
4. Each **split is assigned to a Mapper**.

### Example

File Size	Block Size	Number of Input Splits
512 MB	128 MB	4 splits
1 GB	256 MB	4 splits

### Difference: Block vs Input Split

Aspect	HDFS Block	Input Split
Physical?	Yes (actual data on disk)	No (logical reference to block data)
Controls?	Storage in HDFS	Number of map tasks
Size	Fixed (default 128MB)	Can be adjusted in job configuration
Purpose	File storage	Task scheduling in MapReduce

## Configuring Split Size

Hadoop allows you to control the minimum and maximum split size:

`mapreduce.input.fileinputformat.split.maxsize`

`mapreduce.input.fileinputformat.split.minsize`

If a file is smaller than the block size, one Mapper handles it entirely.

## InputFormat and Input Splits

InputFormat	How It Splits Data
<code>TextInputFormat</code>	Line-by-line, default for text files
<code>KeyValueInputFormat</code>	Splits using delimiter (e.g., tab)
<code>NLineInputFormat</code>	N lines per split
<code>CombineFileInputFormat</code>	Combines small files into one split

### Real-World Problem: Small Files

- Too many tiny files → too many input splits → too many mappers → inefficient.
- **Solution:** Use `CombineFileInputFormat` or tools like **Hadoop Archive (HAR)**.

## Summary

Feature	Description
<b>Input Split</b>	Logical chunk of data assigned to a Mapper
<b>Purpose</b>	Enable parallelism & efficient processing
<b>One split =</b>	One Map Task
<b>Defined by</b>	InputFormat implementation
<b>Optimizable?</b>	Yes (can set min/max split size)

## Analogy:

Think of a big book being read by multiple people. **Input splits** are like assigning **pages** to each reader so everyone reads a different part simultaneously.

# YARN

## Introduction

### What is YARN?

YARN stands for **Yet Another Resource Negotiator**.

It is the **resource management and job scheduling layer** of **Apache Hadoop** (introduced in Hadoop 2.x).

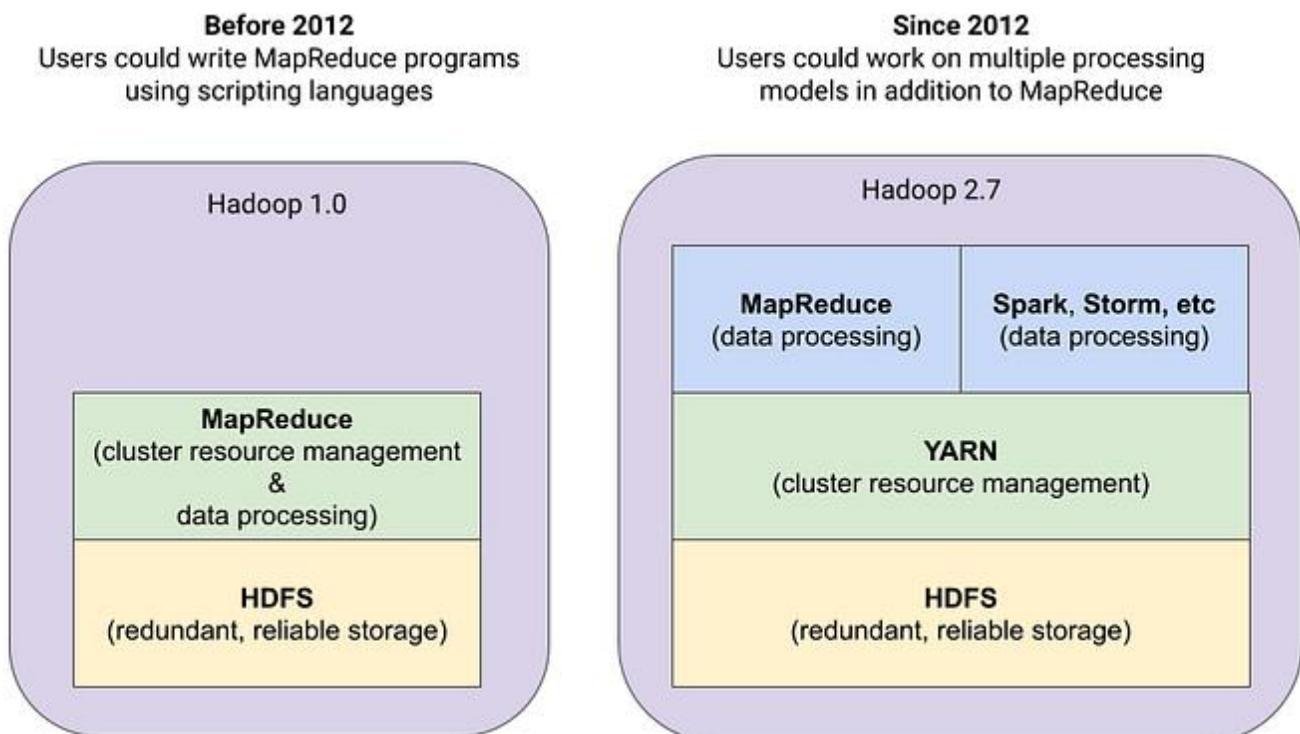
- ◆ **YARN decouples** the resource management and job execution components from the original MapReduce model.
- ◆ It allows **multiple data processing engines** (like MapReduce, Spark, Hive, etc.) to run and share resources in a Hadoop cluster.

### ⌚ Why YARN Was Introduced

Before YARN (in Hadoop 1.x), the **MapReduce framework** handled both **resource management** and **job scheduling**, which made it:

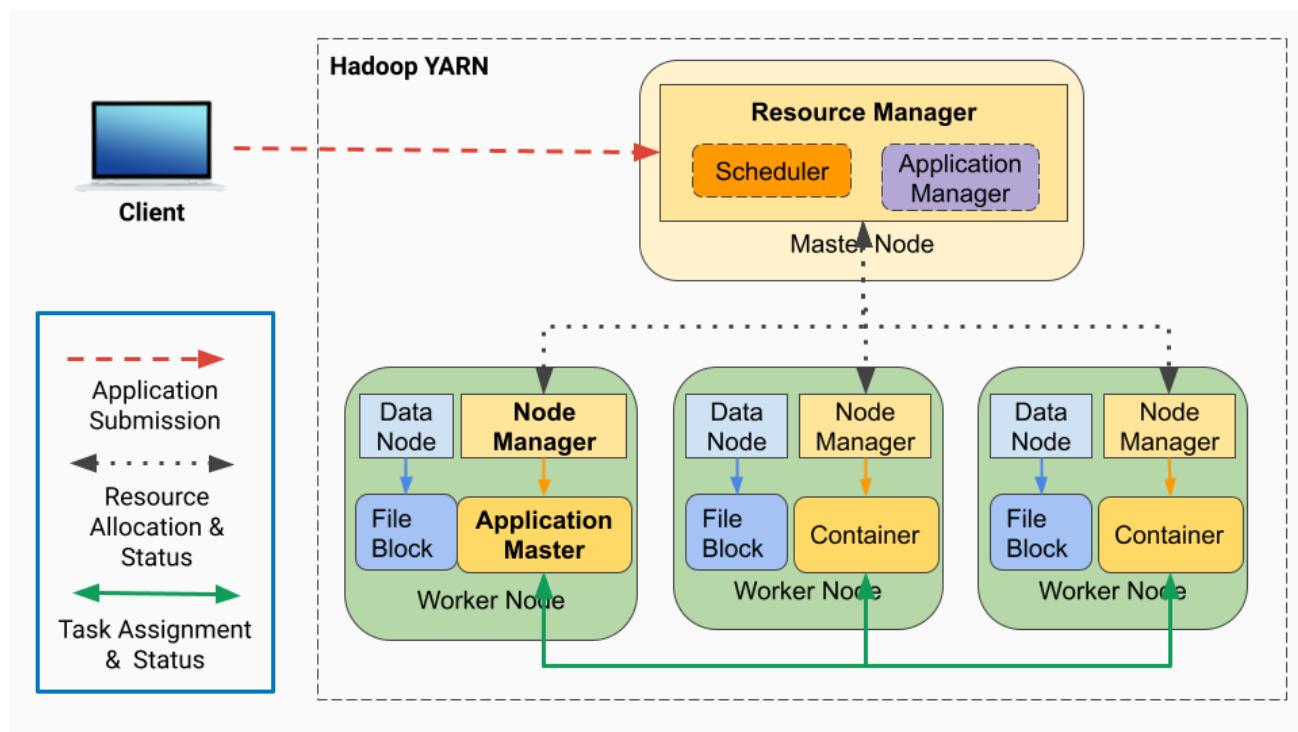
- Inflexible (only supported MapReduce)
- Hard to scale
- Poor at multi-tenant workloads

YARN separates these responsibilities, enabling **scalability**, **multi-framework support**, and **better cluster utilization**.



## Key Components of YARN Architecture

Component	Description
<b>ResourceManager (RM)</b>	The master that manages resources <b>across the cluster</b> . It decides how much and where to allocate resources.
<b>NodeManager (NM)</b>	Runs on each node. <b>Monitors resource usage</b> and <b>executes containers</b> on behalf of the ApplicationMaster.
<b>ApplicationMaster (AM)</b>	Created per job/application. It <b>negotiates resources</b> with the RM and <b>monitors tasks</b> .
<b>Containers</b>	YARN's basic unit of resource allocation (CPU + memory). All tasks run inside containers.



## YARN Job Execution Workflow

Step 1: User submits an application (like a Spark or MapReduce job).

Step 2: ResourceManager allocates a container to launch the ApplicationMaster (AM).

Step 3: AM negotiates with RM to get containers for running tasks.

Step 4: NodeManagers launch containers to run the actual work.

Step 5: AM monitors task progress and informs RM upon completion.

## Example: Running a Spark Job on YARN

Phase	What Happens
<b>Submission</b>	Spark job submitted via CLI
<b>Container 1</b>	Launches Spark Driver (AM)
<b>Containers 2+</b>	Launch Spark Executors (tasks)
<b>NodeManagers</b>	Run the actual work inside containers

## Advantages of YARN

Feature	Benefit
<b>Multi-engine support</b>	Run MapReduce, Spark, Tez, Hive, Flink, etc. on the same cluster
<b>Resource isolation</b>	Each task gets allocated memory and CPU via containers
<b>Scalability</b>	Supports thousands of nodes
<b>Fault tolerance</b>	Automatically reassigns failed containers
<b>Improved utilization</b>	Dynamically allocates resources based on demand

## YARN vs Classic Hadoop (Pre-Hadoop 2.x)

Feature	Hadoop 1.x (Classic)	Hadoop 2.x+ (YARN)
<b>Framework Support</b>	Only MapReduce	MapReduce, Spark, Hive, Flink, etc.
<b>Resource Management</b>	Tied to JobTracker	Done by ResourceManager
<b>Scalability</b>	Limited	Highly scalable
<b>Flexibility</b>	Low	High

## Real-World Analogy

Think of YARN as the **operating system of your Hadoop cluster**:

- **ResourceManager = OS scheduler**
- **NodeManagers = Worker processes**
- **Containers = Programs or threads running on your OS**
- **ApplicationMaster = App manager for each program**

## Summary

Concept	Details
<b>Full Form</b>	Yet Another Resource Negotiator
<b>Role</b>	Resource manager and job scheduler in Hadoop
<b>Main Components</b>	ResourceManager, NodeManager, ApplicationMaster, Containers
<b>Introduced In</b>	Hadoop 2.x
<b>Supports</b>	MapReduce, Spark, Hive, Tez, etc.
<b>Key Benefits</b>	Scalability, flexibility, multi-engine support

# Google Cloud

## Google Cloud Dataproc

**Google Cloud Dataproc** is a **fully managed and fast Apache Spark and Hadoop service** that runs on Google Cloud Platform (GCP). It allows you to process **big data workloads using familiar open-source tools** like Spark, Hadoop, Hive, Pig, and more—without needing to manually manage clusters.

### ❖ What is Google Cloud Dataproc?

**Dataproc** is Google Cloud's managed solution for running **open-source distributed data processing frameworks** (like Spark and Hadoop) in a **cost-effective, scalable, and easy-to-use way**.

### Core Features

Feature	Description
Managed Clusters	Quickly create and manage Hadoop/Spark clusters
Autoscaling	Automatically scales nodes based on workload
Pre-installed Tools	Hadoop, Spark, Hive, Pig, and other libraries are pre-configured
Per-second Billing	Pay only for what you use
Integration with GCP	Works with GCS (Google Cloud Storage), BigQuery, Cloud Logging, etc.
Custom Images	Supports initialization actions and custom software configurations

### Architecture Overview



## Components of a Dataproc Cluster

Component	Description
<b>Master Node</b>	Manages cluster and job scheduling (like YARN ResourceManager)
<b>Worker Nodes</b>	Run actual data processing tasks (MapReduce/Spark jobs)
<b>Preemptible Workers</b>	Cost-saving temporary workers (optional)

## How It Works

### 1. Create a Dataproc Cluster

```
gcloud dataproc clusters create my-cluster \
--region=us-central1 \
--num-workers=2 \
--image-version=2.1-debian11 \
--project=my-gcp-project
```

### 2. Submit a Spark Job

```
gcloud dataproc jobs submit spark \
--cluster=my-cluster \
--region=us-central1 \
--class=org.apache.spark.examples.SparkPi \
--jars=file:///usr/lib/spark/examples/jars/spark-examples.jar \
-- 100
```

### 3. View Logs and Metrics

- Logs go to **Cloud Logging**
- Metrics to **Cloud Monitoring**
- Outputs to **GCS or BigQuery**

## 🧠 Key Integrations

GCP Service	Integration Role
<b>Cloud Storage (GCS)</b>	Store input/output data
<b>BigQuery</b>	Run analytics queries on processed data
<b>Cloud Composer (Airflow)</b>	Automate job orchestration
<b>Cloud Logging</b>	Monitor job logs and cluster events
<b>Cloud Monitoring</b>	Track system and cluster health

## Use Cases

Use Case	Example
ETL Pipelines	Ingest → Transform → Load large datasets
Log Processing	Analyze terabytes of server logs
Machine Learning	Train models using Spark MLlib
Batch + Streaming Mix	Use Spark Structured Streaming with batch analysis
Genomics / Bioinformatics	Analyze scientific data at scale

## Pricing Highlights

Cost Item	Notes
Per-second billing	Only pay for the duration the cluster runs
Separate storage	Data stored in GCS, not tied to cluster size
Autoscaling	Dynamically adjusts worker count
Preemptible VMs	Save up to 80% on temporary compute nodes

## ✓ Pros and Benefits

- ✓ Easy to spin up and tear down clusters
- ✓ Fully compatible with Apache Hadoop ecosystem
- ✓ Seamless integration with GCP services
- ✓ Ideal for **temporary workloads**
- ✓ Scales from dev testing to petabyte-scale pipelines

## ⚠ Limitations

- ✗ Still requires understanding of Hadoop/Spark concepts
- ✗ Manual cluster start/stop needed for basic setups
- ✗ Less ideal for continuous real-time streaming (use **Dataflow** instead)

## Summary

Feature	Google Cloud Dataproc
Purpose	Managed Hadoop/Spark/Pig/Hive environment
Setup Time	~90 seconds to spin up a cluster
Billing	Per-second, VM-based
Tools Supported	Spark, Hadoop, Hive, Pig, Jupyter, Dask
Use Case	ETL, ML, analytics, log processing
Storage Backend	Google Cloud Storage (GCS)

## Guide

Assign a name to the cluster , and select the location.

**Name**

Cluster name \*  ?

**Location**

Region \*  ? Zone \*  ?

**Cluster type**

- Standard (1 master, N workers)
- Single Node (1 master, 0 workers)  
Provides one node that acts as both master and worker. Good for proof-of-concept or small-scale processing
- High availability (3 masters, N workers)  
Hadoop high availability mode provides uninterrupted YARN and HDFS operations despite single-node failures or reboots

Cluster Type	Masters	Workers	Use Case	Fault Tolerance
Standard	1 Master	N Workers	General workloads	Low (single master failure causes downtime)
Single Node	1 (Master + Worker)	0	Testing, small-scale, PoC	None
High Availability	3 Masters	N Workers	Production-grade, critical systems	High (tolerates master failures)

Select the image in versioning section. Don't select most newest versions since may unstable.

## Choose Image Version

### STANDARD DATAPROC IMAGE

### CUSTOM IMAGE

Cloud Dataproc uses versioned images to bundle the operating system, big Platform connectors into one package that is deployed on your cluster. [Learn more](#)

- 2.3 (Debian 12, Hadoop 3.3, Spark 3.5)  
First released on 06/09/2025
- 2.3 (RockyLinux 9, Hadoop 3.3, Spark 3.5)  
First released on 06/09/2025
- 2.3 (Ubuntu 22 LTS, Hadoop 3.3, Spark 3.5)  
First released on 06/09/2025
- 2.2 (Debian 12, Hadoop 3.3, Spark 3.5)  
First released on 8 December 2023.
- 2.2 (RockyLinux 9, Hadoop 3.3, Spark 3.5)  
First released on 8 December 2023.

#### Enable Advanced Optimizations

- Turns on **Catalyst** and **Tungsten** optimizations in Spark.
- Includes **predicate pushdown**, **logical plan simplification**, **code generation**, etc.
- Speeds up transformations and reduces shuffle overhead.

*Good for performance-sensitive ETL or SQL workloads.*

#### Enable Advanced Execution Layer

- Enables **project-specific enhancements** for Spark DAG execution.
- Might include **Dynamic Allocation**, **Adaptive Query Execution**, or **improved scheduling algorithms**.
- Aimed at **minimizing shuffle**, **reducing memory pressure**, and **faster job execution**.

*Best for mixed and multi-tenant Spark workloads.*

#### Enable Google Cloud Storage (GCS) Caching

- Caches frequently accessed files from GCS in memory or disk on DataNodes.
- Reduces network IO latency for iterative Spark jobs or SQL reads.

*Recommended when reading Parquet, Avro, or ORC files multiple times from GCS.*

## AUTOSCALING AND FLEXIBILITY

### Auto-Scaling

- Automatically **adds or removes worker nodes** based on CPU, memory, or job queue usage.
- Requires a **defined autoscaling policy**.

Cost-effective: scales down during idle time.

### Enhanced Flexibility Mode (EFM)

- Solves the common problem of **shuffle data loss** when nodes are removed mid-job.
- Two shuffle management modes:

#### 1. Primary Worker Shuffle:

- Designates a subset of nodes to **persist shuffle data**.
- Removed nodes never carry important shuffle blocks.

#### 2. HCFS Shuffle (Hadoop Compatible File System):

- Uses **cloud storage** (e.g., GCS) for writing shuffle files.
- Ideal for long jobs with frequent up/down scaling.

Use EFM to avoid job restarts during autoscaling.

## NETWORK CONFIGURATION

### Primary Network & Subnetwork

- Determines the **VPC and subnet** where Dataproc VMs are deployed.
- Controls IP allocation, firewall rules, and routing.

Use a shared VPC if clusters span projects or need centralized control.

### Network Tags

- Tags are used to **apply firewall rules** and **route configurations**.
- E.g., tag spark-worker could allow SSH only to Spark worker nodes.

## METADATA & METASTORE

### Dataproc Metastore

- Centralized **Hive-compatible metastore** for storing table and schema metadata.
- Allows **multiple Dataproc clusters** to share the same Hive Metastore.
- Keeps metadata **persistent** across cluster restarts.

*Essential for Hive, SparkSQL, Trino, and Presto compatibility.*

## COMPONENT GATEWAY & COMPONENTS

### Enable Component Gateway

- Exposes **web UIs** (Spark UI, YARN UI, Jupyter, etc.) using **secure URLs** (IAM authenticated).
- Replaces SSH tunneling for component access.

### Optional Components (Brief Explanation):

Component	Description
Hive WebHCat	Web interface to Hive via REST API; used for external app access.
Jupyter Notebook	Interactive Python/Scala notebooks; good for data exploration or ML pipelines.
Zeppelin Notebook	Similar to Jupyter; supports Spark, Hive, Flink, SQL, shell, etc.
Trino (PrestoSQL)	Distributed SQL query engine for big data (can query HDFS, GCS, Hive, etc).
ZooKeeper	Distributed coordination service; needed for HBase, Kafka, Trino.
Ranger	Provides data access governance and auditing for Hadoop components.
Flink	Real-time stream processing engine (better than Spark Streaming for complex flows).
Docker	Support for custom container-based workloads on the cluster.
Solr	Open-source enterprise search platform (indexed search engine like Elasticsearch).
Hudi	Incremental data lake platform for ACID updates/deletes on Parquet (Hive/Spark).
Iceberg	Open table format optimized for analytical workloads (huge datasets).
Delta	Databricks' Delta Lake format; supports schema enforcement and versioning.

## Summary Tips

Task Type	Recommended Option
Batch ETL + SQL	Enable advanced optimizations + metastore + GCS caching
Interactive analysis	Use Jupyter or Zeppelin + component gateway
Scalable jobs	Enable autoscaling + enhanced flexibility mode
Stream processing	Add Flink + ZooKeeper
Governance & Security	Add Ranger

## NODE CONFIGURATION IN DATAPROC

### ◆ Types of Nodes in a Dataproc Cluster

Node Type	Description
Master Node	Runs the cluster manager (YARN ResourceManager), Spark driver, and NameNode. Usually one node.
Worker Nodes	Run Spark executors, HDFS DataNodes, etc. Perform the actual compute and storage work.
Secondary Worker (Preemptible)	Optional; cheaper, short-lived nodes for bursty workloads. Good for auto-scaling clusters.

## NODE SETTINGS YOU CAN CONFIGURE

### 1. Machine Type (CPU + RAM)

- Choose from standard, high-CPU, high-memory (e.g., n2-standard-4, e2-highmem-8).
- Master can be smaller than workers if you're running large parallel workloads.

 Tip: Match machine type to workload type — CPU-bound, memory-bound, or I/O-bound.

### 2. Number of Nodes

- Set **initial worker count**, **minimum**, and **maximum** for auto-scaling.
- Example for autoscaling:
  - **Initial workers:** 2
  - **Min:** 2
  - **Max:** 20

### 3. Preemptible Workers (optional)

- Cost-effective, but can be shut down at any time.
- Used for **non-critical**, fault-tolerant compute like Spark tasks with retries.
- Usually defined in **secondary-worker configuration**.

### 4. Disk Settings

Option	Description
<b>Boot Disk Size</b>	Usually 100 GB or more for master and worker
<b>Disk Type</b>	Choose between pd-standard (HDD) and pd-ssd (faster)
<b>Local SSDs</b>	Optional; useful for Spark shuffle, caching, or high I/O

👉 For large shuffles or big joins, SSDs greatly improve performance.

### 5. Image Version

- Choose Dataproc image version like 2.1-debian12 or 2.0-ubuntu20.
- Determines the software versions (Spark, Hadoop, JDK, Python, etc.)

### 6. Custom Initialization Actions

- Run scripts on each node at startup to:
  - Install extra libraries
  - Configure Spark settings (like executor memory)
  - Mount storage buckets or attach secret keys

### 7. Networking

- Set:
  - **VPC network & subnetwork**
  - **IP allocation**
  - **Network tags** (for firewall rules, routing)

✓ Important for clusters in **shared VPC** setups or for securing access via firewall.

## 8. Labels & Tags

- Useful for:
  - **Billing**
  - **IAM policies**
  - **Monitoring & filtering** in the GCP console

## 9. Node Auto-Repair & Auto-Upgrade

- Auto-repair = VM will be restarted automatically if a hardware/software failure occurs.
- Auto-upgrade = Nodes will be updated with latest security patches.

### Best Practice Recommendations

Goal	Recommendation
<b>Cost efficiency</b>	Use preemptible workers for transient compute
<b>High I/O</b>	Use SSDs or enable GCS caching
<b>Memory-intensive jobs</b>	Use highmem or custom machine types
<b>Long-running pipelines</b>	Enable <b>auto-repair</b> and <b>flexibility mode</b>
<b>Security and management</b>	Use custom network + tags + IAM roles
<b>Scale-out jobs</b>	Enable autoscaling with proper policy tuning

### Example Configuration (Typical Medium Spark Job)

Setting	Value
<b>Master node type</b>	n2-standard-2
<b>Worker node type</b>	n2-standard-4
<b>Number of workers</b>	4 (autoscaling: 2–10)
<b>Preemptible workers</b>	4
<b>Disk type / size</b>	SSD / 200 GB
<b>GCS caching</b>	Enabled
<b>Component gateway</b>	Enabled
<b>Image version</b>	2.1-debian12
<b>Metastore</b>	Enabled (shared)

After creating the cluster, you can access terminal , Jupiter lab and other under the web interfaces.

# Higher Order Functions in Python

## What are Higher Order Functions?

In Python, **Higher Order Functions (HOFs)** are functions that **either**:

1. **Take another function as an argument**, or
2. **Return a function as a result**, or both.

👉 This is possible because **functions in Python are first-class citizens**. That means functions can be:

- Stored in variables
- Passed as arguments
- Returned from other functions

## Why Use Higher Order Functions?

Higher Order Functions help you:

- Write **clean, reusable, and modular code**
- Follow **functional programming** practices
- Improve code **abstraction and flexibility**

## Example 1: Passing Functions as Arguments

```
def greet(name):  
    return f"Hello, {name}!"  
  
def call_with_greeting(func, value):  
    return func(value)  
  
print(call_with_greeting(greet, "Alice")) # Output: Hello, Alice!
```

### Explanation:

- `greet` is a simple function.
- `call_with_greeting` is a higher-order function. It **takes another function** (`greet`) as an argument and calls it.

## Example 2: Returning Functions from Functions

```
def power_of(n):
    def power(x):
        return x ** n
    return power

square = power_of(2)
cube = power_of(3)

print(square(5)) # Output: 25
print(cube(2)) # Output: 8
```

### Explanation

- `power_of` returns a new function `power`.
- `square` and `cube` are customized functions returned by `power_of`.

## Built-in Higher Order Functions in Python

Python comes with several built-in higher-order functions. Let's explore the most useful ones:

### 1. `map(function, iterable)`

Applies a function to **each element** of an iterable.

```
numbers = [1, 2, 3, 4]
squared = list(map(lambda x: x**2, numbers))
print(squared) # Output: [1, 4, 9, 16]
```

### 2. `filter(function, iterable)`

Filters elements for which the function returns True.

```
numbers = [1, 2, 3, 4, 5]
even = list(filter(lambda x: x % 2 == 0, numbers))
print(even) # Output: [2, 4]
```

### 3. `reduce(function, iterable)`

Performs a **rolling computation** on a sequence (needs `functools` module).

```
from functools import reduce

numbers = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, numbers)
print(product) # Output: 24
```

## 4. sorted(iterable, key=function)

Sorts the iterable using a custom function as the sorting key.

```
words = ["banana", "apple", "cherry"]
sorted_words = sorted(words, key=len)
print(sorted_words) # Output: ['apple', 'banana', 'cherry']
```

## Lambda Functions with HOFs

Lambda functions (anonymous functions) are often used with higher-order functions:

```
nums = [1, 2, 3, 4]
doubled = list(map(lambda x: x * 2, nums))
print(doubled) # Output: [2, 4, 6, 8]
```

### Real-Life Analogy

Imagine a machine (function) that can **accept other machines as input** and **build new machines** using them — that's what higher order functions are!

### Benefits of Higher Order Functions

Benefit	Description
<b>Reusability</b>	Avoid repeating logic by using functions as inputs/outputs
<b>Abstraction</b>	You can focus on “what” to do, not “how”
<b>Clean Code</b>	Reduces boilerplate and makes code more elegant
<b>Functional Programming</b>	Encourages stateless, side-effect-free programming

### Pro Tip: Be Cautious with Side Effects

Higher order functions work best when the functions they use:

- Are **pure** (no side effects)
- **Return results** rather than modifying things

# SPARK

## Introduction to Apache Spark (Beginner-Friendly)

### Why Do We Need Spark?

Before Spark, **Hadoop MapReduce** was the most popular big data processing framework. But it had some problems

#### Problems with Hadoop MapReduce

- **Slow:** Writes data to disk after every step → very slow for complex workflows.
- **Hard to use:** Programming was verbose and required low-level code.
- **Not interactive:** No support for quick analysis or exploration of data.
- **Batch-only:** No support for real-time or streaming data.

Spark was created to overcome these limitations.

### What is Apache Spark?

**Apache Spark** is a fast, powerful, and easy-to-use **open-source big data processing framework**. It can handle **large-scale data processing** in a distributed environment.

#### Key Features

- **In-memory computation:** Faster than Hadoop by keeping data in memory.
- **Ease of use:** APIs available in Python, Java, Scala, R, and SQL.
- **Unified engine:** Handles **batch, streaming, SQL, ML, and graph processing**.
- **Fault-tolerant:** Automatically recovers from failures.

### Spark Ecosystem

Component	Description
<b>Spark Core</b>	The engine – handles basic I/O, scheduling, memory management.
<b>Spark SQL</b>	Work with structured data using SQL queries or DataFrames.
<b>Spark Streaming</b>	Real-time data processing (e.g., logs, sensor data).
<b>Mlib</b>	Machine learning library (classification, clustering, etc.).
<b>GraphX</b>	Library for graph processing (e.g., social networks).

### How Spark Works (Simplified)

1. **Cluster Setup:** Spark runs on a **cluster** (a group of machines).
2. **Driver Program:** Your Spark code is submitted to the **Driver**, which coordinates the execution.
3. **Transformations** (e.g., map(), filter()): Lazy operations that define a logical plan.
4. **Actions** (e.g., count(), collect()): Trigger execution.
5. **Executors:** Worker nodes execute tasks and return results.

### RDD – Resilient Distributed Dataset

- The basic data structure in Spark.
- Immutable, distributed collection of objects.
- Automatically split across machines.

## Spark vs Hadoop: Quick Comparison

Feature	Spark	Hadoop MapReduce
Speed	Very fast (in-memory)	Slower (disk-based)
Ease of Use	Simple APIs	Complex Java code
Real-time support	Yes (Spark Streaming)	No
Machine Learning	Yes (MLlib)	Limited
Fault Tolerance	Yes (RDD lineage)	Yes

## Real-Life Use Cases of Spark

- **Netflix:** Recommender systems using Spark MLlib.
- **Uber:** Analyzing real-time traffic data.
- **Amazon:** Processing millions of transactions.
- **Financial firms:** Fraud detection in real-time.

## Summary

Concept	Explanation
Why Spark?	To overcome Hadoop's limitations in speed, usability, and versatility.
What is Spark?	A fast, general-purpose big data processing engine.
Core Advantage	In-memory processing + unified API for all data types.
Works on	Local machines, Hadoop YARN, Kubernetes, or cloud (AWS, GCP, Azure).

## Common Questions

**Do I need to know Hadoop and YARN to learn Spark?**

**No, not required.**

But it's helpful to know:

- **Spark can run on YARN** (Hadoop's cluster manager), but also works standalone, on Kubernetes, or in the cloud.
- You can learn **Spark independently** using your local machine.

*Start with Spark first. Learn Hadoop/YARN later if needed for deployment.*

**Can Spark work with other file systems (not just HDFS)?**

**Yes. Spark works with many file systems:**

Common ones:

- **Local filesystem** (e.g., /home/user/data.csv)
- **Amazon S3**
- **Azure Blob Storage**
- **Google Cloud Storage**
- **HDFS** (Hadoop)
- **DBFS** (Databricks File System)

Spark reads data in **Parquet, JSON, CSV, Avro**, etc., from these file systems easily.

**Some practical use cases of Spark?**

1.  **ETL Pipelines:** Clean, transform and store huge datasets.
2.  **Data Analytics:** Query terabytes of logs, metrics, or transactions.
3.  **Machine Learning:** Spark MLlib trains models at scale (e.g., fraud detection).
4.  **Real-time streaming:** Process clickstreams, IoT sensor data using Spark Streaming.
5.  **Recommendation Engines:** Like those used by Netflix and Amazon.

**Spark on Pseudo-distributed system vs Clusters?**

Mode	Description	Good For
------	-------------	----------

<b>Pseudo-distributed (local)</b>	All nodes simulated on one machine (e.g., your laptop)	<b>Learning, testing</b>
<b>Cluster mode</b>	Multiple real machines (workers + master)	<b>Production, big data workloads</b>

- Start on local (pseudo-distributed), move to cluster as your data grows.

## How is Spark different from traditional databases (e.g., MySQL) or Pandas?

Feature	Spark	Traditional DB (MySQL) / Pandas
<b>Scale</b>	Handles TB–PB of data	Limited by RAM/disk of single machine
<b>Processing</b>	Distributed, in-memory (fast)	Disk-based (DB), in-memory (Pandas, small data)
<b>Use Case</b>	Big data, streaming, ML	Small to medium data, OLTP
<b>APIs</b>	SQL, Python, Scala, Java	SQL (DB), Python (Pandas)

💡 Spark ≠ Database → It processes data from databases or files in a distributed way.

## Why not just horizontally split MySQL database without using Spark?

Good thought, but...

⚠ Problems with horizontal MySQL sharding

- Difficult to **maintain consistency** and **indexes**.
- Complex **queries across shards** (joins, aggregations are hard).
- No **built-in parallel computation** or **in-memory processing**.
- **Limited fault tolerance**, recovery is complex.

Spark solves all of this

- Built-in parallelism and fault tolerance.
- Handles huge joins, aggregations across many nodes easily.
- Integrates with many databases and systems.

 Sharded DB = Storage trick

 Spark = Processing engine for distributed data

## Why use Databricks?

Databricks is a **managed Spark platform** built by Spark's creators.

 Reasons to use:

- No cluster setup needed.
- Easy notebook interface (like Jupyter).
- Auto-scaling, auto-optimizations.
- Built-in tools for ML, streaming, and Delta Lake.
- Secure, collaborative (great for teams).

 Think of it as “Spark as a Service” with enterprise power.

## Limitations of MapReduce

### 1. Heavily Dependent on Disk (Slow Performance)

- After each processing step (map → reduce), data is written to **disk (HDFS)**.
  - No in-memory processing → **very slow** for multi-step workflows.
  - Example: Running multiple joins or aggregations becomes painfully slow.
-  Spark, in contrast, uses **in-memory computing**, making it up to **100x faster** in many cases.

### 2. Batch Processing Only

- Designed for **batch jobs** that process static data.
- **No support for real-time or streaming data.**
- Can't handle continuous data flows like IoT, logs, or live user activity.

 Spark supports **real-time streaming** with **Spark Streaming** and **Structured Streaming**.

### 3. Code Complexity

- Requires writing a lot of **boilerplate Java code**.
- Developers must manage **map()**, **reduce()**, and **key-value logic** explicitly.
- Difficult for beginners and hard to debug.

 Spark provides **high-level APIs** in Python, Scala, SQL — much more readable and intuitive.

#### 4. Rigid and Limited Programming Model

- Follows a strict **Map → Shuffle → Reduce** structure.
- Not flexible for complex operations like machine learning, graph analytics, or iterative algorithms.

 Spark offers flexible operations (e.g., map, reduceByKey, join, flatMap, groupBy, window, etc.)

#### 5. Lack of Built-in Monitoring or UI

- **No default UI** for tracking job progress or performance.
- Monitoring and debugging MapReduce jobs is difficult.
- External tools (like Ambari) are needed for visibility.

 Spark includes a **built-in Web UI** to monitor jobs, stages, tasks, and memory usage.

#### 🚧 Other Limitations:

Issue	Explanation
Inefficient for Iterative Algorithms	Each iteration writes to disk — very slow for ML or graph processing.
No Built-in Libraries	No native support for ML, SQL, or streaming — everything must be built manually.
Hard to Integrate	Not easily pluggable with modern tools like Kafka, MLflow, or Delta Lake.

### Apache Spark Ecosystem

🔧 Component	📋 Description	☑ Use Cases
<b>Spark Core</b>	The <b>engine</b> of Spark. Handles task scheduling, memory management, fault tolerance, and basic I/O.	Underlies all other components
<b>Spark SQL</b>	Module for working with <b>structured data</b> using <b>SQL</b> , <b>DataFrames</b> , and <b>Datasets</b> .	Data analysis, ETL, querying big data with SQL
<b>Spark Streaming</b>	Enables <b>real-time data processing</b> using micro-batching.	Live log processing, fraud detection, IoT pipelines
<b>Structured Streaming</b>	High-level streaming API built on top of Spark SQL (newer than Spark Streaming).	Real-time dashboards, streaming joins, event-time processing

<b>MLlib</b>	Built-in <b>machine learning library</b> in Spark. Supports algorithms like regression, clustering, classification, and recommendation.	ML at scale, churn prediction, product recommendations
<b>GraphX</b>	API for <b>graph-parallel computation</b> (nodes + edges).	Social network analysis, recommendation graphs
<b>Delta Lake (optional, external)</b>	<b>ACID-compliant data lake</b> layer built on top of Spark and Parquet.	Unified batch + streaming, time-travel, data reliability
<b>SparkR</b>	R interface for Spark. Allows data scientists familiar with R to use Spark's power.	Statistical analysis at scale
<b>PySpark</b>	Python API for Spark. One of the most popular interfaces.	Big data processing and ML in Python
<b>Spark on Kubernetes</b>	Deployment of Spark on <b>Kubernetes</b> for containerized environments.	Cloud-native big data pipelines
<b>Spark GraphFrames</b>	Enhanced graph processing with <b>DataFrame API</b> .	Graph computations with SQL-like syntax

### Optional Tools Often Used with Spark

Tool	Purpose
<b>Apache Kafka</b>	Real-time data ingestion (often paired with Spark Streaming)
<b>Apache Hive</b>	Data warehouse system; Spark SQL can query Hive tables
<b>Apache HBase / Cassandra</b>	NoSQL databases Spark can read/write to
<b>Airflow / Luigi / Prefect</b>	Orchestrate Spark jobs in pipelines
<b>Jupyter Notebooks</b>	Interactive environment for PySpark development
<b>Databricks</b>	Managed platform built around Spark, with notebook interface, auto-scaling, ML support

## Resilient Distributed Datasets

Resilient Distributed Datasets (RDD) is a fundamental data structure of Spark. It is an immutable distributed collection of objects. Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster. RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.

Formally, an RDD is a read-only, partitioned collection of records. RDDs can be created through deterministic operations on either data on stable storage or other RDDs. RDD is a fault-tolerant collection of elements that can be operated on in parallel.

There are two ways to create RDDs – **parallelizing** an existing collection in your driver program, or **referencing a dataset** in an external storage system, such as a shared file system, HDFS, HBase, or any data source offering a Hadoop Input Format.

Spark makes use of the concept of RDD to achieve faster and efficient MapReduce operations. Let us first discuss how MapReduce operations take place and why they are not so efficient.

### Data Sharing is Slow in MapReduce

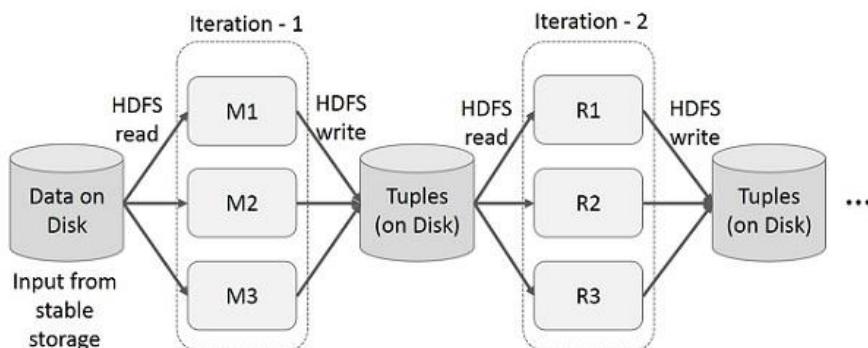
MapReduce is widely adopted for processing and generating large datasets with a parallel, distributed algorithm on a cluster. It allows users to write parallel computations, using a set of high-level operators, without having to worry about work distribution and fault tolerance.

Unfortunately, in most current frameworks, the only way to reuse data between computations (Ex – between two MapReduce jobs) is to write it to an external stable storage system (Ex – HDFS). Although this framework provides numerous abstractions for accessing a clusters computational resources, users still want more.

Both **Iterative** and **Interactive** applications require faster data sharing across parallel jobs. Data sharing is slow in MapReduce due to **replication**, **serialization**, and **disk IO**. Regarding storage system, most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations.

### Iterative Operations on MapReduce

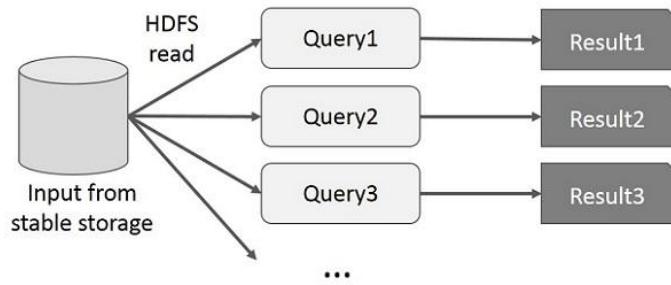
Reuse intermediate results across multiple computations in multi-stage applications. The following illustration explains how the current framework works, while doing the iterative operations on MapReduce. This incurs substantial overheads due to data replication, disk I/O, and serialization, which makes the system slow.



## Interactive Operations on MapReduce

User runs ad-hoc queries on the same subset of data. Each query will do the disk I/O on the stable storage, which can dominate application execution time.

The following illustration explains how the current framework works while doing the interactive queries on MapReduce.



## Data Sharing using Spark RDD

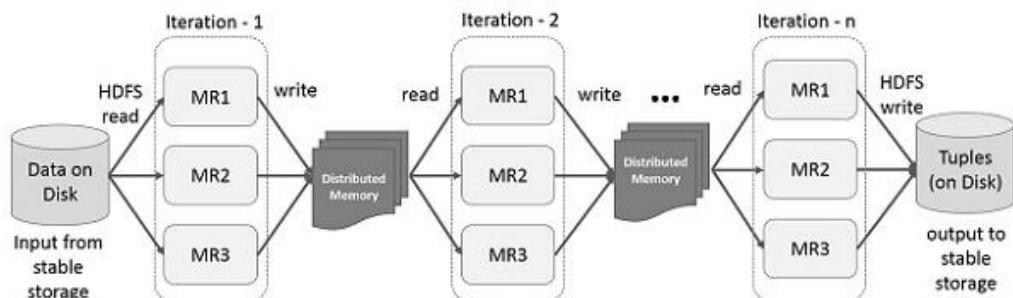
Data sharing is slow in MapReduce due to **replication**, **serialization**, and **disk IO**. Most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations.

Recognizing this problem, researchers developed a specialized framework called Apache Spark. The key idea of spark is **Resilient Distributed Datasets (RDD)**; it supports in-memory processing computation. This means, it stores the state of memory as an object across the jobs and the object is sharable between those jobs. Data sharing in memory is 10 to 100 times faster than network and Disk.

## Iterative Operations on Spark RDD

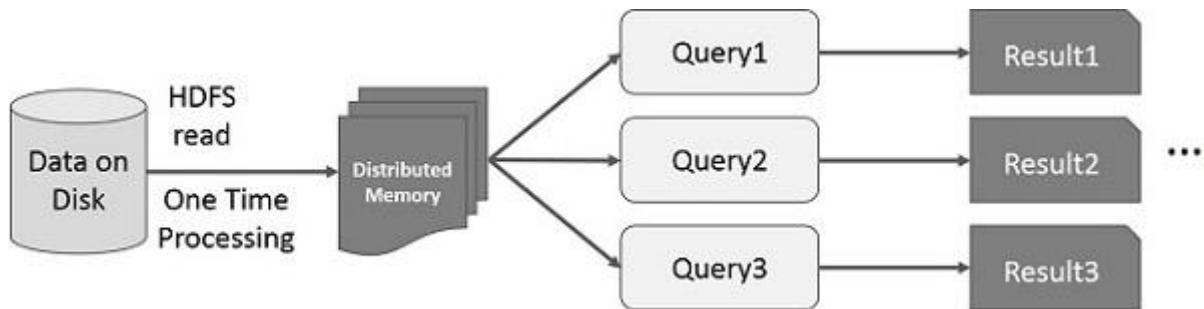
The illustration given below shows the iterative operations on Spark RDD. It will store intermediate results in a distributed memory instead of Stable storage (Disk) and make the system faster.

**Note** – If the Distributed memory (RAM) is not sufficient to store intermediate results (State of the JOB), then it will store those results on the disk.



## Interactive Operations on Spark RDD

This illustration shows interactive operations on Spark RDD. If different queries are run on the same set of data repeatedly, this particular data can be kept in memory for better execution times.



By default, each transformed RDD may be recomputed each time you run an action on it. However, you may also **persist** an RDD in memory, in which case Spark will keep the elements around on the cluster for much faster access, the next time you query it. There is also support for persisting RDDs on disk, or replicated across multiple nodes.

# Spark Architecture Deep Dive

## Spark Components: Driver, Executors, Cluster Manager

Apache Spark follows a **master-slave architecture**. Understanding the role of each component helps you visualize how your code turns into distributed execution.

### Key Components:

#### 1. Driver Program

- The **brain of a Spark application**.
- Runs on the machine where the job is submitted.
- Responsible for:
  - Maintaining **SparkContext** (entry point for the program)
  - Creating the **DAG (Directed Acyclic Graph)**
  - Requesting resources from Cluster Manager
  - Scheduling tasks and monitoring execution.

#### 2. Executors

- **Workers** that actually run your code.
- Each executor:
  - Executes assigned tasks.
  - Stores **RDDs** or DataFrames in memory (caching).
  - Sends status info back to the driver.
- Typically **one executor per worker node**.

#### 3. Cluster Manager

- Assigns and manages **resources** (CPU, memory).
- Spark can work with:
  - **Standalone** (built-in)
  - **YARN** (Hadoop ecosystem)
  - **Kubernetes**
  - **Mesos**



Driver → Cluster Manager → Executors → Execute Tasks

## Understanding DAG (Directed Acyclic Graph)

When you write Spark code, you don't execute it immediately. Spark **builds a DAG** behind the scenes.

### 💡 What is a DAG?

- DAG = **Directed Acyclic Graph**
- Represents the **sequence of computations** (transformations) your data will go through.
- "Directed" → flow has a direction (step-by-step)
- "Acyclic" → no loops or cycles

### Example

```
df = spark.read.csv("data.csv")
df2 = df.filter(df["age"] > 25)
df3 = df2.groupBy("city").count()
df3.show()
```

💡 Spark creates a DAG like this:

Read CSV → Filter Age > 25 → Group by City → Count → Show

Only when `.show()` is called (an **action**), Spark submits the DAG for execution.

## Logical Plan vs Physical Plan

When the DAG is built, Spark doesn't directly jump to execution. It **optimizes your queries** through multiple stages.

### Logical Plan

- Unoptimized representation of your code.
- Derived from the syntax of your operations (SQL or DataFrame).
- E.g., you ask for a filter → join → groupBy

### Optimized Logical Plan

- Spark applies **Catalyst Optimizer** rules.
- Removes unnecessary steps, reorders filters, etc.

### Physical Plan

- Final **execution strategy**.
- Converts logical plan into **RDD operations**, which are distributed and scheduled on executors.
- Spark selects the **most efficient plan** (cost-based).

You can inspect the plans using:

```
df.explain()
```

## Execution Flow & Fault Tolerance

### Execution Flow

1. **User Code** (DataFrame or RDD) is written.
2. Spark creates a **Logical Plan** → Optimizes it.
3. Builds **DAG** of stages and tasks.
4. DAG is submitted to **Cluster Manager**.
5. Tasks are distributed to **Executors**.
6. **Driver monitors** execution and collects results.

### Stages and Tasks

- Spark splits a job into **stages**, based on **shuffle boundaries**.
- Each stage is divided into **tasks**, which are run in parallel across executors.

## Fault Tolerance in Spark

Spark provides **fault tolerance** through **lineage** and **RDD immutability**.

### **Lineage:**

- Every RDD/DataFrame remembers how it was created (from which parent RDDs).
- If an executor fails, Spark can **recompute lost data** from previous transformations.

### **Example:**

If  $\text{RDD3} = \text{RDD1}.map().filter()$  and  $\text{RDD3}$  is lost due to failure,  
→ Spark will recompute from  $\text{RDD1}$ .

 No need to save intermediate results to disk — this is faster than MapReduce!

## Summary:

Component	Purpose
<b>Driver</b>	Orchestrates the application
<b>Executors</b>	Perform computation and store results
<b>Cluster Manager</b>	Allocates resources to Spark
<b>DAG</b>	Represents job as a sequence of stages/tasks
<b>Logical Plan</b>	User code interpreted (not yet optimized)
<b>Physical Plan</b>	Actual execution plan (after optimization)
<b>Lineage</b>	Enables fault tolerance via recomputation

## RDD – Resilient Distributed Dataset

### What is an RDD?

An **RDD** is the **fundamental data structure** in Apache Spark.

It's an:

**Immutable, distributed** collection of objects that can be processed in parallel.

Think of it like a big array, spread out over many machines.

### Features of RDD:

Feature	Description
<b>Resilient</b>	Can recover from failure using lineage (rebuild from original data)
<b>Distributed</b>	Data is spread across nodes in a cluster
<b>Immutable</b>	Once created, RDDs can't be changed; you create new ones from transformations
<b>In-memory</b>	Supports caching in memory for faster processing
<b>Typed</b>	RDDs hold elements of a specific type (e.g., integers, tuples)

### How to create an RDD?

```
# From a collection
rdd = sc.parallelize([1, 2, 3, 4, 5])

# From external storage
rdd2 = sc.textFile("path/to/file.txt")
```

### Why RDDs?

- Fine-grained control over data and operations
- Fault tolerance through lineage
- Best for low-level transformations or when you need full control

Modern Spark uses **DataFrames** more often, but RDDs are still important for custom, complex workflows.

## Transformations vs Actions

### Transformations

Transformations are **operations that define a new RDD** from an existing one.

They are **lazy** — they don't run until an action is called. [-like collecting items to shopping cart]

#### Common Transformations:

- map(func) – apply a function to each element
- filter(func) – filter elements based on condition
- flatMap(func) – similar to map, but flattens the result
- reduceByKey(func) – reduce values for each key
- union(), join(), distinct()

```
rdd = sc.parallelize([1, 2, 3, 4])
```

```
rdd2 = rdd.map(lambda x: x * 2) # Transformation
```

## Actions

Actions **trigger the actual execution** of transformations and return results to the driver or save to storage. [Submitted tasks – like checkout]

### Common Actions:

- collect() – fetch all elements to driver
- count() – count number of elements
- take(n) – take first n elements
- reduce(func) – reduce elements using a function
- saveAsTextFile() – save to disk

```
result = rdd2.collect() # Action → triggers execution
```

## Lazy Evaluation in Spark

### What is Lazy Evaluation?

In Spark, **nothing is executed immediately** when you write code — even transformations like map() or filter().

Spark **waits until an action is called**, and then:

- Builds a **DAG (Directed Acyclic Graph)** of all the steps
- Optimizes the DAG
- Executes it in the most efficient way

## Benefits of Lazy Evaluation:

Benefit	Why it matters
<b>Optimization</b>	Spark optimizes the entire execution pipeline before running
<b>Efficiency</b>	Avoids unnecessary computations or intermediate storage
<b>Fault Tolerance</b>	Keeps track of lineage for recomputation in case of failure

## Example of Lazy Evaluation:

```
rdd = sc.parallelize([1, 2, 3, 4])
rdd2 = rdd.map(lambda x: x + 1)    # Transformation
rdd3 = rdd2.filter(lambda x: x > 2)  # Another transformation

# Nothing has happened yet...

print(rdd3.collect())  # Now Spark executes the DAG
Until .collect() is called, Spark just records the steps, not runs them.
```

## Summary Table

Concept	Description
<b>RDD</b>	Immutable, distributed collection of data
<b>Transformations</b>	Define new RDDs, <b>lazy</b> , not executed immediately
<b>Actions</b>	Trigger actual computation and return results

Feature / Aspect	Transformations	Actions
<b>Definition</b>	Operations that <b>create a new RDD/DataFrame</b>	Operations that <b>trigger computation</b> and return result
<b>Execution</b>	<b>Lazy</b> – not executed immediately	<b>Eager</b> – executed immediately
<b>Return Type</b>	Returns <b>another RDD/DataFrame</b>	Returns <b>a value</b> to the driver or writes to storage
<b>Purpose</b>	To define the <b>logical steps</b> of the computation	To <b>initiate</b> and complete the computation
<b>Triggers DAG Execution</b>	No	Yes
<b>Chaining</b>	Can be <b>chained</b> together	Usually the <b>final step</b> in a pipeline
<b>Examples (RDD)</b>	map(), filter(), flatMap(), union(), reduceByKey()	collect(), count(), take(), saveAsTextFile()
<b>Examples (DataFrame)</b>	select(), filter(), groupBy(), withColumn()	show(), count(), write(), collect()
<b>Optimization</b>	Recorded and optimized as part of <b>DAG</b>	Not optimized – just executes what was planned
<b>Used for</b>	<b>Building the pipeline</b>	<b>Getting the result</b>

## Let's Dive Deeper: What Happens Under the Hood?

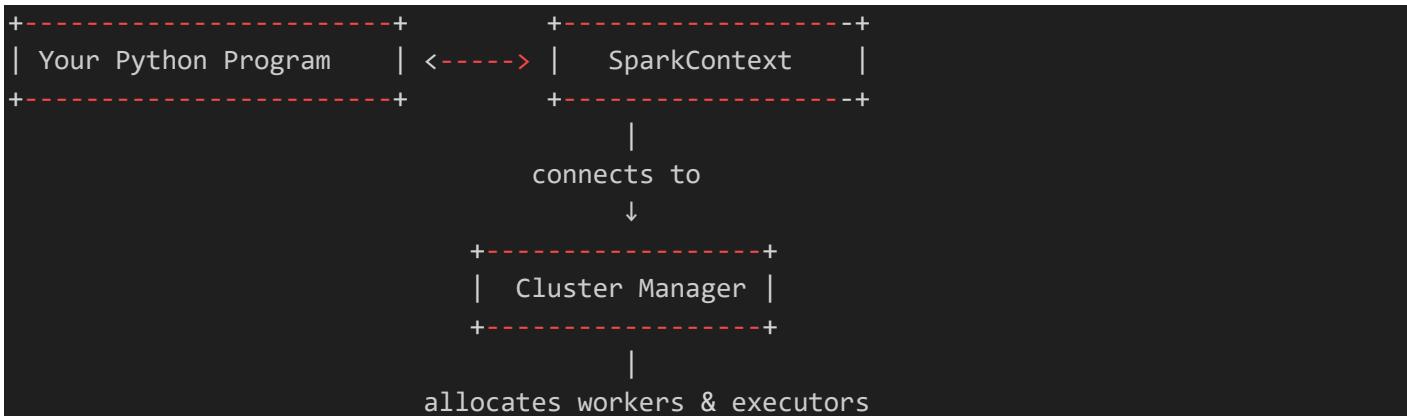
Imagine you're a **data engineer** writing a Spark program

```
rdd = sc.parallelize([1, 2, 3, 4, 5])
rdd2 = rdd.map(lambda x: x * 2)
rdd3 = rdd2.filter(lambda x: x > 5)
result = rdd3.collect()
```

## Step-by-Step Breakdown

### 1. SparkContext Initialization

- When you write `sc.parallelize(...)`, Spark creates the **SparkContext** object.
- It connects to the **Cluster Manager** (e.g., Standalone, YARN, Kubernetes).
- The Cluster Manager gives Spark **executors** (worker processes) on different nodes.



## 2. sc.parallelize([1,2,3,4,5])

- Creates an RDD from a Python list.
- Spark divides the data into **partitions**.

Let's say 2 partitions: [1,2,3] and [4,5].

- No computation happens yet.
- Spark **records this RDD lineage**:
- `RDD1 = parallelize(data)`

## 3. rdd.map(lambda x: x \* 2)

- You're telling Spark: "I want to **transform** each element by doubling it."
- But Spark doesn't do anything yet — it adds this transformation to a **DAG** (Directed Acyclic Graph).
- It tracks:
- `RDD2 = RDD1.map(lambda x: x * 2)`

## 4. rdd2.filter(lambda x: x > 5)

- Again, you're just *declaring* another transformation.
- Spark adds another node to the DAG:
- `RDD3 = RDD2.filter(lambda x: x > 5)`

**Still no data has been processed yet!**

## 5. collect() – The Trigger

- This is an **action**.
- It says: "Okay Spark, I want the final result — go do all the work now."

- Spark finally:
  - Analyzes the DAG ( $\text{RDD1} \rightarrow \text{RDD2} \rightarrow \text{RDD3}$ )
  - Splits it into **stages** (based on shuffles)
  - Converts each stage into **tasks**
  - Sends those tasks to executors

DAG:  $\text{RDD1} \rightarrow \text{map} \rightarrow \text{RDD2} \rightarrow \text{filter} \rightarrow \text{RDD3} \rightarrow \text{collect}$

## 6. Execution Plan & DAG Scheduler

- Spark's **DAG Scheduler**:
  - Figures out what tasks are needed
  - Sends them to available executors
- **Executors run the transformations on partitions**
  - $[1,2,3] \rightarrow \text{map} \rightarrow [2,4,6] \rightarrow \text{filter} \rightarrow [6]$
  - $[4,5] \rightarrow \text{map} \rightarrow [8,10] \rightarrow \text{filter} \rightarrow [8,10]$

## 7. Final Result

- Spark gathers the filtered results from all partitions:  $[6, 8, 10]$
- Returns them to your **Driver program** via `collect()`

### ⚠ Lazy Evaluation is the Secret Sauce

Without lazy evaluation:

- Every `.map()` or `.filter()` would trigger a new job (expensive).
- Spark would not be able to optimize your job pipeline.

But with **lazy evaluation**:

- Spark **analyzes your whole job pipeline** at once
- It removes unnecessary operations (called *pipelining*)
- It minimizes data shuffles (which are expensive in distributed systems)

## **Visual Flow Summary:**

You write code:

```
RDD1 = sc.parallelize(...)
```

```
RDD2 = RDD1.map(...)
```

```
RDD3 = RDD2.filter(...)
```

```
Result = RDD3.collect()
```

Spark builds:

DAG: RDD1 → RDD2 → RDD3 → collect()

Then:

→ Schedules Tasks → Runs on Executors → Gathers Result → Sends to Driver

## **Why This Matters for Data Engineers:**

- Helps you **debug and optimize** pipelines.
- Avoid using `collect()` on huge datasets — can cause driver crashes.
- Think in **pipelines**, not steps — Spark will optimize them for you.
- Know where **shuffles** happen — they cause performance hits.
- Use `.persist()` or `.cache()` when intermediate results are reused.

## Core Properties of RDDs in Spark

RDDs are the **low-level, core abstraction** of Spark that enable **fault-tolerant, distributed computation** over large datasets.

### 1. Immutability

- Once an RDD is created, it cannot be changed.
- You can only create **new RDDs** by applying transformations to existing ones.

#### Why?

- Immutability helps **simplify fault recovery** and makes execution predictable.
- Spark tracks all transformations (lineage) to **recompute** lost data when needed.

```
rdd = sc.parallelize([1, 2, 3])
new_rdd = rdd.map(lambda x: x * 2) # rdd is unchanged
```

Original rdd remains untouched — Spark creates new\_rdd.

### 2. Lazy Evaluation

- Transformations (like map, filter) are **not executed immediately**.
- Instead, Spark builds a **logical plan (DAG)**.
- Execution is **triggered only when an action** (like collect(), count()) is called.

#### Benefit:

- **Optimizations:** Spark can analyze the entire computation pipeline and optimize it.
- **Efficiency:** No unnecessary computations or shuffles happen early.

```
rdd2 = rdd.map(lambda x: x + 1) # No execution yet
print(rdd2.collect())           # Now it runs
```

### 3. Partitioned (Distributed)

- RDDs are **divided into partitions**, and each partition is **processed in parallel** on a different node in the cluster.

#### Why it matters:

- Enables **parallelism** and **scalability**.
- Partitions can be **recomputed independently**, enabling fault tolerance.

```
rdd.getNumPartitions() # Check number of partitions
```

- Spark tries to **co-locate** data and computation for speed.

Advanced: You can **repartition** or **coalesce** RDDs to control parallelism.

## 4. Fundamental Data Structure of Spark

- RDD is the **foundation** for higher-level APIs like:
  - **DataFrames** (structured data with schema)
  - **Datasets** (typed objects in Scala/Java)
- Provides **fine-grained control** over transformations and actions.
- Ideal for **custom, low-level** data processing tasks (e.g., complex ETL, legacy systems).

### Bonus: Other Important Properties

Property	Description
<b>Typed</b>	RDD holds elements of specific type (e.g., Int, String, Tuple)
<b>Fault Tolerant</b>	Can recompute lost partitions using lineage (no replication needed)
<b>In-Memory</b>	Can cache/persist RDDs in memory for faster repeated access
<b>Composable</b>	Transformations can be <b>chained together</b> to form complex pipelines

### Summary Table

Property	Meaning
<b>Immutability</b>	RDDs cannot be modified after creation
<b>Lazy Evaluation</b>	Transformations are recorded, not executed until an action is called
<b>Partitioned</b>	Data is split across partitions/nodes for parallel processing
<b>Fundamental</b>	RDD is the base abstraction in Spark for distributed data computation

## Processing in Spark: Local vs Distributed vs Parallel

### 1. Local Processing (Single Machine Mode)

What it means:

- Spark runs **on your laptop or a single machine**.
- Still uses **parallelism** across **multiple CPU cores**.

Example:

```
spark = SparkSession.builder.master("local[*]").getOrCreate()
```

- "local[\*]" uses all available cores.
- Even though you're not on a cluster, Spark **splits data into partitions** and processes them in **parallel threads**.

### 2. Parallel Processing

What it means:

- Spark breaks data into **partitions**, and each partition is processed **independently and simultaneously**.
- This happens **within a node (local parallelism)** or **across multiple nodes (distributed parallelism)**.

Example:

```
rdd = sc.parallelize(range(10), numSlices=4)
print(rdd.getNumPartitions()) # 4
```

- Spark will create **4 partitions**, which can be processed **in parallel**.
- Each partition will contain a chunk of the data:
- Partition 0: [0,1]
- Partition 1: [2,3]
- Partition 2: [4,5]
- Partition 3: [6,7,8,9]

### 3. Distributed Processing (Cluster Mode)

#### What it means

- Spark runs on **multiple machines (nodes)** in a **cluster**.
- Each node has **executors**, which:
  - Process data partitions
  - Store intermediate data
  - Report back to the driver

 Cluster managers like **YARN**, **Kubernetes**, or **Standalone** coordinate resource allocation.

#### What is a Partition in Spark?

A **partition** is:

A logical chunk of data processed as a **single unit of parallel work**.

#### Key Points:

- Spark **splits RDDs and DataFrames into partitions** automatically.
- Each **partition is processed independently** by an executor/task.
- More partitions = better **parallelism** (up to a point).

#### How Partitions Work Internally

Step	What Happens
1	Spark reads data from storage (e.g., HDFS, S3, local file)
2	It splits the file into <b>blocks/partitions</b>
3	Each partition becomes a <b>task</b>
4	Tasks are <b>assigned to executors</b> to run in parallel
5	Executors perform operations (map, filter, reduce) on each partition
6	Final results are shuffled, collected, or saved

## Partitioning Strategy

### Default Partition Count:

- `textFile()` → default = number of HDFS blocks (128MB each)
- `parallelize()` → default = depends on environment (e.g., `local[*]` uses CPU cores)

### You can control it with:

```
rdd = sc.textFile("data.txt", minPartitions=10)
rdd = rdd.repartition(6)
rdd = rdd.coalesce(2) # reduce without shuffle
```

## Summary Table

Concept	Description
<b>Local</b>	Run on single machine with multithreading
<b>Parallel</b>	Multiple tasks/processes work at the same time
<b>Distributed</b>	Work is distributed across a cluster of machines
<b>Partition</b>	Smallest unit of data processed in parallel; drives Spark's scalability

## Examples Using `getNumPartitions()` in Spark

```
rdd.getNumPartitions()
```

This method returns the **number of partitions** an RDD has — which is critical to understand for **parallelism and performance tuning**.

### Case 1: Default Parallelization (Local Mode)

```
rdd = sc.parallelize(range(100))
print("Partitions:", rdd.getNumPartitions())
```

**What Happens:**

- Spark automatically chooses the number of partitions.
- On local mode, this is usually equal to the **number of CPU cores**.

Example Output:

```
Partitions: 8 # On an 8-core machine
```

### Case 2: Specify Partitions with `numSlices`

```
rdd = sc.parallelize(range(100), numSlices=5)
print("Partitions:", rdd.getNumPartitions())
```

**What Happens:**

- You explicitly request 5 partitions.
- Spark will divide the 100 elements roughly equally into 5 parts.

Output:

```
Partitions: 5
```

### Case 3: Reading File from HDFS or Local

```
rdd = sc.textFile("large_file.txt")
print("Partitions:", rdd.getNumPartitions())
```

**What Happens:**

- Spark determines the number of partitions based on:
  - File size
  - Default HDFS block size (usually 128MB)
- A 512MB file might give you 4 partitions.

Output (depends on file size):

```
Partitions: 4
```

## Case 4: Repartition for More Parallelism

```
rdd = sc.parallelize(range(10), 2)
print("Before repartition:", rdd.getNumPartitions())

rdd2 = rdd.repartition(4)
print("After repartition:", rdd2.getNumPartitions())
```

### What Happens

- Repartition **increases** the number of partitions using a **full shuffle**.
- Useful before expensive operations like **joins** or **groupBy**.

Output:

Before repartition: 2

After repartition: 4

## Case 5: Coalesce to Reduce Partitions

```
rdd = sc.parallelize(range(100), 8)
print("Before coalesce:", rdd.getNumPartitions())

rdd2 = rdd.coalesce(2)
print("After coalesce:", rdd2.getNumPartitions())
```

### What Happens

- Coalesce reduces partitions **without a full shuffle** (efficient).
- Useful before **writing small output files** (e.g., .coalesce(1) to write 1 file).

Output:

Before coalesce: 8

After coalesce: 2

## Case 6: After GroupByKey or Join (Watch the Partition Count)

```
rdd = sc.parallelize([(1, 'a'), (2, 'b'), (3, 'c')], 2)
grouped = rdd.groupByKey()
print("Partitions after groupByKey:", grouped.getNumPartitions())
```

### 💡 What Happens:

- **Wide transformations** like groupByKey, reduceByKey, or join trigger **shuffles**.
- The partition count might increase or remain same depending on config.

Output:

Partitions after groupByKey: 2

## Why Use `getNumPartitions()`?

- To check **how well your data is distributed**
- To ensure **enough parallelism** before expensive operations
- To avoid **small files problem** during writes (use `.coalesce()`)
- To debug performance issues (too few or too many partitions)

# DataFrames, Datasets, and Optimizations

## What are DataFrames and Datasets?

❖ Concept ❁ Think of it like...	
<b>DataFrame</b>	A <b>table of data</b> with rows and columns (like an Excel sheet or SQL table)
<b>Dataset</b>	A <b>typed table</b> – you know what kind of data is in each column ahead of time

## Real-World Example

Let's say you have a CSV file of students:

name	age	grade
Alice	14	A
Bob	15	B

- **DataFrame** lets you read this file and treat it like a table.
- **Dataset** lets you read the file and know: “age is an integer”, “grade is a string”, etc. (but only in Scala/Java).

## DataFrame vs Dataset — Simple Comparison

Feature	DataFrame	Dataset
Available in Python?	Yes	No
Easy to use?	Yes	Medium (only for Scala/Java)
Type safety (error check before running)	No	Yes
Best for:	Quick data tasks & SQL-style queries	When you want full control of data types

## When Should You Use Each?

Situation	Use...
You're writing in Python	DataFrame
You need SQL-like commands (select, filter)	DataFrame
You're writing in Scala and want type safety	Dataset
You want fast development	DataFrame

## Spark's Superpowers: Optimizations

Spark has two engines under the hood that make it fast: **Catalyst** and **Tungsten**.

### 1. Catalyst Optimizer – Smart Query Planner

Think of it as Spark's brain. You write simple code, and it figures out the best way to run it.

#### Example:

```
df.filter(df.age > 18).select("name")
```

 Catalyst will:

- Move the filter close to the data source
- Skip columns you're not using
- Choose the fastest way to get the result

### 2. Tungsten Engine – Speed and Memory Booster

Tungsten is like Spark's **engine** that:

- Speeds things up using **binary formats**
- Uses less memory
- Generates **optimized machine code** at runtime

This is what makes Spark faster than many traditional tools.

## Summary Table

❖ Feature	DataFrame	Dataset
<b>Works in Python</b>	✓ Yes	✗ No (Scala/Java only)
<b>Simple to Use</b>	✓ Beginner-friendly	🟡 Requires more coding
<b>SQL-like Support</b>	✓ Yes	✓ Yes
<b>Type Safety</b>	✗ No	✓ Yes
<b>Optimized by Catalyst</b>	✓ Yes	✓ Yes
<b>Uses Tungsten Engine</b>	✓ Yes	✓ Yes

# Spark Programming - Python (PySpark)

## What is PySpark?

**PySpark** is the Python API for Apache Spark.

It lets you write Spark applications using Python — a language that's easy to learn and widely used in data engineering and data science.

Think of PySpark as the bridge between **big data processing** and **Python programming**.

## Why Use PySpark?

 Benefit	 Explanation
<b>Easy to Learn</b>	Python syntax is simple and readable
<b>Powerful &amp; Scalable</b>	You can process <b>huge datasets</b> across many computers
<b>Integrates with Pandas</b>	Convert between <b>Pandas DataFrames</b> and <b>Spark DataFrames</b>
<b>Built-in ML &amp; SQL</b>	Use <b>Spark MLlib</b> for machine learning and <b>Spark SQL</b> for queries
<b>Part of Spark Ecosystem</b>	Same engine used in Scala/Java — just controlled via Python

## Basic PySpark Setup

### Step 1: Install PySpark

```
pip install pyspark
```

### Step 2: Start SparkSession

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("MyFirstPySparkApp") \
    .getOrCreate()
```

### Step 3: Read a CSV file

```
df = spark.read.csv("students.csv", header=True, inferSchema=True)
df.show()
```

## Common PySpark Operations

Operation	PySpark Code Example	What It Does
<b>Read CSV</b>	<code>spark.read.csv("file.csv", header=True)</code>	Loads data as DataFrame
<b>Show Data</b>	<code>df.show()</code>	Prints first 20 rows
<b>Filter Rows</b>	<code>df.filter(df.age &gt; 18)</code>	Filters rows where age > 18
<b>Select Cols</b>	<code>df.select("name", "grade")</code>	Picks specific columns
<b>Group By</b>	<code>df.groupBy("grade").count()</code>	Counts students in each grade
<b>Write Data</b>	<code>df.write.csv("output_folder")</code>	Writes DataFrame to CSV

## PySpark vs Pandas

Feature	Pandas	PySpark
<b>Works on</b>	Single machine	Distributed (multiple machines)
<b>Speed</b>	Fast for small data	Faster for big data
<b>Memory usage</b>	May crash with large files	Handles large files with ease
<b>Syntax</b>	Pythonic, very similar	Slightly more verbose

## PySpark Limitations

- A bit slower than native Scala/Java APIs
- Requires Java (JDK) and Spark installation under the hood
- Can't do everything Pandas does (especially in plotting/visuals)

## Summary

- **PySpark** is the best way to use **Apache Spark with Python**
- Ideal for **big data**, **ETL**, and **ML** workflows
- Supports most Spark features: **DataFrames**, **SQL**, **Mlib**, **Streaming**
- Great for **data engineers**, **data scientists**, and **analysts**

# SparkSession in PySpark

## What is SparkSession?

**SparkSession** is the **entry point** to programming with Apache Spark using DataFrames and Datasets.

It's like the "main door" into the Spark world — everything you do in PySpark starts from a SparkSession.

In earlier versions of Spark (before v2.0), you had to use SparkContext and SQLContext separately. But now, **SparkSession combines them all** into one object.

## Why is SparkSession Important?

 Purpose	 What it Does
Entry point to Spark	Creates the core connection to the Spark engine
Access to DataFrame API	Enables loading, transforming, and saving data using DataFrames
Integration with SQL	Allows you to write SQL queries directly on your data
Manages Spark Configuration	Sets things like memory size, number of cores, and Spark app name

## How to Create a SparkSession

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("MyApp") \
    .master("local[*]") \
    .getOrCreate()
```

### Explanation:

- builder → starts the configuration
- appName("MyApp") → names your Spark application
- master("local[\*]") → runs Spark locally using all CPU cores ([\*])
- getOrCreate() → creates a new session or returns an existing one

## Example: Using SparkSession

```
from pyspark.sql import SparkSession

# Step 1: Create SparkSession
spark = SparkSession.builder \
    .appName("StudentGrades") \
    .getOrCreate()

# Step 2: Read data
df = spark.read.csv("students.csv", header=True, inferSchema=True)

# Step 3: Show data
df.show()
```

## Useful Methods in SparkSession

Method	Purpose
<b>spark.read</b>	Reads data (CSV, JSON, Parquet, etc.)
<b>spark.createDataFrame()</b>	Converts Python lists or Pandas DataFrames
<b>spark.sql("SELECT ...")</b>	Runs SQL queries on Spark tables
<b>spark.catalog.listTables()</b>	Shows all tables registered with Spark
<b>spark.stop()</b>	Stops the session and releases resources

## Real-Life Use Cases

### 1. ETL Pipelines

Load large files, transform data, and write to databases or cloud storage.

### 2. Data Analysis

Filter, group, and join massive datasets using SQL or DataFrame API.

### 3. Machine Learning

Use spark.ml with SparkSession to train models on large-scale data.

### 4. Stream Processing

Use SparkSession.readStream to process real-time data streams.

## SparkSession with Pandas

You can convert between Spark and Pandas:

```
# Convert to Pandas
pdf = df.toPandas()

# Convert from Pandas to Spark
sdf = spark.createDataFrame(pdf)
```

## Summary Table

◊ Feature	💡 Description
<b>What it is</b>	Main entry point to Spark
<b>Introduced in</b>	Spark 2.0+
<b>Replaces</b>	SparkContext, SQLContext, HiveContext
<b>Core use</b>	Create/read/write DataFrames, run SQL
<b>Typical methods</b>	.read, .sql, .createDataFrame(), .stop()

## Stop the SparkSession (Best Practice)

```
spark.stop()
```

This ensures all Spark resources are released after your job finishes.

## spark.sparkContext.parallelize()

### What is it?

spark.sparkContext.parallelize() is a method in PySpark used to **create an RDD (Resilient Distributed Dataset)** from a **local Python collection** (like a list or range).

It's like saying: "Hey Spark, here's a list — please distribute it across the cluster so we can process it in parallel."

### When Should You Use It?

- When you want to **create an RDD manually** for testing or learning
- When you have **small local data** and want to parallelize it into partitions
- When you're **starting from Python data** instead of loading from a file

### Syntax

```
rdd = spark.sparkContext.parallelize(data, numSlices)
```

- data: Any Python collection (e.g., list, tuple, range)
- numSlices (optional): Number of **partitions** you want to split the data into

### Simple Example

```
data = [1, 2, 3, 4, 5]
rdd = spark.sparkContext.parallelize(data)

print(rdd.collect())
```

#### Output:

```
[1, 2, 3, 4, 5]
```

### Example with Partitions

```
rdd = spark.sparkContext.parallelize([10, 20, 30, 40, 50], 3)
print("Partitions:", rdd.getNumPartitions())
```

#### Output:

```
Partitions: 3
```

This divides the data across **3 partitions**, enabling **parallel execution** on multiple cores or nodes.

## Visualizing Partitioning

```
rdd.glom().collect()
```

This shows how the data is divided into partitions:

### Output Example:

```
[[10, 20], [30, 40], [50]]
```

### Common RDD Actions After parallelize()

Action	Description
<code>rdd.collect()</code>	Returns all elements as a list
<code>rdd.count()</code>	Counts the number of elements
<code>rdd.sum()</code>	Sums all the elements
<code>rdd.map()</code>	Applies a function to each element
<code>rdd.filter()</code>	Filters elements based on a condition

### Important Notes

- Don't use `parallelize()` for **large datasets** — use `spark.read` for those.
- It's mostly used in **experiments, learning, and test cases**.
- By default, Spark uses the number of **available cores** as default partition count.

### Summary Table

Feature	Description
<b>Purpose</b>	Create RDD from a local Python collection
<b>Input</b>	List, range, tuple, etc.
<b>Output</b>	RDD with elements distributed across partitions
<b>Best for</b>	Small data, testing, and local simulations
<b>Common after-ops</b>	<code>map, filter, collect, count, glom</code>

# RDD Operations

## Set Up PySpark and Pandas

```
import pandas as pd
from pyspark.sql import SparkSession

# Start Spark session
spark = SparkSession.builder \
    .appName("RDD Operations Example") \
    .getOrCreate()
```

## Step 1: Create a Sample Pandas DataFrame

```
# Create a small DataFrame
df = pd.DataFrame({
    'customer_id': [1, 2, 3, 4, 5, 6],
    'status': ['active', 'inactive', 'active', 'inactive', 'active', 'active'],
    'purchase': [300, 0, 450, 0, 150, 600]
})
print(df)
```

### Output

```
customer_id  status  purchase
0           1  active     300
1           2 inactive      0
2           3 active     450
3           4 inactive      0
4           5 active     150
5           6 active     600
```

## Step 2: Convert to RDD using parallelize()

```
rdd = spark.sparkContext.parallelize(df.to_dict("records"))
```

This creates an **RDD of dictionaries**, where each element is like:

```
{'customer_id': 1, 'status': 'active', 'purchase': 300}
```

## Step 3: Basic Actions and Transformations

**.collect() → Action: returns all elements to driver**

```
rdd.collect()
```

Collects and returns all rows (use carefully with large data).

### **.first() → Action: returns first element**

```
rdd.first()
```

Returns the first dictionary in the RDD. [Like head(1) in pandas]

### **.filter() → Transformation: filters based on a condition**

```
active_customers = rdd.filter(lambda row: row['status'] == 'active')
```

Lazy operation that doesn't compute anything yet.

### **.map() → Transformation: extract selected columns**

```
customer_purchases = active_customers.map(lambda row: (row['customer_id'],  
row['purchase']))
```

Now, each RDD element is a tuple like (customer\_id, purchase).

## **Combined So Far**

```
rdd = spark.sparkContext.parallelize(df.to_dict("records"))  
result = (  
    rdd.filter(lambda row: row['status'] == 'active')      # transformation  
        .map(lambda row: (row['customer_id'], row['purchase']))  # transformation  
)  
  
print(result.collect())  # action
```

### **Output**

[(1, 300), (3, 450), (5, 150), (6, 600)]

### **.distinct() → Transformation: remove duplicates**

```
distinct_status = rdd.map(lambda row: row['status']).distinct()  
print(distinct_status.collect())  
# Output: ['active', 'inactive']
```

### **.take(n) → Action: take first n elements**

```
print(rdd.take(3))
```

Like head() in Pandas.

## .reduceByKey() → Transformation

Used when data is in (key, value) format.

Let's compute **total purchases by status**.

```
rdd2 = rdd.map(lambda row: (row['status'], row['purchase']))
reduced = rdd2.reduceByKey(lambda a, b: a + b)

print(reduced.collect())
# Output: [('active', 1500), ('inactive', 0)]
```

**What's Happening?**

- **map**: turns data into key-value pairs: ('active', 300), etc.
- **reduceByKey**: groups by key (status) and reduces values using the function  $a + b$ .

reduceByKey is **distributed & efficient**, combines results locally before shuffling.

## .countByValue() → Action: frequency count

Count how many times each **status** appears:

```
status_count = rdd.map(lambda row: row['status']).countByValue()

print(dict(status_count))
# Output: {'active': 4, 'inactive': 2}
```

Returns a Python dictionary.

Internally aggregates values on executors before returning to driver.

## Putting It All Together — Full Example

```
rdd = spark.sparkContext.parallelize(df.to_dict("records"))

# Step-by-step pipeline
result = (
    rdd.filter(lambda row: row['status'] == 'active')                      # filter active users
        .map(lambda row: (row['customer_id'], row['purchase']))              # select 2 columns
)

# Print results
print("Filtered Customer Purchases:")
print(result.collect()) # Action

# Count distinct statuses
status_count = rdd.map(lambda row: row['status']).countByValue()
print("\nCount By Status:")
print(dict(status_count))

# Total purchase amount by status
status_sum = rdd.map(lambda row: (row['status'], row['purchase'])) \
    .reduceByKey(lambda a, b: a + b)
```

```

print("\nTotal Purchases By Status:")
print(status_sum.collect())

# Show distinct statuses
print("\nDistinct Statuses:")
print(rdd.map(lambda row: row['status']).distinct().collect())

```

```

+-----+
| Initial RDD (rdd) | <- parallelize(df.to_dict("records"))
+-----+
|
+-----+-----+-----+
|           |           |
| filter(row['status']=='active') | .map(row['status']) | .map(row['status'])
|           |           |           |
| .map((customer_id, purchase)) | .countByValue() | .distinct()
|           |           |           |
| .collect() (Action) | dict() (Action) | .collect()
| (Action)          |           |           |
|
|           |
| .map((status, purchase))
|           |
| .reduceByKey(lambda a,b: a+b)
|           |
| .collect() (Action)

```

## Summary Table

Operation	Type	Purpose
.map()	Transformation	Transform each element
.filter()	Transformation	Filter elements by condition
.distinct()	Transformation	Remove duplicates
.reduceByKey()	Transformation	Group + reduce values by key
.collect()	Action	Return all data to driver
.first()	Action	First element
.take(n)	Action	First n elements
.countByValue()	Action	Count frequencies of values

## More

### map() – Element-wise Transformation

map() transforms **each element** in an RDD **independently**. It applies a user-defined function and returns a new RDD of the same length.

```
rdd.map(lambda x: f(x))
```

#### Key Points

- **Transformation** (lazy)
- Operates element-wise
- Doesn't change the number of elements (unless function removes/expands)

#### Real-World Analogy

Imagine scanning a list of receipts and multiplying each value by 1.15 to apply tax—each receipt is independently transformed.

#### Example

```
rdd = spark.sparkContext.parallelize([1, 2, 3, 4])
rdd.map(lambda x: x * 2).collect()
# Output: [2, 4, 6, 8]
```

#### Use Cases

- Extracting fields from dictionaries/JSONs
- Converting data formats (e.g., strings to integers)
- Calculating derived values (e.g., price × quantity)

### filter() – Condition-Based Row Removal

#### Definition

filter() returns a new RDD by **keeping only those elements** that satisfy a boolean condition.

```
rdd.filter(lambda x: condition(x))
```

#### Key Points

- **Transformation**
- Only keeps elements that return True
- Doesn't modify the elements, just includes/excludes them

#### Real-World Analogy:

You have a stack of resumes and you only want those with "5+ years of experience." You **filter** out the rest.

## Example:

```
rdd = spark.sparkContext.parallelize([1, 2, 3, 4, 5])
rdd.filter(lambda x: x % 2 == 0).collect()
# Output: [2, 4]
```

## Use Cases

- Remove null/invalid records
- Filter customers by location, status, or spending
- Select logs above a severity threshold

## reduceByKey() – Group and Reduce (Optimized Shuffle)

### Definition

reduceByKey() combines values with the **same key** using a **reduce function**, such as addition or max. It **minimizes shuffling** by combining values **locally first**, then across nodes.

```
rdd.reduceByKey(lambda a, b: a + b)
```

### Key Points

- Input must be in (key, value) format
- Performs local aggregation before shuffle → efficient
- Used heavily for **grouping and summarizing**

### Real-World Analogy

Imagine every store in a city calculates its **own sales total locally**, then all stores send the **summary total** to HQ for final aggregation.

## Example: Sales by Region

```
rdd = spark.sparkContext.parallelize([
    ('east', 100), ('west', 200), ('east', 300), ('west', 100)
])

rdd.reduceByKey(lambda a, b: a + b).collect()
# Output: [('east', 400), ('west', 300)]
```

## Use Cases

- Total revenue per category
- Aggregating clicks per user
- Merging word counts from multiple documents

## Behind the Scenes

- **Map Phase:** Each partition locally reduces (key, value) pairs.
- **Shuffle Phase:** Only one value per key is shuffled to the reducer.

- **Reduce Phase:** Final merge across partitions.

This makes it **much faster** and more scalable than `groupByKey()` (which moves all values across the network before reducing).

## countByValue() – Frequency Count

### Definition

Returns a dictionary of the **number of times** each distinct value occurs in the RDD.

```
rdd.countByValue()
```

### Key Points:

- **Action**
- Operates on entire RDD → returns result to the driver
- Not memory-efficient for large cardinality

### Real-World Analogy

You're counting how many people from each department submitted a form.

### Example

```
rdd = spark.sparkContext.parallelize(['apple', 'banana', 'apple', 'banana', 'orange'])
rdd.countByValue()
# Output: {'apple': 2, 'banana': 2, 'orange': 1}
```

### Use Cases

- Frequency of error codes or log levels
- Count how many users selected each option
- Popularity of tags, brands, categories

### ⚠ Warning

Returns a Python dict, **collected on driver** — avoid using on very large RDDs with many unique values.

## Example : Employee Records

ID	Name	Department	Salary	Status
1	Alice	HR	50000	active
2	Bob	IT	65000	active
3	Charlie	IT	70000	inactive
4	David	Finance	48000	active
5	Eva	HR	52000	active
6	Frank	IT	62000	inactive
7	Grace	Finance	55000	active
8	Henry	IT	67000	active
9	Irene	HR	51000	inactive
10	Jack	Finance	59000	active

## Step 1: Create an RDD from this data

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("FilterMapExample").getOrCreate()
sc = spark.sparkContext

# Simulate the dataset as a list of tuples
data = [
    (1, "Alice", "HR", 50000, "active"),
    (2, "Bob", "IT", 65000, "active"),
    (3, "Charlie", "IT", 70000, "inactive"),
    (4, "David", "Finance", 48000, "active"),
    (5, "Eva", "HR", 52000, "active"),
    (6, "Frank", "IT", 62000, "inactive"),
    (7, "Grace", "Finance", 55000, "active"),
    (8, "Henry", "IT", 67000, "active"),
    (9, "Irene", "HR", 51000, "inactive"),
    (10, "Jack", "Finance", 59000, "active")
]

rdd = sc.parallelize(data)
```

## Goal:

- Filter **only "active"** IT employees
- For each, return a formatted string like  
"Bob from IT earns 65000"

## Step 2: Apply .filter() and .map()

```
result = rdd.filter(lambda x: x[2] == "IT" and x[4] == "active").map(lambda x: f"{x[1]}  
from {x[2]} earns {x[3]}")
```

## Explanation

Code Part	What It Does
rdd.filter(lambda x: ...)	Filters records that match a condition
x[2] == "IT"	Checks if Department is IT
x[4] == "active"	Checks if Status is active
.map(lambda x: ...)	Transforms each record to a formatted string
f"{x[1]} from {x[2]} earns {x[3]}"	Constructs a readable sentence

## Step 3: Collect the Result

```
print(result.collect())
```

### Output

```
['Bob from IT earns 65000', 'Henry from IT earns 67000']
```

## More Advanced Combined Examples

### Example 1: Increase salary by 10% for active HR employees

```
rdd.filter(lambda x: x[2] == "HR" and x[4] == "active") \  
.map(lambda x: (x[1], x[3] * 1.1)) \  
.collect()
```

### Output:

```
[('Alice', 55000.0), ('Eva', 57200.0)]
```

\*\*\*\*\* The backslash \ in Python is a **line continuation character**. It tells Python that the **code on the next line is still part of the current line**.

**Example 2: Get names of employees with salary > 60000 and status = "inactive"**

```
rdd.filter(lambda x: x[3] > 60000 and x[4] == "inactive") \  
    .map(lambda x: x[1]) \  
    .collect()
```

**Output:**

```
['Charlie', 'Frank']
```

**Example 3: Convert to (Department, 1) pair and count employees per department**

```
rdd.map(lambda x: (x[2], 1)) \  
    .reduceByKey(lambda a, b: a + b) \  
    .collect()
```

**Output**

```
[('HR', 3), ('IT', 4), ('Finance', 3)]
```

# Narrow vs Wide Transformations

## What Are Transformations in Spark?

Transformations are operations on RDDs/DataFrames that **create new RDDs/DataFrames** without immediately computing them. They're evaluated **lazily** (only when an action like `.collect()` or `.count()` is called).

These transformations are categorized as:

Type	Data Shuffling Involved?	Performance Impact
Narrow	No	Faster
Wide	Yes	Slower

## Narrow Transformations

### Definition

Narrow transformations are transformations **where each partition of the parent RDD/DataFrame is used by at most one partition of the child**.

No data movement across the cluster.

### Examples

- `map()`
- `filter()`
- `union()`
- `sample()`
- `glom()`
- `flatMap()`

### Performance

- **Fast** because there's **no shuffle**.
- Can be **pipelined** (chained together and executed in a single stage).

### Example

```
rdd = sc.parallelize([1, 2, 3, 4])
mapped = rdd.map(lambda x: x * 2)
filtered = mapped.filter(lambda x: x > 4)
```

Both map and filter are narrow. No shuffle occurs. Execution remains within the same partition.

## Wide Transformations

### Definition

Wide transformations require **shuffling of data** between partitions — **data from multiple partitions in the parent RDD is needed to compute one partition in the child**.

Triggers shuffle — expensive operation

### Examples

- groupByKey()
- reduceByKey()
- distinct()
- join()
- repartition()
- cogroup()
- sortByKey()

### Performance

- **Slower** due to:
  - Disk & network I/O
  - Data serialization
  - Shuffle file writing and reading

### Example

```
rdd = sc.parallelize([('a', 1), ('b', 1), ('a', 2)])
reduced = rdd.reduceByKey(lambda a, b: a + b)
```

This requires **grouping by keys**, which may be distributed across partitions — so Spark **shuffles** data to bring same keys together.

### Visual Comparison

Narrow Transformation		Wide Transformation
<b>Dependency</b>	One-to-one	Many-to-one or many-to-many
<b>Data Movement</b>	No (within same partition)	Yes (shuffle between nodes)
<b>Execution Stage</b>	Same stage	New stage (shuffle boundary)
<b>Speed</b>	Faster	Slower
<b>Examples</b>	map, filter, union	reduceByKey, groupByKey, join

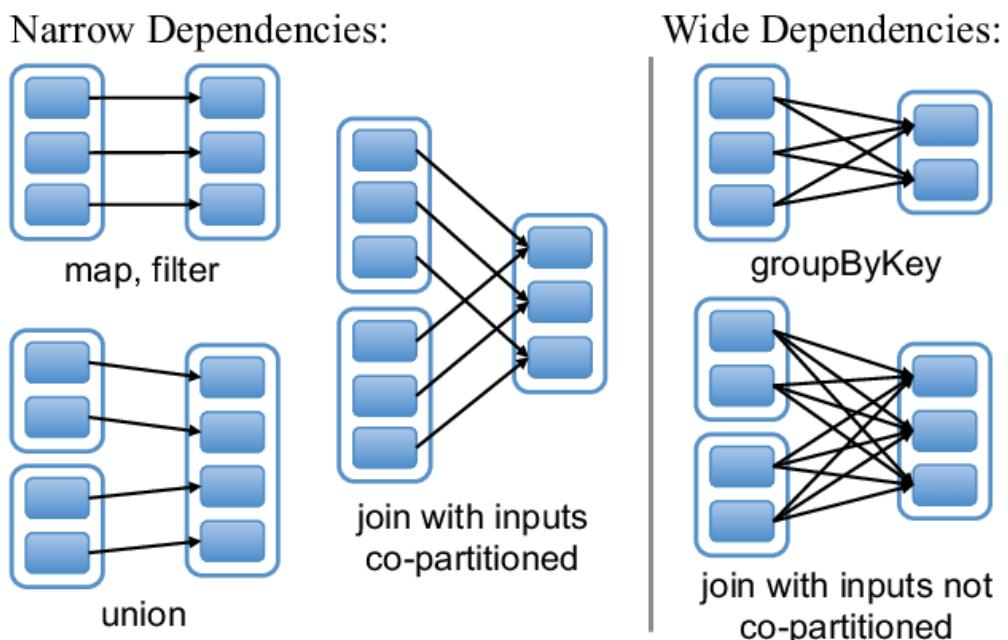
## Why This Matters?

Understanding this helps you:

- Optimize performance
- Reduce costly shuffles
- Design better data pipelines

## Practical Tips

- Prefer **reduceByKey()** over **groupByKey()** (avoids unnecessary shuffles)
- Use **coalesce()** over **repartition()** when decreasing partitions
- Chain **narrow transformations** to stay within a stage



# Working with HDFS in Spark

## What is HDFS?

**HDFS (Hadoop Distributed File System)** is a fault-tolerant, distributed file system used to store large datasets across many nodes.

Spark can **read from and write to HDFS** just like a local file system, thanks to its Hadoop compatibility.

## Basic Requirements

- Spark must be built with Hadoop support (or packaged with Hadoop).
- HDFS must be accessible from Spark cluster (ensure hostname or IPs are resolvable).
- Files in HDFS are usually accessed with paths like:
  - `hdfs://namenode_host:port/path/to/file`
  - or
  - `hdfs:///path/to/file` # If Spark is running in the same Hadoop cluster

## Read & Write Operations

### Reading from HDFS

#### ► Read a text file (each line becomes a row)

```
rdd = spark.sparkContext.textFile("hdfs:///user/data/input.txt")
```

#### ► Read a CSV file as a DataFrame

```
df = spark.read.csv("hdfs:///user/data/employees.csv", header=True, inferSchema=True)
```

#### ► Read JSON

```
df = spark.read.json("hdfs:///user/data/logs.json")
```

### Writing to HDFS

#### ► Write a DataFrame to CSV

```
df.write.csv("hdfs:///user/output/employees_out", header=True, mode='overwrite')
```

#### ► Write as Parquet (efficient binary format)

```
df.write.parquet("hdfs:///user/output/employees_parquet", mode='overwrite')
```

## ► Write as JSON

```
df.write.json("hdfs:///user/output/employees_json", mode='overwrite')
```

## Example: Read, Transform, and Write Back

```
# Step 1: Read CSV from HDFS
df = spark.read.csv("hdfs:///user/data/employees.csv", header=True, inferSchema=True)

# Step 2: Filter active employees and select required columns
active_df = df.filter(df.status == "active").select("name", "department")

# Step 3: Save to HDFS as Parquet
active_df.write.parquet("hdfs:///user/data/active_employees", mode="overwrite")
```

## Handy Options and Modes

Option	Description
<b>header=True</b>	Treat first row as column names
<b>inferSchema=True</b>	Let Spark guess data types
<b>mode='overwrite'</b>	Replace existing files
<b>mode='append'</b>	Add data to existing folder

## Advanced Tips

### Working with Partitioned Output

```
df.write.partitionBy("department").parquet("hdfs:///user/output/partitioned_employees")
```

This will write files into subdirectories like:

```
.../department=HR/
.../department=Finance/
```

### Check file contents (from terminal)

```
hdfs dfs -ls /user/data/
hdfs dfs -cat /user/data/employees.csv
```

## Working in Local vs HDFS

Path Type	Example	Usage
Local	"file:///home/user/data.csv"	For local debugging
HDFS	"hdfs:///user/data/data.csv"	For distributed processing

## Why Spark + HDFS?

Benefit	Description
Scalability	Handles petabyte-scale data
Fault Tolerance	HDFS replicates blocks (default 3 copies)
Compatibility	Works with Hive, Pig, MapReduce
High Throughput	Optimized for batch processing
Lazy Evaluation	Spark optimizes read/write at execution

# Jobs, Stages, and Tasks in Spark UI

## Why It Matters

When you write PySpark code like

```
df.filter(df.age > 25).groupBy("country").count().show()
```

Spark doesn't run line-by-line — it builds an execution plan divided into:

- **Jobs**
- **Stages**
- **Tasks**

Each level breaks the computation into smaller, manageable, and optimizable parts.

## Spark Execution Hierarchy

Level	Description
<b>Job</b>	Triggered by an <b>action</b> (e.g., count(), collect()), represents the <b>entire query</b>
<b>Stage</b>	A division of a job — determined by <b>shuffle boundaries</b> (narrow vs wide transformations)
<b>Task</b>	The <b>smallest unit</b> — each stage runs multiple tasks (one per partition)

## Job

### What is it?

- A **Job** is triggered whenever you perform an **action** like .count(), .collect(), .show(), etc.
- A job is the **entry point for execution**.

## In Spark UI

- Found under the **“Jobs” tab**.
- Shows Job ID, DAG visualization, time taken, and number of stages.

## Stage

### What is it?

- A **Stage** is a **set of transformations** that can be executed **without shuffling** (only narrow dependencies).
- Spark breaks a job into **one or more stages**:
  - **Stage 0 → Stage 1 → ...**

**Spark creates a new stage at each wide transformation (e.g., reduceByKey, join).**

## In Spark UI:

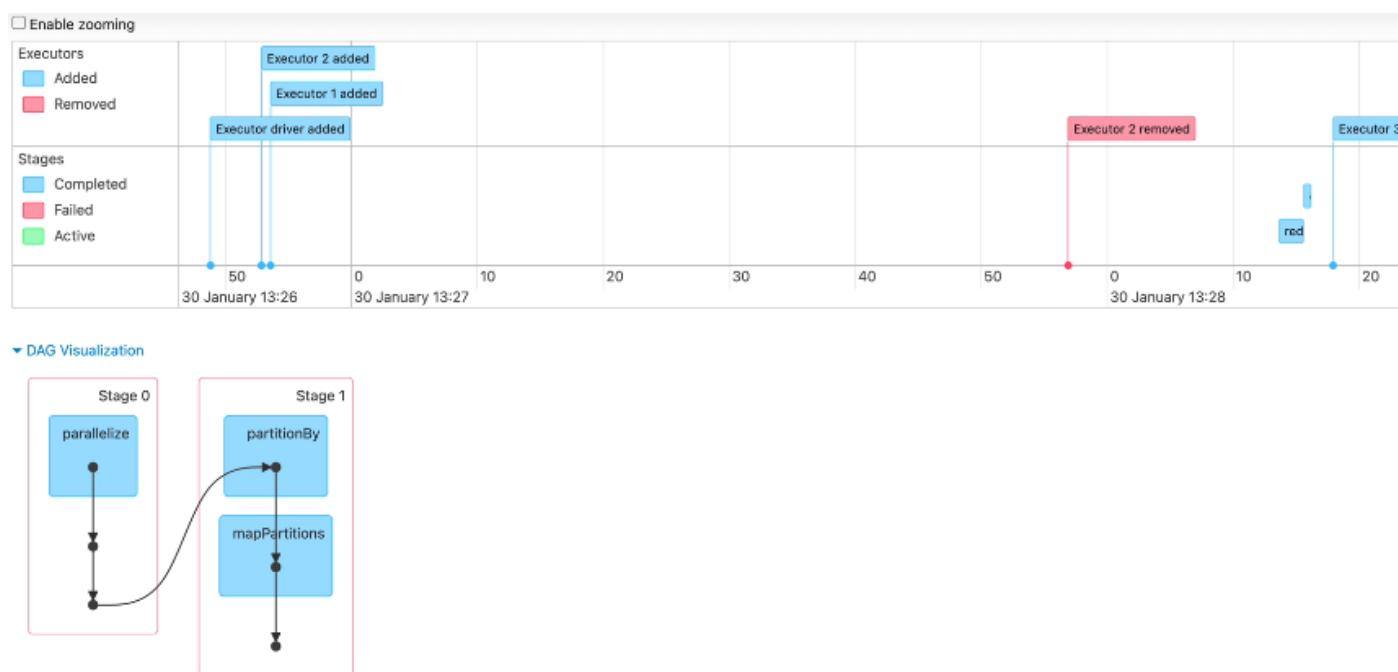
- Found inside a Job under the "**Stages**" tab
- Shows DAG for each stage, number of tasks, and stage time breakdown.

## Task

### What is it?

- A **Task** is a **unit of work** sent to a Spark executor — **each task handles a single partition** of the data.
- Multiple tasks = **parallelism**

If a Stage has 10 partitions, it runs 10 tasks — one for each partition.



## In Spark UI

- Inside each Stage, under "**Tasks**" tab
- You can see:
  - Task IDs
  - Execution times
  - GC (Garbage Collection) time
  - Input size
  - Host (which executor/node ran it)

## Example

Let's say each row is

```
data = [
    [1, "Alice", "HR", 5000, "active"],
    [2, "Bob", "Engineering", 6000, "inactive"],
    [3, "Charlie", "HR", 5500, "active"],
    [4, "David", "Engineering", 7000, "active"],
    [5, "Eva", "HR", 5200, "inactive"]
]
```

## PySpark Code with map(), reduceByKey(), and count()

```
from pyspark.sql import SparkSession

# Start Spark
spark = SparkSession.builder.appName("RDDEexample").getOrCreate()
sc = spark.sparkContext

# Parallelize data into RDD (2 partitions)
rdd = sc.parallelize(data, 2)

# Step 1: Filter only active employees
active_employees = rdd.filter(lambda x: x[4] == "active")

# Step 2: Map as (department, salary)
dept_salary = active_employees.map(lambda x: (x[2], x[3]))

# Step 3: Reduce by key (sum salaries per department)
dept_total_salary = dept_salary.reduceByKey(lambda a, b: a + b)

# Step 4: Count number of departments (action)
result = dept_total_salary.count()

print("Number of active departments:", result)
```

## What's Happening Internally

### Action Triggers a Job

- .count() is an **action**, so it **triggers a job**.
- Spark builds a **DAG** of all transformations:
  - filter → map → reduceByKey

## Stage Breakdown

Stage	Operation	Type of Transformation
Stage 0	.filter() & .map()	<b>Narrow</b> (no shuffle)
Stage 1	.reduceByKey()	<b>Wide</b> (shuffle needed to group keys)

## Tasks and Executors

Concept	What Happens
Partitions	Data is divided into 2 partitions — so 2 tasks per stage
Tasks	Each task handles one partition
Executors	Run tasks in parallel, send intermediate output for shuffle
Shuffle	Required in reduceByKey() to group department keys together
Final Result	Count of departments is returned to the driver after reduceByKey

## Visualization

```
Job 0: count()

Stage 0: filter → map      (narrow)
[ Partition1
| Partition2 ]  --> Tasks on Executors

Stage 1: reduceByKey      (wide)
[ Shuffling
| Group by Key ]  --> sum(salary)
```

## Output Example

With the data:

```
# After filtering active:
[
  [1, "Alice", "HR", 5000, "active"],
  [3, "Charlie", "HR", 5500, "active"],
  [4, "David", "Engineering", 7000, "active"]
]
```

The mapped output:

```
[  
  ("HR", 5000),  
  ("HR", 5500),  
  ("Engineering", 7000)  
]
```

Then reduced:

```
[  
  ("HR", 10500),  
  ("Engineering", 7000)  
]
```

Final .count() → **2 departments**

## Spark UI View

Tab	What You'll See
Jobs	1 Job (from count())
Stages	2 Stages: 0 (filter+map), 1 (reduceByKey)
Tasks	2 tasks per stage (1 per partition)
Executors	How each executor handled partitions/tasks
Shuffle Read/Write	Shown for reduceByKey stage

## Summary

Step	Operation	Type	Notes
1	filter(lambda x[4])	Narrow	No shuffle
2	map(lambda x[2], x[3])	Narrow	Dept & salary
3	reduceByKey()	<b>Wide</b>	Shuffle to group depts
4	count()	Action	Result sent to driver
5	Executors	Parallel tasks	Handled 2 tasks per stage

## groupByKey() and reduceByKey()

Both are **wide transformations** used on **Pair RDDs** ((key, value) format).

They perform **grouping or aggregation** based on the key, but their internal working and performance differ greatly.

### Syntax

```
rdd.groupByKey()  
rdd.reduceByKey(lambda x, y: x + y)
```

### Under the Hood

Concept	groupByKey()	reduceByKey()
Purpose	Groups values by key	Reduces (aggregates) values by key
Data Movement	Sends all values to one place per key	Combines values locally before shuffling
Shuffling	More data shuffling	Less data shuffling
Efficiency	Slower and memory-intensive	Faster and more scalable
Usage	Use <b>only when</b> aggregation isn't possible	Use for <b>summing/counting</b> scenarios

### Example Use Case

Let's say we have this RDD

```
data = [  
    ("HR", 5000),  
    ("Engineering", 7000),  
    ("HR", 5500),  
    ("Engineering", 6000),  
    ("HR", 5200)  
]  
  
rdd = sc.parallelize(data)
```

### groupByKey()

```
grouped = rdd.groupByKey()  
for key, values in grouped.collect():  
    print(key, list(values))
```

### Output

HR [5000, 5500, 5200]

Engineering [7000, 6000]

**Good for:** When you want to access all values per key (e.g., to compute custom stats like median, std. dev).

**Bad for:** Performance at scale (huge shuffle, memory pressure).

### reduceByKey()

```
reduced = rdd.reduceByKey(lambda a, b: a + b)
print(reduced.collect())
```

**Output:**

```
[('HR', 15700), ('Engineering', 13000)]
```

**Good for:** Fast aggregations — sum, count, min, max.

**Not useful if** you need the individual values (use groupByKey then).

### Performance: Why reduceByKey() Is Better

Step	groupByKey()	reduceByKey()
1	Send all (key, value) pairs to 1 machine	Combine values in each partition first
2	Group all values together	Then shuffle reduced (smaller) output
3	Needs more memory (all values together)	Requires less memory and network traffic

### Illustration

Imagine you have:

```
[("HR", 1), ("HR", 2), ("HR", 3)]
```

- groupByKey() → sends all 3 to 1 machine → groups to [1, 2, 3] → then you sum.
- reduceByKey() → sums  $(1 + 2) = 3$  locally → sends (HR, 3) + (HR, 3) → final sum = 6

### Warning for Data Engineers

- groupByKey() can **cause OutOfMemory errors** on large datasets.
- Always **prefer reduceByKey() or aggregateByKey()** when possible.

## When to Use What?

Goal	Use
<b>Summing, counting, reducing</b>	reduceByKey()
<b>Need all raw values per key</b>	groupByKey()
<b>Advanced aggregations (mean)</b>	Prefer combineByKey() or DataFrame APIs

## Pro Tip: Use DataFrames Instead

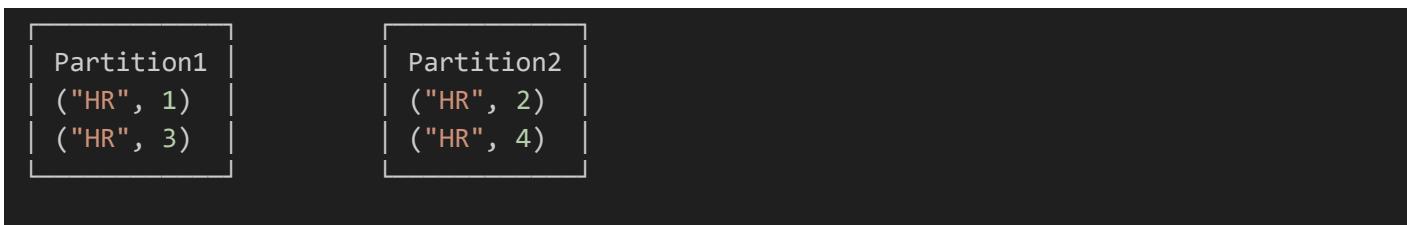
For many cases, especially with large datasets or complex aggregations, it's more efficient to use Spark SQL/DataFrames:

```
df.groupBy("department").agg(F.sum("salary"))
```

Spark handles all optimization under the hood via Catalyst and Tungsten engines.

## Text Diagram: groupByKey()

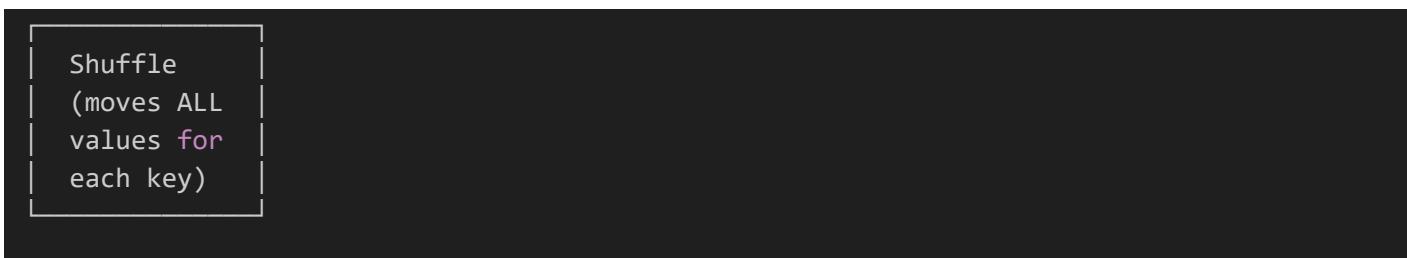
Partitions Before



groupByKey() Execution

- 1 No local aggregation → all values for "HR" must go to one reducer.
- 2 Data shuffled across the cluster (massive network I/O).
- 3 Final stage groups [1, 2, 3, 4] for "HR".

Shuffle Phase:



Result After Shuffle:

```
[ ("HR", [1,2,3,4]) |
```

Issues

- ✖ More data sent over network.
- ✖ Higher memory use → risk of OOM.

**Text Diagram: reduceByKey()**

Partitions Before:

```
Partition1  
("HR", 1)  
("HR", 3)
```

```
Partition2  
("HR", 2)  
("HR", 4)
```

reduceByKey() Execution:

- 1 Local combiner: each partition aggregates its own keys:

Partition1 → ("HR", 1+3=4)

Partition2 → ("HR", 2+4=6)

Shuffle Phase

```
Shuffle  
(sends ONLY  
partial sums)|
```

Result After Shuffle:

```
[ ("HR", 4+6=10) |
```

Benefits:

- Less data shuffled (partial sums smaller than raw values).
- Lower memory use, faster execution.

### Key Differences Summarized

Step	groupByKey()	reduceByKey()
Local aggregation	 No	<input checked="" type="checkbox"/> Yes
Network traffic	 High (all values shuffled)	<input checked="" type="checkbox"/> Low (only combined values)
Memory requirement	 High (all values grouped in reducer)	<input checked="" type="checkbox"/> Low (aggregated in steps)
Performance	 Slower	<input checked="" type="checkbox"/> Faster

### What Happens in Executors?

- Executors process tasks for partitions
- Shuffle files are written/read on disk during wide transformation
- For reduceByKey:
  - Local aggregation reduces shuffle size
    - For groupByKey:
  - All values must move → larger shuffle, more GC, higher chance of OOM

## Exercise

```
! hadoop fs -ls -h /tmp/
Found 6 items
-rw-r--r--  2 mahelapandukabandara hadoop      10.0 M 2025-08-01 07:29 /tmp/customers_10mb.csv
-rw-r--r--  2 mahelapandukabandara hadoop    160.7 M 2025-08-01 07:30 /tmp/customers_150mb.csv
-rw-r--r--  2 mahelapandukabandara hadoop      1.0 M 2025-08-01 07:26 /tmp/customers_1mb.csv
-rw-r--r--  2 mahelapandukabandara hadoop   327.4 M 2025-08-01 08:48 /tmp/customers_350mb.csv
drwxrwxrwt  - hdfs                      hadoop      0 2025-06-30 09:52 /tmp/hadoop-yarn
drwx-wx-wx  - hive                      hadoop      0 2025-06-30 09:53 /tmp/hive
```

```
! hadoop fs -head /tmp/customers_1mb.csv
customer_id,name,city,state,country,registration_date,is_active
0,Customer_0,Pune,Maharashtra,India,2023-06-29,False
1,Customer_1,Bangalore,Tamil Nadu,India,2023-12-07,True
2,Customer_2,Hyderabad,Gujarat,India,2023-10-27,True
3,Customer_3,Bangalore,Karnataka,India,2023-10-17,False
4,Customer_4,Ahmedabad,Karnataka,India,2023-03-14,False
5,Customer_5,Hyderabad,Karnataka,India,2023-07-28,False
6,Customer_6,Pune,Delhi,India,2023-08-29,False
7,Customer_7,Ahmedabad,West Bengal,India,2023-12-28,True
8,Customer_8,Pune,Karnataka,India,2023-06-22,True
9,Customer_9,Mumbai,Telangana,India,2023-01-05,True
10,Customer_10,Pune,Gujarat,India,2023-08-05,True
11,Customer_11,Delhi,West Bengal,India,2023-08-02,False
12,Customer_12,Chennai,Gujarat,India,2023-11-21,False
13,Customer_13,Chennai,Karnataka,India,2023-11-06,True
14,Customer_14,Hyderabad,Tamil Nadu,India,2023-02-07,False
15,Customer_15,Mumbai,Gujarat,India,2023-03-02,True
16,Customer_16,Chennai,Karnataka,India,2023-04-05,False
17,Customer_17,Hyderabad,West Bengal,India
```

```
hdfs_path = "/tmp/customers_1mb.csv"
rdd = spark.sparkContext.textFile(hdfs_path)
header = rdd.first()
rdd_no_header = rdd.filter(lambda row: row != header).map(lambda row : row.split(','))
rdd_no_header.first()
['0', 'Customer_0', 'Pune', 'Maharashtra', 'India', '2023-06-29', 'False']
```

```
reduced_by_rdd = rdd_no_header.map(lambda row : (row[2],1)).reduceByKey(lambda x,y : x+y)
reduced_by_rdd.collect()
[('Pune', 2243),
 ('Hyderabad', 2242),
 ('Mumbai', 2142),
 ('Delhi', 2200),
 ('Bangalore', 2211),
 ('Ahmedabad', 2198),
 ('Chennai', 2194),
 ('Kolkata', 2223)]
```

```
grouped_by_rdd = rdd_no_header.map(lambda row: (row[2],1)).groupByKey()
grouped_by_rdd.collect()
[('Pune', <pyspark.resultiterable.ResultIterable at 0x7f22f1912c10>),
 ('Hyderabad', <pyspark.resultiterable.ResultIterable at 0x7f22f2f2a9d0>),
 ('Mumbai', <pyspark.resultiterable.ResultIterable at 0x7f22f192b590>),
 ('Delhi', <pyspark.resultiterable.ResultIterable at 0x7f22f2ef8750>),
 ('Bangalore', <pyspark.resultiterable.ResultIterable at 0x7f22f2f42e50>),
 ('Ahmedabad', <pyspark.resultiterable.ResultIterable at 0x7f22f2f5b210>),
```

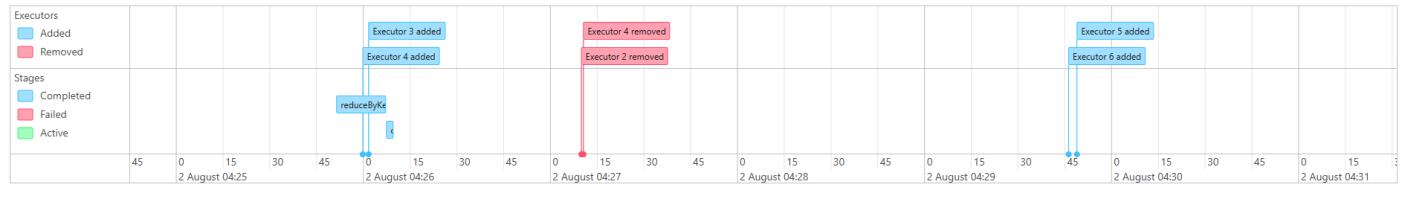
```
('Chennai', <pyspark.resultiterable.ResultIterable at 0x7f22f1928490>),
('Kolkata', <pyspark.resultiterable.ResultIterable at 0x7f22f192b9d0>)]
```

```
groupd_by_result = grouped_by_rdd.map(lambda row: (row[0],len(row[1])))
groupd_by_result.collect()
[('Pune', 2243),
 ('Hyderabad', 2242),
 ('Mumbai', 2142),
 ('Delhi', 2200),
 ('Bangalore', 2211),
 ('Ahmedabad', 2198),
 ('Chennai', 2194),
 ('Kolkata', 2223)]
```

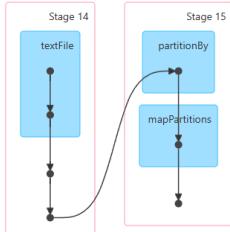
\*\*\*groupByKey() gives an intermediate step with a collection of required data

Lets go fo 350mb file

```
hdfs_path = "/tmp/customers_350mb.csv"
rdd = spark.sparkContext.textFile(hdfs_path)
rdd.map(lambda row: row.split(',')).map(lambda row: (row[2],1)).reduceByKey(lambda x,y :x+y).collect()
[('Bangalore', 661013),
 ('Kolkata', 660174),
 ('Mumbai', 661241),
 ('Hyderabad', 662281),
 ('Chennai', 660249),
 ('Ahmedabad', 660218),
 ('Delhi', 661025),
 ('Pune', 660737),
 ('city', 1)]
```



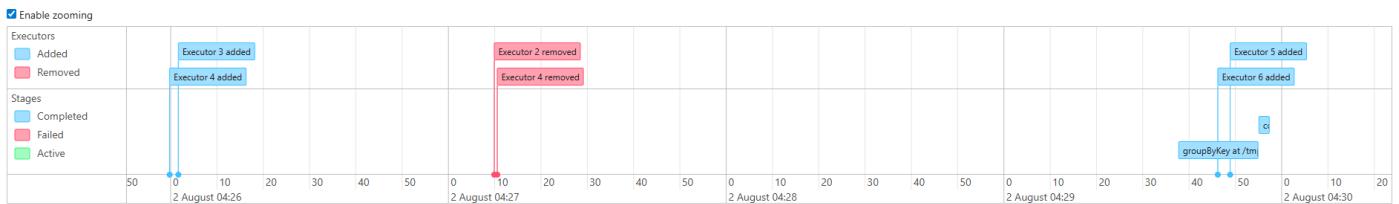
▼ DAG Visualization



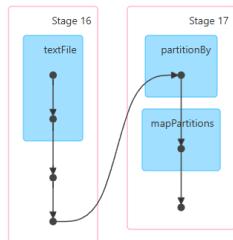
▼ Completed Stages (2)

Completed Stages (2)		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
15	collect at /tmpipykernel_2894/3422105066.py:3	+details 2025/08/02 04:26:07	2 s	3/3			993.0 B	
14	reduceByKey at /tmpipykernel_2894/3422105066.py:3	+details 2025/08/02 04:25:51	16 s	3/3	327.5 MiB			993.0 B

```
rdd.map(lambda row: row.split(',')).map(lambda row: (row[2],1)).groupByKey().map(lambda row: (row[0],len(row[1]))).collect()
[('Kolkata', 660174),
 ('Bangalore', 661013),
 ('Mumbai', 661241),
 ('Hyderabad', 662281),
 ('Chennai', 660249),
 ('Ahmedabad', 660218),
 ('city', 1),
 ('Pune', 660737),
 ('Delhi', 661025)]
```



▼ DAG Visualization



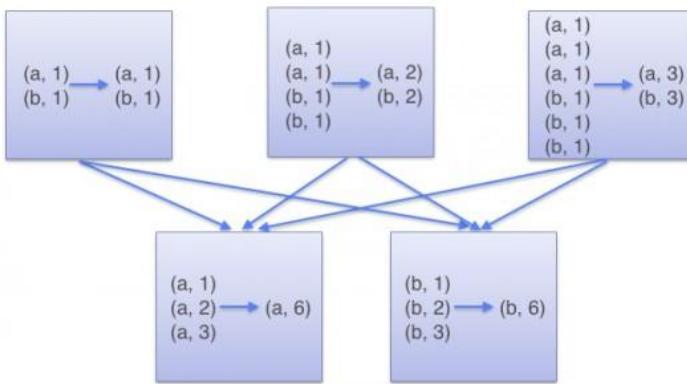
▼ Completed Stages (2)

Page: 1      1 Pages. Jump to  . Show  items in a page. [Go](#)

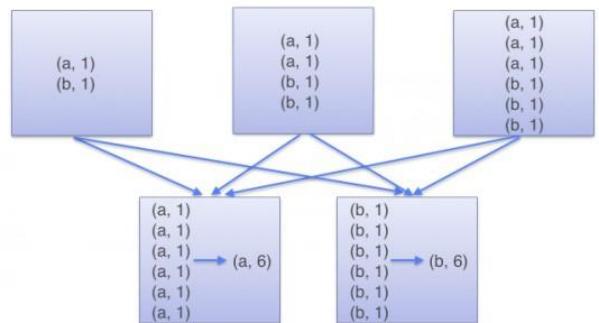
Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
17	collect at /tmp/ipykernel_2894/3114971801.py:1	+details 2025/08/02 04:29:54	2 s	3/3			57.9 KiB	
16	groupByKey at /tmp/ipykernel_2894/3114971801.py:1	+details 2025/08/02 04:29:37	17 s	3/3	327.5 MiB			57.9 KiB

Note the Shuffle read & Shuffle Write is higher in reduceByKey()

## ReduceByKey



## GroupByKey



GroupByKey involves a lot of shuffling causing more memory usage and less write Shuffle read & Shuffle Write leading it to out of memory error. Also parallelism is getting restricted.

# Number of partitions

## Why Do Partitions Matter?

- **Too few partitions** → not enough parallelism → underutilized CPU/cores.
- **Too many partitions** → high overhead managing tasks → network/disk/memory stress.

## When to Increase or Decrease Partitions?

Scenario	Action	Why
Reading small file (e.g., < 1 GB)	✗ Don't increase	Spark already parallelizes based on default settings.
Reading large file (e.g., > 100 GB)	✓ Increase	To parallelize reading → better throughput & core utilization.
After a filter() that removes many rows	✓ Repartition (↓)	To reduce unused partitions & avoid wasting executor resources.
Before a join/groupBy/reduceByKey	✓ Repartition (↑)	To reduce skew, and ensure better data distribution across executors.
Writing small files (many) to HDFS	✓ Coalesce (↓)	Avoid too many small files → combine into fewer partitions.
After shuffle-heavy transformations	✗ Don't repartition blindly	Spark may auto-optimize partitions post-shuffle.

## Spark Partition APIs

1. **rdd.repartition(numPartitions)** → Increases or redistributes partitions (always causes shuffle).
2. **rdd.coalesce(numPartitions)** → Decreases partitions (tries to avoid full shuffle).

### Example: Increase Partitions

```
rdd = spark.sparkContext.textFile("hdfs:///large-file.txt")
rdd = rdd.repartition(100) # More partitions for parallelism
```

**Why?** → If original file had only 10 blocks, it would only make 10 partitions. Repartitioning to 100 allows better use of 100 executor cores.

### Example: Decrease Partitions (Coalesce)

```
rdd = rdd.filter(lambda x: "error" in x)
rdd = rdd.coalesce(4) # Reduce partitions for efficiency in output
rdd.saveAsTextFile("hdfs:///errors")
```

**Why?** → If filtering kept only 5% of data, many partitions might be mostly empty. Coalescing reduces overhead and avoids creating too many tiny output files.

## Rule of Thumb

Dataset Size	Recommended Partitions
< 1 GB	2–4
~10 GB	10–50
~100 GB	100+
~1 TB+	500+

Use:

```
rdd.getNumPartitions()
```

to inspect current partitions.

## Extra Tip: Based on Core Count

Use:

```
spark.sparkContext.defaultParallelism
```

to get the default number of partitions Spark assigns — usually 2x number of CPU cores.

## coalesce() and repartition()

Both are used to **change the number of partitions** in a Spark RDD or DataFrame. But they behave differently:

Feature	repartition()	coalesce()
Purpose	Increase or redistribute partitions	Decrease number of partitions
Shuffle?	<input checked="" type="checkbox"/> Yes (full shuffle)	<span style="color: orange;">⚠</span> Optional (avoids shuffle if possible)
Performance	Slower (shuffle involved)	Faster (no shuffle, if possible)
Use case	To increase parallelism	To optimize output or reduce small files
Can increase partitions	<input checked="" type="checkbox"/> Yes	<span style="color: red;">✗</span> No (only decrease)

### Syntax

```
# repartition (can increase or decrease with shuffle)
rdd2 = rdd.repartition(100)

# coalesce (only decrease, avoids shuffle)
rdd3 = rdd.coalesce(10)
```

### When to Use repartition()

- After reading large files (e.g., 1 big file → 1 partition → bad).
- Before **joins** or **groupByKey** to ensure even data distribution.
- When increasing partitions to improve parallelism.

### Example:

```
large_rdd = spark.sparkContext.textFile("hdfs:///bigdata.txt")
balanced_rdd = large_rdd.repartition(100) # distribute data across 100 partitions
```

## When to Use coalesce()

- After filtering and the data is **small**.
- Before writing output to avoid **too many small files**.
- Ideal for **downscaling partitions**.

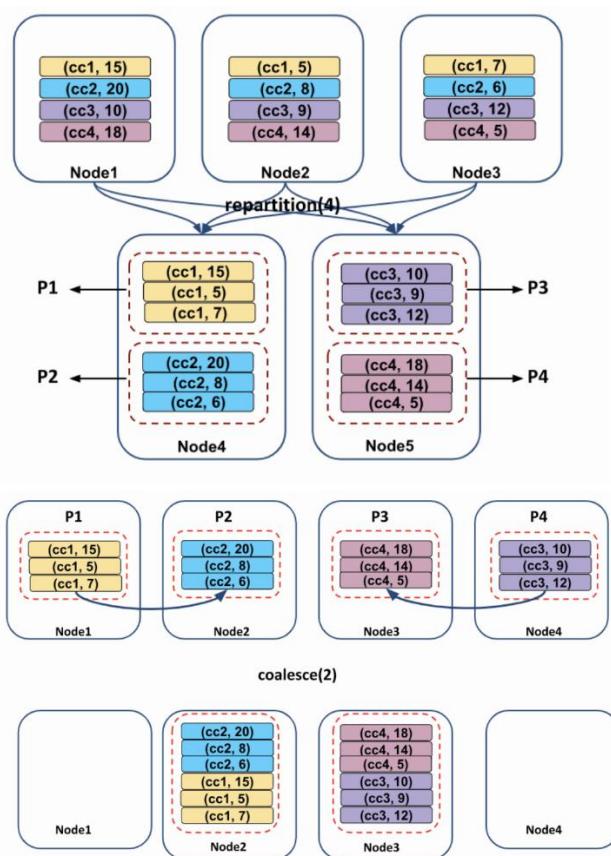
### Example:

```
error_logs = rdd.filter(lambda x: "ERROR" in x)
error_logs = error_logs.coalesce(1) # Combine into 1 file
error_logs.saveAsTextFile("hdfs:///logs/errors")
```

## Practical Case: Difference in Behavior

```
df = spark.range(0, 100)

df1 = df.repartition(10)    # Data is SHUFFLED and evenly spread
df2 = df1.coalesce(2)       # Data is NOT shuffled, just merged (uneven)
```



## Performance Tip:

- Use `repartition()` when **going up** in partition count or **better distribution** is needed.
- Use `coalesce()` when **going down** and you want to avoid the cost of a shuffle.

# Higher-Level APIs in Apache Spark

## What Are Higher-Level APIs?

Spark started with low-level RDDs (Resilient Distributed Datasets), but higher-level APIs like

**DataFrame** and **Spark SQL** were introduced to:

- Make development easier
- Provide **automatic optimizations** (Catalyst, Tungsten)
- Enable **SQL support** and **schema-based processing**

Feature	RDD	DataFrame	Dataset (Scala/Java only)
API Level	Low-level (functional)	High-level (SQL-like)	High-level + compile-time safety
Type Safety	✓ (strong typing)	✗ (rows are not type-safe)	✓ (with case classes)
Performance	✗ Slower (no optimization)	✓ Optimized via Catalyst & Tungsten	✓ Same as DataFrame
Ease of Use	✗ Verbose	✓ Simple	⚠ Limited in Python
Language Support	All (Python, Java, Scala)	All	Scala, Java only

## DataFrame API

### What is a DataFrame?

- Think of it like a **table in a relational database**.
- **Rows and columns** with a defined schema.
- Supports **lazy evaluation, transformations**, and **actions**.
- Built on top of RDDs.

### Features

- Easy to use (SQL-like syntax)
- Memory and CPU optimized via **Catalyst** and **Tungsten**
- Language support: Python (PySpark), Scala, Java, R

## Example (PySpark)

```
df = spark.read.csv("data.csv", header=True, inferSchema=True)
# Select columns
df.select("name", "age").show()
# Filter rows
df.filter(df.age > 30).show()
# Group and aggregate
df.groupBy("department").agg({"salary": "avg"}).show()
```

## Spark SQL Tables

### What Are Spark Tables?

Spark allows you to create **logical tables** using SQL, which can be:

Type	Description
<b>Temporary</b>	Exists during the session, backed by DataFrame/view
<b>Managed</b>	Spark manages the table & data in warehouse directory
<b>External</b>	Table points to externally managed data (e.g., HDFS)

#### Temporary View (Session-scoped table)

```
df.createOrReplaceTempView("employees")

# SQL Query
spark.sql("SELECT * FROM employees WHERE department = 'HR'").show()
```

#### Global Temporary View (available across sessions)

```
df.createOrReplaceGlobalTempView("global_employees")

spark.sql("SELECT * FROM global_temp.global_employees").show()
```

## Creating Permanent Tables

### Managed Table (Spark stores metadata and data)

```
CREATE TABLE students (id INT, name STRING)
USING parquet
OPTIONS ('path' '/user/data/students');
Or using PySpark:
df.write.saveAsTable("students")
```

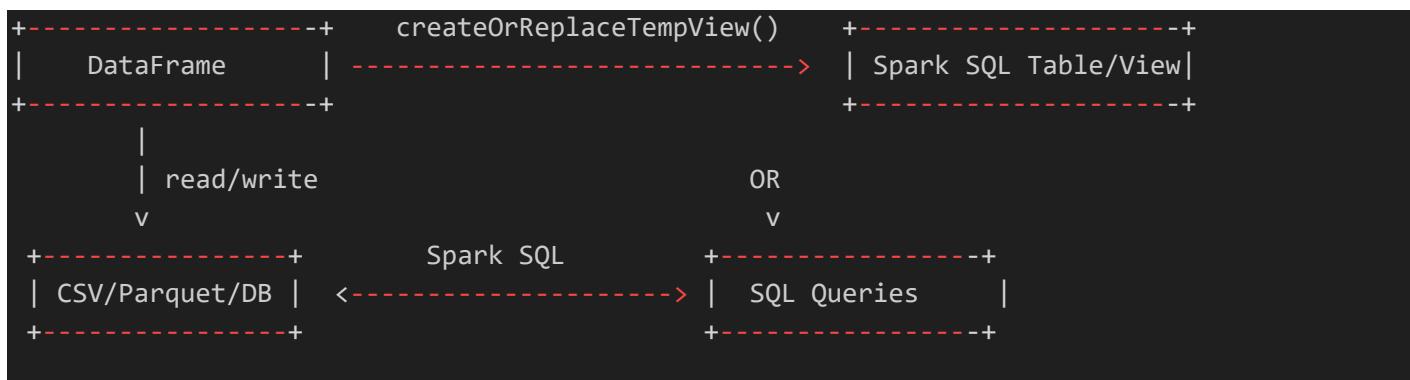
### External Table (you manage the data path)

```
CREATE EXTERNAL TABLE logs (
    timestamp STRING, message STRING
)
LOCATION '/user/log_data/' ;
```

## DataFrame vs Spark Table (SQL-based)

Feature	DataFrame	Spark Table / View
<b>Language Usage</b>	Python/Scala APIs	SQL Queries
<b>Temporary</b>	Yes (DataFrame is in-memory)	Yes (Views, Temp Tables)
<b>Persistence</b>	No (unless written)	Yes (Managed/External tables)
<b>Optimization</b>	Catalyst & Tungsten	Catalyst & Tungsten
<b>Use Case</b>	Code-first data processing	SQL-first analytics, reporting

## Visualization (Text Diagram)



## Summary

Concept	Purpose	Key Method
<b>DataFrame</b>	Distributed table API	<code>spark.read</code> , <code>df.select</code> , <code>df.filter</code>
<b>Temp View</b>	Register DataFrame as table	<code>createOrReplaceTempView()</code>
<b>SQL Table</b>	Persisted table for SQL access	<code>saveAsTable()</code> / <code>CREATE TABLE</code>

## Example

```
df = spark.read.format('csv')\n.option('header','true')\\n.option('inferSchema','true')\\n.load('/tmp/customers_1mb.csv')
```

```
df.head(5)\n[Row(customer_id=0, name='Customer_0', city='Pune', state='Maharashtra', country='India', registration_date=datetime.date(2023, 6, 29), is_active=False),\n Row(customer_id=1, name='Customer_1', city='Bangalore', state='Tamil Nadu', country='India', registration_date=datetime.date(2023, 12, 7), is_active=True),\n Row(customer_id=2, name='Customer_2', city='Hyderabad', state='Gujarat', country='India', registration_date=datetime.date(2023, 10, 27), is_active=True),\n Row(customer_id=3, name='Customer_3', city='Bangalore', state='Karnataka', country='India', registration_date=datetime.date(2023, 10, 17), is_active=False),\n Row(customer_id=4, name='Customer_4', city='Ahmedabad', state='Karnataka', country='India', registration_date=datetime.date(2023, 3, 14), is_active=False)]
```

```
print(df.show(5))\n+-----+-----+-----+-----+-----+\n|customer_id|    name|     city|      state|country|registration_date|is_active|\n+-----+-----+-----+-----+-----+\n|      0|Customer_0|      Pune|Maharashtra|  India|  2023-06-29|   false|\n|      1|Customer_1|Bangalore|  Tamil Nadu|  India|  2023-12-07|    true|\n|      2|Customer_2|Hyderabad|     Gujarat|  India|  2023-10-27|    true|\n|      3|Customer_3|Bangalore|   Karnataka|  India|  2023-10-17|   false|\n|      4|Customer_4|Ahmedabad|   Karnataka|  India|  2023-03-14|   false|\n+-----+-----+-----+-----+-----+\nonly showing top 5 rows
```

```
df.printSchema()\nroot\n|-- customer_id: integer (nullable = true)\n|-- name: string (nullable = true)\n|-- city: string (nullable = true)\n|-- state: string (nullable = true)\n|-- country: string (nullable = true)\n|-- registration_date: date (nullable = true)\n|-- is_active: boolean (nullable = true)
```

# Spark DataFrame function

## Common Spark DataFrame Functions

Function	Description	Example
<code>select()</code>	Pick specific columns	<code>df.select("name", "salary").show()</code>
<code>filter() / where()</code>	Filter rows based on condition	<code>df.filter(df.age &gt; 30).show()</code>
<code>withColumn()</code>	Add new or update column	<code>df.withColumn("bonus", df.salary * 0.1).show()</code>
<code>groupBy()</code>	Group rows by a column	<code>df.groupBy("department").count().show()</code>
<code>agg()</code>	Aggregate stats with expressions	<code>df.groupBy("department").agg({"salary": "avg"}).show()</code>
<code>orderBy()</code>	Sort the DataFrame	<code>df.orderBy("salary", ascending=False).show()</code>
<code>drop()</code>	Remove column(s)	<code>df.drop("age").show()</code>
<code>distinct()</code>	Remove duplicate rows	<code>df.select("department").distinct().show()</code>
<code>dropDuplicates()</code>	Remove duplicate rows by columns	<code>df.dropDuplicates(["department"]).show()</code>
<code>describe()</code>	Summary statistics for numeric columns	<code>df.describe().show()</code>
<code>na.drop()</code>	Remove rows with null values	<code>df.na.drop().show()</code>
<code>na.fill()</code>	Replace null values	<code>df.na.fill({"salary": 0}).show()</code>

```

data = [
    (1, "Alice", 30, "HR", 3000),
    (2, "Bob", 45, "IT", 4000),
    (3, "Cathy", 29, "HR", 3200),
    (4, "David", 35, "IT", 4100),
    (5, "Eve", 25, "Finance", 2800)
]

columns = ["id", "name", "age", "department", "salary"]

df = spark.createDataFrame(data, columns)
df.show()
+---+-----+-----+-----+
| id| name|age|department|salary|
+---+-----+-----+-----+
| 1|Alice| 30|      HR| 3000|
| 2| Bob| 45|      IT| 4000|
| 3|Cathy| 29|      HR| 3200|
| 4|David| 35|      IT| 4100|
| 5| Eve| 25| Finance| 2800|
+---+-----+-----+-----+

```

```

# Select Columns
df.select("name", "salary").show()
+-----+
| name|salary|
+-----+
|Alice| 3000|
| Bob| 4000|
|Cathy| 3200|
|David| 4100|
| Eve| 2800|
+-----+

```

```

# Filter rows based on conditions
df.filter(df.age > 30).show()
+---+-----+-----+-----+
| id| name|age|department|salary|
+---+-----+-----+-----+
| 2| Bob| 45|      IT| 4000|
| 4|David| 35|      IT| 4100|
+---+-----+-----+-----+
#Add new or update column
df.withColumn("bonus", df.salary * 0.1).show()
+-----+-----+-----+-----+
| id| name|age|department|salary|bonus|
+-----+-----+-----+-----+
| 1|Alice| 30|      HR| 3000|300.0|
| 2| Bob| 45|      IT| 4000|400.0|
| 3|Cathy| 29|      HR| 3200|320.0|
| 4|David| 35|      IT| 4100|410.0|
| 5| Eve| 25| Finance| 2800|280.0|
+-----+-----+-----+-----+

```

```
#Group rows by a column  
df.groupBy("department").count().show()
```

```
+-----+-----+  
|department|count|  
+-----+-----+  
|      IT|     2|  
|      HR|     2|  
| Finance|     1|  
+-----+-----+
```

```
# Aggregate stats with expressions
```

```
df.groupBy("department").agg({"salary": "avg"}).show()
```

```
+-----+-----+  
|department|avg(salary)|  
+-----+-----+  
|      IT|    4050.0|  
|      HR|    3100.0|  
| Finance|    2800.0|  
+-----+-----+
```

```
#Sort the DataFrame
```

```
df.orderBy("salary", ascending=False).show()
```

```
+-----+-----+-----+-----+  
| id| name|age|department|salary|  
+-----+-----+-----+-----+  
|  4|David| 35|        IT|   4100|  
|  2| Bob| 45|        IT|   4000|  
|  3|Cathy| 29|        HR|   3200|  
|  1|Alice| 30|        HR|   3000|  
|  5|  Eve| 25|  Finance|   2800|  
+-----+-----+-----+-----+
```

```
#Remove column(s)
```

```
df.drop("age").show()
```

```
+-----+-----+-----+  
| id| name|department|salary|  
+-----+-----+-----+  
|  1|Alice|      HR|   3000|  
|  2| Bob|      IT|   4000|  
|  3|Cathy|      HR|   3200|  
|  4|David|      IT|   4100|  
|  5|  Eve|  Finance|   2800|  
+-----+-----+-----+
```

```
# Remove duplicate rows
```

```
df.select("department").distinct().show()
```

```
+-----+  
|department|  
+-----+  
|      IT|  
|      HR|  
| Finance|  
+-----+
```

```
#Remove duplicate rows by columns
```

```
df.dropDuplicates(["department"]).show()
```

```
+-----+-----+-----+-----+  
| id| name|age|department|salary|  
+-----+-----+-----+-----+  
|  5|  Eve| 25|  Finance|   2800|  
|  1|Alice| 30|        HR|   3000|  
|  2| Bob| 45|        IT|   4000|  
+-----+-----+-----+
```

```
#Summary statistics for numeric columns
```

```
df.describe().show()
```

summary	id	name	age	department	salary
count	5	5	5	5	5
mean	3.0	NULL	32.8	NULL	3420.0
stddev	1.5811388300841898	NULL	7.694153624668537	NULL	593.295878967653
min	1	Alice	25	Finance	2800
max	5	Eve	45	IT	4100

```
active_customers_ = df_2.filter( df_2.is_active==True)
```

```
active_customers_.show(5)
```

# Reading Data from HDFS in Spark

Spark can **directly read files from HDFS** using the spark.read API.

## Example HDFS path

```
hdfs://namenode:9000/user/data/employee.csv
```

### Generic Syntax

```
df = spark.read \
    .format("format_type") \
    .option("key", "value") \
    .load("hdfs://path/to/file")
```

## Common read() Options by File Format

Format	Key Option	Value Description
CSV	header	true = First row as column names
	inferSchema	true = Infer column types. used when reading structured files (like CSV, JSON, etc.) to tell Spark to automatically detect the data types (schema) of each column based on the content.
	delimiter	Custom delimiter (e.g., ',', '\t')
	mode	How to handle bad records: PERMISSIVE, DROPMALFORMED, FAILFAST
JSON	multiline	true = Handles multi-line JSON
Parquet	<i>No special options needed</i>	Schema is embedded
All	path	HDFS path to read from
All	schema	Define schema manually instead of inferring

## Examples by File Type

### 1 Read CSV from HDFS

```
df_csv = spark.read.format("csv") \
    .option("header", "true") \
    .option("inferSchema", "true") \
    .load("hdfs://namenode:9000/user/data/employee.csv")
```

### 2 Read JSON from HDFS

```
df_json = spark.read.format("json") \
    .option("multiline", "true") \
    .load("hdfs://namenode:9000/user/data/employee.json")

df_json.printSchema()
```

### 3 Read Parquet from HDFS

```
df_parquet = spark.read.parquet("hdfs://namenode:9000/user/data/employee.parquet")
df_parquet.select("name", "salary").show()
```

#### Example

```
df_2 = spark.read\
    .format('csv')\
    .option('header','true')\
    .option('inferSchema','true')\
    .load('/data/first_100_customers.csv')

df_2.show(5)
+-----+-----+-----+-----+-----+
|customer_id|      name|     city|     state|country|registration_date|is_active|
+-----+-----+-----+-----+-----+
|      0|Customer_0|      Pune|Maharashtra|   India|2023-06-29|  false|
|      1|Customer_1|Bangalore|Tamil Nadu|   India|2023-12-07|   true|
|      2|Customer_2|Hyderabad|Gujarat|   India|2023-10-27|   true|
|      3|Customer_3|Bangalore|Karnataka|  India|2023-10-17|  false|
|      4|Customer_4|Ahmedabad|Karnataka|  India|2023-03-14|  false|
+-----+-----+-----+-----+-----+
only showing top 5 rows
```

Reading data in Spark using `spark.read` is **neither a transformation nor an action** — it's part of the **DataFrame API's lazy evaluation process**.

Concept	Description
<code>spark.read</code>	Triggers <b>no computation</b> — it just builds a logical plan (like a blueprint).
<code>.load()</code> or <code>.csv()</code>	Returns a <b>DataFrame</b> , which is also lazily evaluated.
<code>.show()</code> , <code>.count()</code>	These are <b>actions</b> — they trigger the actual execution of the read and the transformations.

```
df = spark.read.option("header", "true").csv("hdfs://.../file.csv")
```

- This **does not read data yet**. It plans how to read the data (logical plan).

```
df.show()
```

- This is an **action** that **actually triggers the read** and executes the plan.

If the `inferSchema = True` and `header = True` -> Two jobs created. Reading the whole schema is a single job with action (task per each block). `Header = True` can be treated as transformation.

So providing the schema will be save a lot of time.

If data has k blocks & p partitions

Scenario	Behavior	Explanation	Eager or Lazy?	Jobs / Tasks (If stored in k blocks and p partitions)
<code>header=True, inferSchema=False</code>	<input checked="" type="checkbox"/> Headers used <input type="checkbox"/> Schema = all string	Spark uses the first row as headers. Schema defaults to <code>StringType</code> .Lightweight read.	<input checked="" type="checkbox"/> Lazy until Action	◆ 0 jobs / 0 tasks until action ◆ <code>.show()</code> triggers 1 job with ≈ p tasks
<code>header=False, inferSchema=False</code>	<input type="checkbox"/> No headers <input type="checkbox"/> All strings	Column names auto as <code>_c0, _c1, etc</code> .No type inference.	<input checked="" type="checkbox"/> Lazy until Action	◆ 0 jobs / 0 tasks until action ◆ <code>.show()</code> triggers 1 job with ≈ p tasks
<code>header=True, inferSchema=True</code>	<input checked="" type="checkbox"/> Headers used <input checked="" type="checkbox"/> Schema inferred	Spark scans a <b>portion of the data</b> to infer types.This scan reads k blocks partially.	<input checked="" type="checkbox"/> Lazy until Action	◆ 1 extra job for schema inference ◆ <code>.show()</code> triggers another job with ≈ p tasks
<code>header=False, inferSchema=True</code>	<input type="checkbox"/> No headers <input checked="" type="checkbox"/> Schema inferred	No column names, but type inference happens.	<input checked="" type="checkbox"/> Lazy until Action	◆ 1 extra job for type scan ◆ 1 job on action (like <code>.show()</code> ) ≈ p tasks
<code>load() + .show()</code>	Actual read & display	<code>.load()</code> is lazy → <b>no read</b> . <code>.show()</code> triggers read & display	<input type="checkbox"/> <code>.show() = Action</code>	◆ 1 job for <code>.show()</code> ◆ p tasks (based on partitions)
<b>Only spark.read.option(...) used</b>	No computation	No read, no schema inference. Merely creates <b>logical plan</b> .	<input type="checkbox"/> Lazy	◆ 0 jobs / 0 tasks

# Schema Enforcement in Apache Spark

## What is a Schema in Spark?

A **schema** defines the structure of data — the column names, data types, and nullability — in a DataFrame or Dataset.

## Example Schema

```
StructType([
    StructField("id", IntegerType(), True),
    StructField("name", StringType(), True),
    StructField("salary", FloatType(), False)
])
```

## Why Schema Enforcement Matters

Benefit	Description
<b>Performance</b>	Avoids full data scan (no need to infer types)
<b>Type Safety</b>	Prevents unexpected issues during processing
<b>Data Consistency</b>	Detects incorrect or malformed data early
<b>Easier Debugging</b>	Easier to understand and validate structure
<b>Compatible with Tools</b>	Helps integration with BI tools or SQL-based queries

## Two Modes of Schema Application

Mode	Description
<b>Schema Inference</b>	Spark automatically guesses the types by scanning data
<b>Schema Enforcement (Manual)</b>	You <b>define the schema</b> explicitly using StructType and StructField

## Schema Inference vs Schema Enforcement

Feature	Inference	Manual Schema Enforcement
<b>Convenience</b>	Automatically deduces types	You write them manually
<b>Accuracy Risk</b>	Might infer wrong types	Fully reliable
<b>Performance</b>	Requires scanning part of the file	No scan, faster startup
<b>Robustness</b>	Weak (can misinterpret data)	Strong, validates structure strictly

## How to Enforce Schema in Spark (with Example)

### Example: Reading CSV with Schema

```
from pyspark.sql.types import StructType, StructField, StringType, IntegerType

schema = StructType([
    StructField("id", IntegerType(), True),
    StructField("name", StringType(), True),
    StructField("age", IntegerType(), True)
])

df = spark.read.option("header", True).schema(schema).csv("hdfs://data/users.csv")
df.show()
```

Here, Spark **does not infer types**. It uses the structure we defined in schema.

### Strict Type Checking (Fail Fast)

Spark will throw an error if the actual data **does not match the declared schema**.

#### Example Error:

Caused by: org.apache.spark.SparkException: Cannot parse column id as IntegerType

#### Common Mismatches

Declared Type	Actual Data	Result
IntegerType	"abc"	Fails to cast (error)
FloatType	"NaN"	May succeed (as float)
BooleanType	"yes"	Might not be recognized

### Use Case: Enforcing Schema on JSON

```
schema = StructType([
    StructField("user_id", IntegerType(), True),
    StructField("event", StringType(), True),
    StructField("timestamp", StringType(), True)
])

df = spark.read.schema(schema).json("hdfs://logs/events.json")
```

## Evolving Schemas (Schema Evolution)

- Supported in formats like **Parquet** and **Avro**
- Useful in **data lakes**
- Schema may change over time → Spark can read multiple versions

```
df = spark.read.option("mergeSchema", True).parquet("hdfs://data/users/")
```

## Schema-on-Read vs Schema-on-Write

Strategy	Description
<b>Schema-on-Read</b>	Data is stored as-is; schema applied at read time (e.g., CSV, JSON)
<b>Schema-on-Write</b>	Data is validated <b>before storage</b> , typical in databases

Spark follows **schema-on-read**, but **enforced schema lets you simulate schema-on-write safety**.

## Schema Tools in Spark

Function/Method	Purpose
<b>df.schema</b>	View schema of a DataFrame
<b>df.printSchema()</b>	Pretty-print the schema
<b>spark.read.schema(...)</b>	Apply schema manually
<b>StructType()</b>	Define custom schema
<b>toDF()</b>	Create DF with schema from RDD

## Example: RDD to DataFrame with Schema

```
rdd = spark.sparkContext.parallelize([
    (1, "Alice", 24),
    (2, "Bob", 30)
])

schema = StructType([
    StructField("id", IntegerType(), True),
    StructField("name", StringType(), True),
    StructField("age", IntegerType(), True)
])

df = spark.createDataFrame(rdd, schema)
df.printSchema()
```

## Schema Evolution Pitfall

Let's say:

- File1 has columns: id, name
- File2 has: id, name, age

Then:

```
df = spark.read.option("mergeSchema", True).parquet("...") # OK
df = spark.read.schema(schema).parquet("...") # May fail
```

## Tips for Reliable Schema Enforcement

Always define schema explicitly when:

- Reading from **semi-structured** sources like CSV/JSON
- Writing production-ready **ETL pipelines**
- Integrating with **BI tools / SQL engines**
- Handling **critical typed data** (e.g., timestamps, amounts)

## Summary Table

Concept	Manual Schema	Inferred Schema
Speed	Faster (no scan)	Slower (scans data)
Type Safety	Strong	Weak
Robustness	High	Depends on data
Customization	Yes (nullable, types)	Not customizable
Use in Production	Recommended	Avoid

## DDL (Data Definition Language) Schema in Spark

### What is DDL Schema?

DDL (Data Definition Language) in Spark refers to defining schemas **using SQL-like strings** instead of programmatic APIs like StructType.

DDL schemas are concise, readable, and ideal for **SQL compatibility** or quick schema definitions.

### Syntax

```
column_name data_type [NOT NULL] [COMMENT '...']
```

Used inside a string passed to methods like:

```
spark.read.schema("name STRING, age INT, salary DOUBLE")
```

## When to Use DDL Schema?

Scenario	DDL Schema Usefulness
Quick data ingestion	Easy to write and read
Spark SQL table creation	Works well with CREATE TABLE or Hive-compatible SQL
Lightweight ETL scripts	Avoids verbose StructType definitions
External integrations (JDBC etc.)	Compatible with SQL-like ecosystems

## Supported Data Types (DDL Format)

Spark Type	DDL Type
<b>StringType</b>	STRING
<b>IntegerType</b>	INT
<b>LongType</b>	BIGINT
<b>DoubleType</b>	DOUBLE
<b>FloatType</b>	FLOAT
<b>BooleanType</b>	BOOLEAN
<b>DateType</b>	DATE
<b>TimestampType</b>	TIMESTAMP
<b>DecimalType</b>	DECIMAL(p, s)
<b>ArrayType&lt;T&gt;</b>	ARRAY<T>
<b>MapType&lt;K,V&gt;</b>	MAP<K,V>
<b>StructType</b>	STRUCT<...>

## Example 1: Load CSV with DDL Schema

```
ddl_schema = "id INT, name STRING, salary DOUBLE"

df = spark.read \
    .option("header", True) \
    .schema(ddl_schema) \
    .csv("hdfs://data/employees.csv")

df.printSchema()
```

## Benefits of DDL Schema

Benefit	Description
<b>Concise</b>	Easy to define even complex schemas
<b>Human-readable</b>	Good for documentation, SQL engineers
<b>SQL-Compatible</b>	Matches Spark SQL table definitions
<b>No programmatic imports</b>	Avoids StructType and StructField verbosity

## Limitations of DDL Schema

Limitation	Detail
✗ <b>No Nullability control (in read)</b>	Can't define nullable/non-nullable easily in reads
✗ <b>No metadata (comments, etc.)</b>	Not easily extensible like StructType
✗ <b>Typing errors are runtime</b>	Schema parsing happens only during execution

## Behind the Scenes: Internals

- Spark parses the DDL string into a StructType using its built-in **SQL parser**.
- This schema then gets applied to the read plan (logical plan).
- It **avoids schema inference**, ensuring **faster reads** and **consistent types**.

## Summary Table

Feature	DDL Schema	Programmatic Schema (StructType)
<b>Definition Style</b>	String	Object-oriented
<b>Readability</b>	✓ High	✗ Verbose
<b>Flexibility (e.g., nulls)</b>	✗ Less	✓ More control
<b>Nesting Support</b>	✓ Yes (via STRUCT<>, ARRAY<>)	✓ Yes
<b>Recommended For</b>	Quick ETL, SQL tables	Production code, type-sensitive workflows

## Read Modes in Spark

Spark provides **three main modes** to deal with **corrupt or bad records** while reading files (especially CSV, JSON).

Mode	Behavior
PERMISSIVE	Default. Corrupt records are put in a special column _corrupt_record.
FAILFAST	Fails immediately upon reading a corrupt record.
DROPMALFORMED	Drops corrupt records and reads only valid rows.

### Sample CSV File

employees.csv

```
id,name,age,salary
1,John,30,60000
2,Jane,29,55000
3,Alice,not_a_number,70000
4,Bob,25,50000
```

The third row has "not\_a\_number" instead of a numeric value for age.

### 1. PERMISSIVE Mode (default)

```
df = spark.read \
    .option("header", True) \
    .option("mode", "PERMISSIVE") \
    .option("inferSchema", True) \
    .csv("employees.csv")
df.show(truncate=False)
```

Output:

	id	name	age	salary	_corrupt_record
1	1	John	30	60000	null
2	2	Jane	29	55000	null
3	3	null	null	null	"3,Alice,not_a_number,70000"
4	4	Bob	25	50000	null

! Corrupt row stored in \_corrupt\_record column, other fields are null.

## 2. FAILFAST Mode

```
df = spark.read \
    .option("header", True) \
    .option("mode", "FAILFAST") \
    .option("inferSchema", True) \
    .csv("employees.csv")
```

### Output

org.apache.spark.SparkException: Malformed records are detected in record parsing.

✖️ Spark throws an **exception immediately** when it encounters the malformed row (not\_a\_number).

## 3. DROPMALFORMED Mode

```
df = spark.read \
    .option("header", True) \
    .option("mode", "DROPMALFORMED") \
    .option("inferSchema", True) \
    .csv("employees.csv")
df.show()
```

### Output

	<b>id</b>	<b>name</b>	<b>age</b>	<b>salary</b>
	1	John	30	60000
	2	Jane	29	55000
	4	Bob	25	50000

✓ Malformed row (row 3) is **silently dropped** from the output.

## Comparison Summary

Mode	Keeps Bad Records?	Throws Error?	Adds _corrupt_record Column?
PERMISSIVE	✓ Yes	✗ No	✓ Yes
FAILFAST	✗ No	✓ Yes	✗ No
DROPMALFORMED	✗ No	✗ No	✗ No

# writing a CSV file into HDFS using Apache Spark

## Step-by-Step: Writing CSV to HDFS with PySpark

### 1. Start Spark Session

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("WriteCSVtoHDFS") \
    .getOrCreate()
```

### 2. Create Sample DataFrame

You can create a DataFrame from a Python list or read from an existing source.

```
data = [
    (1, "Alice", 28, "HR"),
    (2, "Bob", 35, "Finance"),
    (3, "Cathy", 30, "IT"),
]

columns = ["id", "name", "age", "department"]

df = spark.createDataFrame(data, schema=columns)
```

### 3. Write DataFrame to HDFS as CSV

```
df.write \
    .option("header", True) \
    .csv("hdfs://namenode:9000/user/panduka/employees_csv")
```

- ◆ Replace hdfs://namenode:9000/ with your actual **HDFS NameNode URI**.

### Example Output Location on HDFS

After writing, your HDFS directory (/user/panduka/employees\_csv/) will contain:

```
/user/panduka/employees_csv/
├── part-00000-* .csv
└── _SUCCESS
```

### Optional Settings

Option	Description
.mode("overwrite")	Overwrite existing data
.option("delimiter", "\t")	Set delimiter (e.g., tab-separated)
.partitionBy("department")	Partition output files by a column

```
df.write \  
    .option("header", True) \  
    .mode("overwrite") \  
    .csv("hdfs://namenode:9000/user/panduka/employees_csv")
```

#### 4. Verify in HDFS

Use the Hadoop CLI to verify:

```
hdfs dfs -ls /user/panduka/employees_csv  
hdfs dfs -cat /user/panduka/employees_csv/part-00000-*
```

# Handling data types in Spark DataFrames

## Why Data Types Matter in Spark

In Spark, **data types define** how data is **stored, processed, and optimized** across the cluster.

Handling data types properly:

- Avoids runtime errors
- Improves performance (via Catalyst & Tungsten)
- Enables better schema validation and query planning

## Viewing Data Types in a DataFrame

```
df.printSchema()
```

### Example

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()

data = [(1, "Alice", 25), (2, "Bob", 30)]
df = spark.createDataFrame(data, ["id", "name", "age"])
df.printSchema()
```

**Output:**

```
root
|-- id: long (nullable = true)
|-- name: string (nullable = true)
|-- age: long (nullable = true)
```

## Explicit Schema Definition

You can **define your own schema** using **StructType**

```
from pyspark.sql.types import StructType, StructField, IntegerType, StringType

schema = StructType([
    StructField("id", IntegerType(), True),
    StructField("name", StringType(), True),
    StructField("age", IntegerType(), True)
])

df = spark.createDataFrame(data, schema=schema)
```

## Cast Data Types (Change Types)

```
Use .cast() with .withColumn():
df = df.withColumn("age", df["age"].cast("string"))
```

## Infer Schema (on CSV, JSON, etc.)

When reading files

```
df = spark.read.option("header", True).option("inferSchema", True).csv("file.csv")
df.printSchema()
```

inferSchema=True lets Spark **guess** types from the data

**⚠ Extra scan step** — slower for big files

## Check and Handle Wrong Data Types

Example: Column supposed to be Integer but has string like "abc":

```
from pyspark.sql.functions import col
df = df.withColumn("age", col("age").cast("int"))
df.filter(col("age").isNull()).show()
```

## Use dtypes for Programmatic Type Access

```
df.dtypes
# [('id', 'int'), ('name', 'string'), ('age', 'int')]
```

## Complex Types

Spark supports **nested and complex types**:

Type	Example
<b>ArrayType</b>	Array of values
<b>MapType</b>	Key-value pairs
<b>StructType</b>	Nested fields (JSON-like)

Example:

```
from pyspark.sql.types import ArrayType
schema = StructType([
    StructField("id", IntegerType(), True),
    StructField("tags", ArrayType(StringType()), True)
])
```

## Change Multiple Columns Types

```
df = df.withColumn("age", col("age").cast("int")) \
    .withColumn("salary", col("salary").cast("double"))
```

## Writing Data with Types Preserved

If you want to **preserve schema** while writing (like in Parquet):

```
df.write.parquet("output_path")
```

## Summary Table: Data Type Handling

Task	Method	Notes
<b>View Schema</b>	df.printSchema()	Tree view of data types
<b>Define Schema</b>	StructType()	Use when reading or creating DF
<b>Infer Schema</b>	.option("inferSchema", True)	On reading CSV/JSON
<b>Cast Column Type</b>	df.withColumn(...cast(...))	Change a column's data type
<b>Check All dtypes</b>	df.dtypes	List of (col, type)
<b>Handle Type Errors</b>	Cast then filter nulls	.isNull() shows failed cast rows
<b>Complex Type Support</b>	Arrays, Structs, Maps	Use ArrayType, StructType, etc.

## Date and Timestamp in Spark

Type	Format Example	Use Case
<b>DateTime</b>	2025-08-02	Store calendar date only (no time)
<b>TimestampType</b>	2025-08-02 16:30:00	Store date and exact time (with timezone)

### Creating a DataFrame with Date or Timestamp

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import to_date, to_timestamp

spark = SparkSession.builder.getOrCreate()

data = [("2025-08-01", "2025-08-01 12:30:00")]
df = spark.createDataFrame(data, ["date_str", "timestamp_str"])

df = df.withColumn("date_col", to_date("date_str")) \
    .withColumn("timestamp_col", to_timestamp("timestamp_str"))

df.printSchema()
df.show()
```

#### Output:

```
root
|-- date_str: string
|-- timestamp_str: string
|-- date_col: date
|-- timestamp_col: timestamp
```

### Common Date Functions in PySpark

Function	Purpose	Example
<b>current_date()</b>	Today's date	df.withColumn("today", current_date())
<b>current_timestamp()</b>	Current timestamp	df.withColumn("now", current_timestamp())
<b>datediff()</b>	Days between two dates	datediff(col("end"), col("start"))
<b>date_add(), date_sub()</b>	Add or subtract days	date_add(col("date"), 5)
<b>months_between()</b>	Get fractional months between dates	months_between("end", "start")
<b>year(), month(), dayofmonth()</b>	Extract date parts	year(col("date"))
<b>hour(), minute(), second()</b>	Extract time parts	hour(col("timestamp"))

## Convert String to Date/Timestamp with Format

```
from pyspark.sql.functions import to_date, to_timestamp

df = spark.createDataFrame([("02-08-2025",)], ["date_str"])
df = df.withColumn("date_col", to_date("date_str", "dd-MM-yyyy"))
df.show()
```

## Filtering with Dates

```
from pyspark.sql.functions import lit

df.filter(df["date_col"] > lit("2025-08-01")).show()
```

## Handling Nulls & Bad Formats

```
df = spark.createDataFrame([("2025-08-01",), ("invalid-date",)], ["dt"])
df = df.withColumn("parsed", to_date("dt", "yyyy-MM-dd"))
df.show()
```

**Output:**

```
+-----+-----+
| dt      | parsed    |
+-----+-----+
| 2025-08-01 | 2025-08-01 |
| invalid-date | null    |
```

## Sorting by Date

```
df.orderBy("date_col").show()
```

## Writing Dates to Files (CSV, Parquet)

Spark **automatically preserves date types** in formats like **Parquet**, but not in **CSV**.

```
df.select("date_col", "timestamp_col").write.parquet("hdfs://.../dates_output")
```

For CSV:

```
df.select("date_col", "timestamp_col") \
  .write.option("header", True) \
  .csv("hdfs://.../dates_output")
```

## Tips for Working with Dates

Tip	Why Important?
<b>Always convert string dates using to_date()</b>	Ensures correct type and format
<b>Use correct format strings (dd-MM-yyyy, etc.)</b>	Avoid parsing errors
<b>Use current_date() and current_timestamp()</b>	Dynamic date/time columns in ETL pipelines
<b>Avoid using string comparison for dates</b>	Spark won't optimize string comparisons as well

# Spark SQL and Spark Tables

## What is Spark SQL?

**Spark SQL** is a module in Apache Spark for structured data processing.

It allows you to **query structured data using SQL**, as well as **DataFrame APIs**.

- ✓ You can write SQL queries just like you do in traditional databases (SELECT \* FROM table WHERE ...), but Spark runs them distributedly and fast!

## Why use Spark SQL?

<input checked="" type="checkbox"/> Benefit	 Explanation
Familiar	You can use standard SQL syntax — great for data analysts & SQL-savvy users.
Interoperable	Works seamlessly with DataFrames and Datasets. You can convert between them.
Optimized	Queries are compiled to an efficient <b>execution plan</b> via Catalyst optimizer.
Hive-compatible	You can integrate with Hive, OR use Spark SQL as your lightweight data warehouse.

## Using SQL on Spark Data

To use SQL, **register your DataFrame as a temporary or persistent table**, then run SQL on it.

## Creating Spark Tables from DataFrames

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("SparkSQLExample").getOrCreate()

data = [("Alice", "HR", 3000), ("Bob", "IT", 4000)]
df = spark.createDataFrame(data, ["name", "dept", "salary"])
```

## Temporary Table (Session-SScoped)

```
df.createOrReplaceTempView("employees")

spark.sql("SELECT * FROM employees WHERE salary > 3000").show()
```

## Temporary Table

- **Session-scoped:** Only available during the current Spark session.
- **In-memory only:** Not saved to disk.
- Automatically removed when session ends.

## Global Temporary Table

```
df.createOrReplaceGlobalTempView("employees")
spark.sql("SELECT * FROM global_temp.employees WHERE dept = 'HR'").show()
```

### Global Temp Table

- Shared across sessions using **global\_temp database**.
- Accessed like: `SELECT * FROM global_temp.tablename`
- Still in-memory and not persisted to disk.
- Useful in notebooks or multi-session apps.
- If need to access from another spark session , must go to the `global\_temp` to find global temp databases.

## Persistent Table (Managed Table)

```
df.write.mode("overwrite").saveAsTable("spark_employees")
```

### Persistent Table

- Stored in Spark warehouse directory (spark-warehouse).
- **Survives session restarts**.
- Schema and data are saved to disk.
- Can use `spark.sql("SELECT * FROM spark_employees")` anytime.

## Table Type Comparison Table

Feature	Temp View	Global Temp View	Persistent Table
<b>Scope</b>	Current Session	Across Sessions	Permanent
<b>Backed by Disk</b>	<input type="checkbox"/> No (in-memory)	<input type="checkbox"/> No (in-memory)	<input checked="" type="checkbox"/> Yes
<b>Prefix Required in Query</b>	<input type="checkbox"/> (direct)	<input checked="" type="checkbox"/> <code>global_temp.</code> required	<input type="checkbox"/> (unless in a specific DB)
<b>Automatic Cleanup</b>	Yes (session end)	Yes (app termination)	No
<b>Good For</b>	One-off analysis	Sharing across notebooks	Long-term, reusable datasets

## Persistent Table Storage Location

By default, persistent tables are saved to:

`<your_project>/spark-warehouse/`

But you can configure the location using

```
spark.conf.set("spark.sql.warehouse.dir", "/custom/warehouse/path")
```

## Notes for Beginners

- You **must create a SparkSession** with Hive support enabled if you want to use full Hive SQL features or external tables.
- If you restart the Spark session, **temporary views are lost** but persistent tables remain.
- **Global temporary views** live until the Spark application terminates.

## Summary

Concept	Description
<b>Spark SQL</b>	Module for querying structured data with SQL
<b>Temp Table</b>	In-memory, session-limited
<b>Global Temp</b>	In-memory, app-limited, accessible across sessions
<b>Persistent</b>	Disk-backed, survives restarts, stored in warehouse

## Spark SQL operations

### Start Spark Session

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName("SparkSQLEExample") \
    .config("spark.sql.warehouse.dir", "/user/hive/warehouse") \
    .enableHiveSupport() \
    .getOrCreate()
```

### Create a Sample DataFrame

```
data = [
    (1, "Alice", "HR", 3500),
    (2, "Bob", "IT", 4000),
    (3, "Cathy", "Finance", 4200),
    (4, "David", "HR", 3000),
    (5, "Eva", "IT", 5000)
]

columns = ["id", "name", "department", "salary"]

df = spark.createDataFrame(data, columns)
df.show()
+---+-----+-----+
| id| name|department|salary|
+---+-----+-----+
| 1|Alice|        HR| 3500|
| 2| Bob|        IT| 4000|
| 3|Cathy|    Finance| 4200|
| 4|David|        HR| 3000|
| 5| Eva|        IT| 5000|
+---+-----+-----+
```

### Register DataFrame as Temporary SQL Table

```
df.createOrReplaceTempView("employees")
Now you can run SQL queries using spark.sql("...").
```

## SQL Operations Examples

### a) Simple SELECT with WHERE

```
spark.sql("SELECT * FROM employees WHERE salary > 3500").show()
+---+-----+-----+
| id| name|department|salary|
+---+-----+-----+
| 2| Bob|        IT| 4000|
| 3|Cathy|    Finance| 4200|
| 5| Eva|        IT| 5000|
+---+-----+-----+
```

## b) Aggregation with GROUP BY

```
spark.sql("""
    SELECT department, AVG(salary) as avg_salary
    FROM employees
    GROUP BY department
""").show()
+-----+-----+
|department|avg_salary|
+-----+-----+
|      IT|    4500.0|
|      HR|    3250.0|
| Finance|    4200.0|
+-----+-----+
```

## c) ORDER BY

```
spark.sql("SELECT * FROM employees ORDER BY salary DESC").show()
+---+---+-----+-----+
| id| name|department|salary|
+---+---+-----+-----+
|  5| Eva|      IT|  5000|
|  3| Cathy|  Finance|  4200|
|  2| Bob|      IT|  4000|
|  1| Alice|     HR|  3500|
|  4| David|     HR|  3000|
+---+---+-----+-----+
```

## d) JOIN Example

Let's create another DataFrame with department head names

```
dept_heads = [
    ("HR", "Susan"),
    ("IT", "Tom"),
    ("Finance", "Karen")
]

dept_df = spark.createDataFrame(dept_heads, ["department", "head"])
dept_df.createOrReplaceTempView("dept_heads")
```

Then join with employees

```
spark.sql("""
    SELECT e.name, e.department, e.salary, d.head as dept_head
    FROM employees e
    JOIN dept_heads d ON e.department = d.department
""").show()
+-----+-----+-----+-----+
| name|department|salary|dept_head|
+-----+-----+-----+-----+
|Cathy|  Finance|  4200|    Karen|
|David|     HR|  3000|    Susan|
|Alice|     HR|  3500|    Susan|
|  Eva|      IT|  5000|     Tom|
|  Bob|      IT|  4000|     Tom|
+-----+-----+-----+-----+
```

## Save the Result as a Persistent Table

```
result_df = spark.sql("""  
    SELECT department, COUNT(*) as num_employees, AVG(salary) as avg_salary  
    FROM employees  
    GROUP BY department  
""")  
  
result_df.write.mode("overwrite").saveAsTable("dept_summary")
```

Now you can query it later

```
spark.sql("SELECT * FROM dept_summary").show()
```

## Optional Cleanup

```
spark.sql("DROP TABLE IF EXISTS dept_summary")
```

```
spark.sql('show tables').show()  
+-----+-----+  
|namespace| tableName|isTemporary|  
+-----+-----+-----+  
|          |dept_heads|      true|  
|          | employees|      true|  
+-----+-----+
```

# Spark SQL Tables: Managed vs External

In Spark, you can create **two types of tables** using Spark SQL:

Type	Also Called As
Managed Table	Internal Table
External Table	Unmanaged Table

## Key Differences

Feature	Managed Table	External Table
Location	Spark manages data location (usually in warehouse dir)	You define the exact data path manually
Table Metadata	Stored in metastore	Stored in metastore
Data Ownership	Spark owns and manages the data	You manage the data
DROP TABLE behavior	Deletes <b>both</b> table metadata and data	Deletes <b>only metadata</b> , data remains in HDFS/S3
Use Case	When Spark should control the full lifecycle	When data is shared or used by multiple tools/systems
Default Location	<code> \${spark.sql.warehouse.dir}/table_name</code>	Custom path (e.g., <code>/user/data/sales/</code> )

## Example – Create Managed Table

```
CREATE TABLE sales_managed (
    id INT,
    product STRING,
    amount DOUBLE
)
USING parquet
```

- Spark saves both data and metadata.
- Dropping this table removes the underlying Parquet files too.

## Example – Create External Table

```
CREATE TABLE sales_external (
    id INT,
    product STRING,
    amount DOUBLE
)
USING parquet
OPTIONS (path "/user/data/sales/")
```

- Spark only manages metadata.
- The Parquet files remain even if the table is dropped.

## When to Use

Situation	Table Type
Temporary, sandboxed, or internal-only processing	<input checked="" type="checkbox"/> Managed
Data shared across systems, tools, or used outside Spark	<input checked="" type="checkbox"/> External
You want full control over file location and layout	<input checked="" type="checkbox"/> External
You want Spark to clean up after dropping the table	<input checked="" type="checkbox"/> Managed

## DROP Behavior Demo

-- Managed Table

```
DROP TABLE sales_managed;
```

-- ⚡ Deletes both metadata and actual files

-- External Table

```
DROP TABLE sales_external;
```

-- 💣 Deletes only metadata, files remain untouched

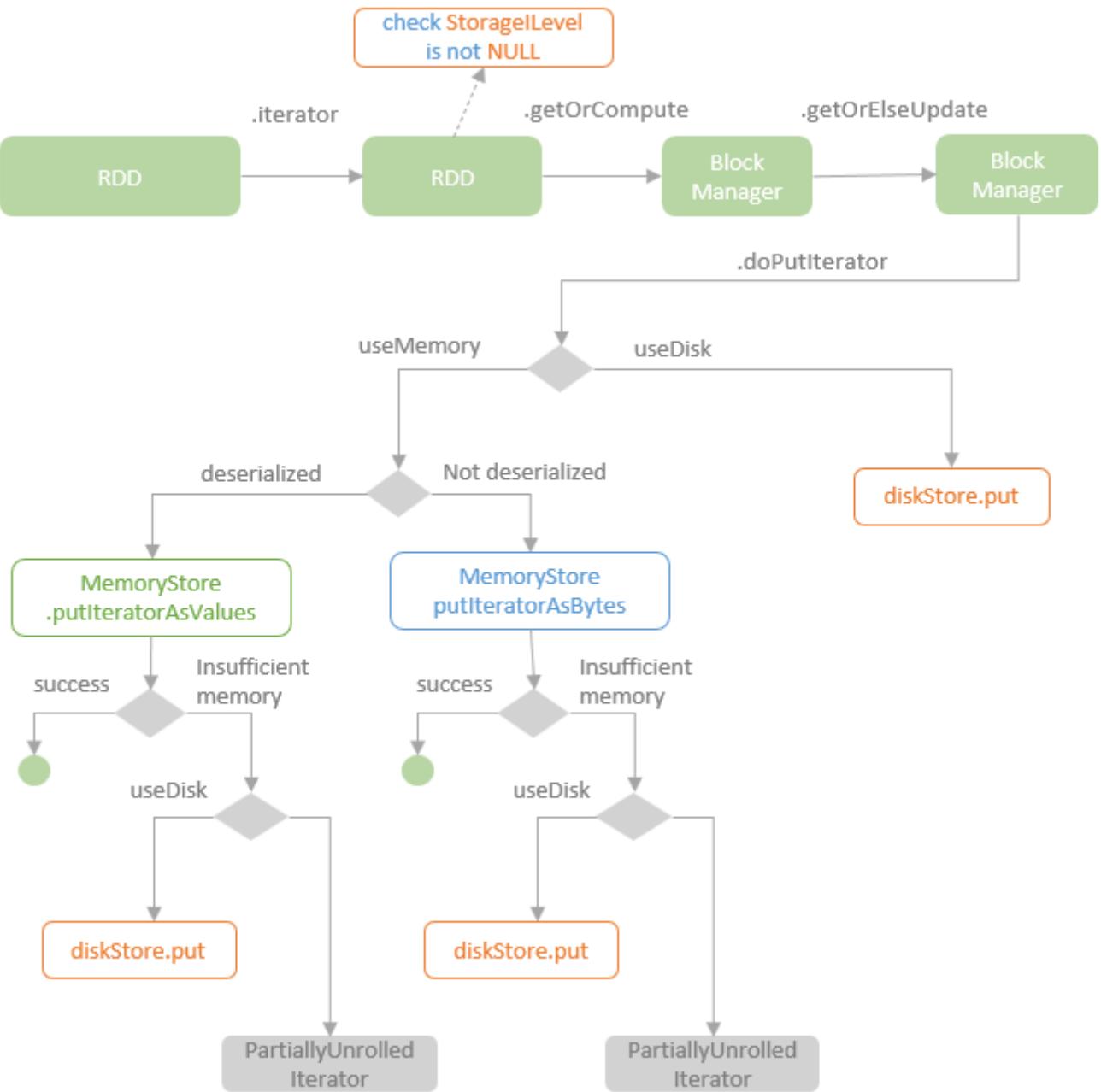
## Bonus: Show Table Type

To check whether a table is managed or external:

```
DESCRIBE EXTENDED table_name
```

Look for the field:

Type: MANAGED or EXTERNAL



# Caching and Persisting in Spark

## Why Caching or Persisting Is Needed in Spark?

Even though **Spark performs in-memory computations, caching/persisting is necessary** in the following cases:

Scenario	Why Caching/Persisting Helps
Reusing the same RDD/DataFrame multiple times	Without caching, Spark recomputes it each time from the source
Expensive computations	Saves time by avoiding recomputation
Iterative algorithms (e.g. ML)	Data is reused across iterations
Actions like <code>.count()</code> , <code>.show()</code> , <code>.collect()</code> on same DF	Triggers multiple jobs unless cached

💡 Spark is *lazy*, so without caching, every action can **trigger full recomputation** from the beginning!

## Cache vs Persist – Key Differences

Feature	cache()	persist()
Default Storage Level	MEMORY_AND_DISK	Must specify or uses MEMORY_AND_DISK by default
Control over storage level	✗ Not configurable	✓ You can define storage levels (e.g., disk-only, memory-only)
Ease of Use	✓ Simple to use	⚠ Slightly more complex due to level selection
Replication	✗ No built-in replication	✓ Possible with certain storage levels
Serialization	✗ No custom serialization	✓ You can use serialized formats
When to Use	When you're okay with default memory+disk	When you want fine control over memory/disk/replication/serialization

Table: Spark Storage Levels

Storage Level	Description	Stored Where	Fault Tolerant?
MEMORY_ONLY	Keeps as many partitions as fit in memory, rest recomputed	RAM only	✗ No
MEMORY_AND_DISK (default)	Tries RAM, spills to disk if not enough	RAM + Disk	✓ Yes
MEMORY_ONLY_SER	Same as MEMORY_ONLY, but uses serialized format	RAM (Serialized)	✗ No
MEMORY_AND_DISK_SER	RAM if possible, else disk; serialized	RAM + Disk (Serialized)	✓ Yes
DISK_ONLY	Stores all partitions on disk	Disk only	✓ Yes

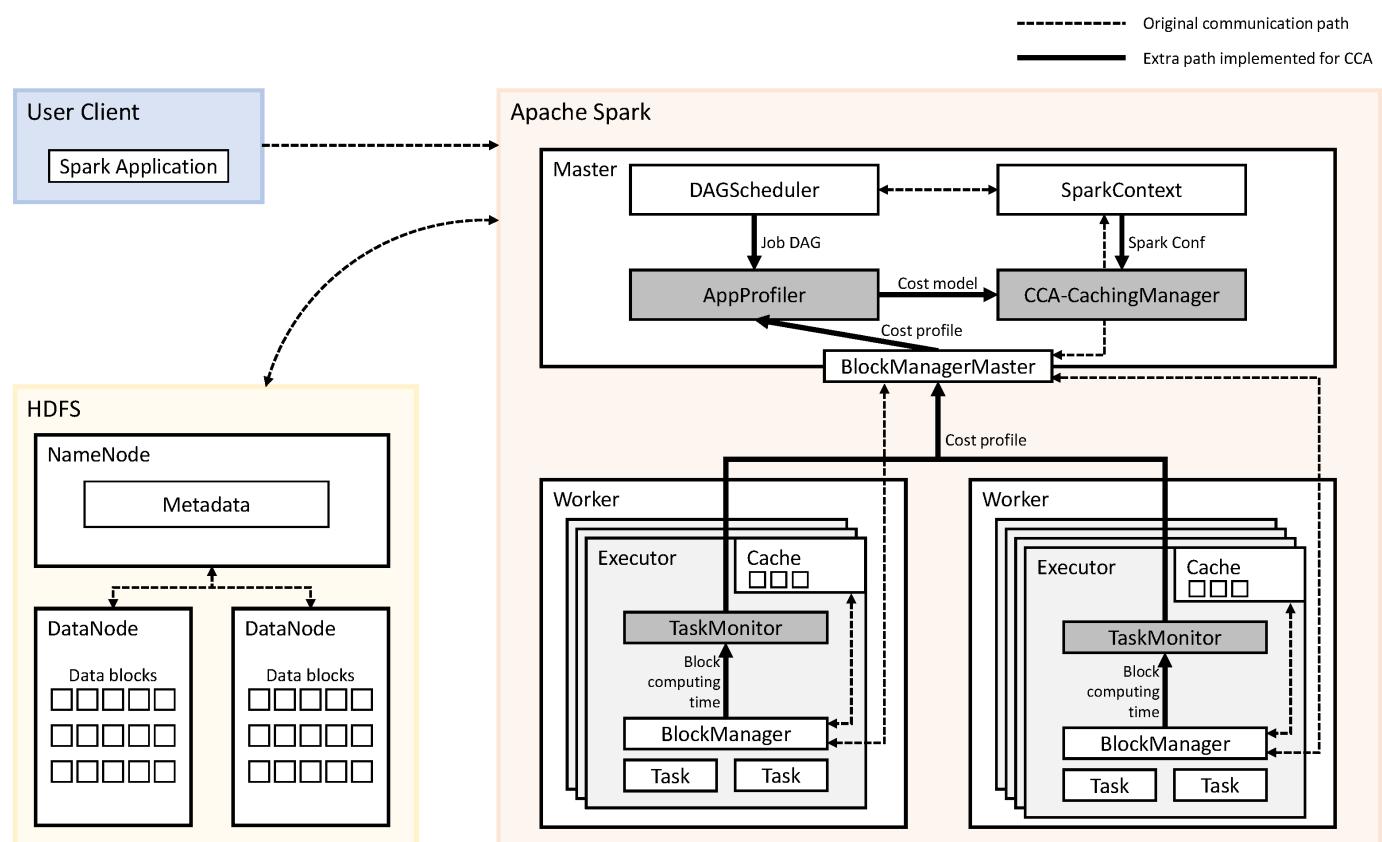
MEMORY_ONLY_2, etc.	Replicates on 2 nodes	RAM (Replicated)	<input checked="" type="checkbox"/> Yes
OFF_HEAP	Uses off-heap memory (Tungsten project, experimental)	External RAM	<input checked="" type="checkbox"/> Yes

## Visual Summary

Operation	Default Storage Level	User Control	Spill to Disk	Can Replicate	Serializable Format
.cache()	MEMORY_AND_DISK	<input type="checkbox"/> No	<input checked="" type="checkbox"/> Yes	<input type="checkbox"/> No	<input type="checkbox"/> No
.persist()	Customizable	<input checked="" type="checkbox"/> Yes	<input checked="" type="checkbox"/> (if set)	<input checked="" type="checkbox"/> (if set)	<input checked="" type="checkbox"/> (if set)

## Pro Tip

- Always **unpersist** your data when no longer needed to free memory.
- Use .persist(StorageLevel.DISK\_ONLY) when memory is very limited.
- Use .cache() when working interactively and just want the default behavior.



## How and Where Does Caching Happen in Spark?

Aspect	Details
Where	Caching happens on <b>executors' local memory (RAM)</b> . If memory isn't enough, Spark can <b>spill</b> data to the <b>local disk</b> (not HDFS).
How	When you use <code>.cache()</code> or <code>.persist()</code> , Spark materializes the RDD/DataFrame <b>after an action</b> (like <code>.count()</code> or <code>.show()</code> ), and <b>stores</b> the partition data for reuse.
Trigger Point	Caching is <b>lazy</b> , meaning it only happens <b>after the first action is triggered</b> .
Eviction	If memory is low, Spark <b>evicts least-used cached blocks</b> using <b>LRU (Least Recently Used)</b> policy.

## Why Spark Uses Local Disk, Not HDFS for Caching?

Reason	Explanation
Speed	Local disk I/O is much faster than HDFS (which is distributed and higher-latency).
Control	Spark manages cached data internally. Writing to HDFS would introduce replication, block placement, and network overhead.
Temporary	Cache data is temporary and used <b>only within the current Spark application</b> – no need to persist across sessions like HDFS.
Performance	HDFS is optimized for throughput, not latency; caching needs <b>fast access</b> , so local disk and memory are preferred.

## When to Use Caching

Situation	Why Caching Helps
Multiple Actions on Same DataFrame	Avoids recomputation each time (e.g., <code>.count()</code> , <code>.show()</code> , <code>.collect()</code> ...)
Iterative Algorithms (ML)	Data is reused across iterations (e.g., gradient descent, k-means)
Expensive Computation	Save time on re-executing long-running jobs (joins, aggregations, filtering)
Interactive Queries (e.g., Notebooks)	Faster responses for exploratory data analysis

## When Not to Use Caching

Situation	Why You Should Avoid
<b>Low-memory Environment</b>	Can lead to out-of-memory errors or heavy disk spilling, slowing performance
<b>One-time Use of DataFrame/RDD</b>	Caching adds overhead; better to recompute once than store unnecessarily
<b>Large Datasets That Don't Fit in Memory/Disk</b>	Spillover and eviction will <b>reduce performance</b>
<b>Cluster Has Weak Local Disks</b>	Slow disks can make caching worse than recomputing

## Where Can You See Cached Data in Spark?

Tool/Location	What You See
<b>Spark UI → Storage Tab</b>	Shows: RDD/DataFrame name, Storage Level, Memory size used, # Cached Partitions
<b>Web UI URL</b>	Usually at <code>http://&lt;driver-node&gt;:4040</code>
<b>df.is_cached (in PySpark)</b>	Returns True if cached
<b>spark.catalog.isCached("table_name")</b>	For tables

## Example: Caching in PySpark

```
df = spark.read.csv("hdfs://data/transactions.csv", header=True, inferSchema=True)

# Cache it
df.cache()

# Trigger an action to materialize it
df.count()

# Verify if cached
print(df.is_cached) # Output: True

# Unpersist if no longer needed
df.unpersist()
```

## Does Caching Decrease Performance?

Yes — **when misused**, caching can **hurt performance**:

Cause	Effect
Memory Pressure	Can lead to <b>GC overhead, disk spill, or OOM errors</b>
Spilled to Disk	Local disk access is slower than recomputing small tasks
Too Much Cached Data	Overwhelms executor memory → triggers eviction, repeated recomputations
No Use of Cached Data	Adds storage overhead without benefits

## Best Practices

 Tip	<input checked="" type="checkbox"/> Recommended
Use <code>.cache()</code> only when data is reused	<input checked="" type="checkbox"/>
Use <code>.persist(StorageLevel)</code> if you need control	<input checked="" type="checkbox"/>
Always unpersist after usage	<input checked="" type="checkbox"/>
Monitor cache usage via Spark UI	<input checked="" type="checkbox"/>
Avoid caching huge data on small clusters	<input checked="" type="checkbox"/>
Don't cache unless there's actual reuse	<input checked="" type="checkbox"/>

## Caching RDDs

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

```
customers_rdd = spark.sparkContext.textFile('/tmp/customers_350mb.csv')
```

```
customers_filtered = customers_rdd.filter(lambda row : 'Mumbai' in row)  
customers_mapped = customers_filtered.map(lambda row : (row.split(',')[0],1))  
customers_reduced = customers_mapped.reduceByKey(lambda x,y:x+y)
```

```
customers_reduced.count() # Slow
```

```
661241
```

```
customers_reduced.cache()
```

```
# Cache this mannually. Normally step before end is cached automatically.
```

```
****This will be shown in spark-ui DAG as a green dot
```

```
customers_reduced.count() # Now its fast due to last step caching
```

```
661241
```

```
customers_reduced.unpersist() # Uncache
```

```
customers_reduced.collect()
```

```
[('7', 1),  
 ('53', 1),  
 ('70', 1),  
 ('117', 1),  
 ('158', 1),  
 ('192', 1),  
 ('248', 1),  
 ('268', 1),  
 ('297', 1),
```

```
spark.stop()
```

## Caching Data-frames

```
from pyspark.sql import SparkSession

spark = SparkSession.builder \
    .appName('DataFrame_caching_demo') \
    .enableHiveSupport() \
    .getOrCreate()
```

```
customers_df= spark.read.option('header','true').csv('/tmp/customers_500mb.csv')
customers_df.printSchema()
```

```
customers_df.count()
```

```
customers_df.cache() # ->customers_df.cache().show()  <~> some platforms this is lazy
```

```
customers_df.count()
```

```
customers_df.show(5)
```

```
customers_df.show(5) # ->fast
```

```
tail_df.unpersist()
```

```
customers_df.unpersist()
```

```
tail_df = customers_df.orderBy('customer_id',ascending=False)
tail_df.show(5)
```

```
tail_df.cache()
```

```
tail_df.show(5)
```

```
tail_df.show(5)
```

## Spark Caching is Lazy

```
customers_df.cache()
```

- This **marks the DataFrame for caching**, but **nothing is cached yet**.
- Spark will cache partitions **only when they are actually computed** through an *action* (like `.count()`, `.show()`, etc.).
- So if only the first few partitions are touched by `.show()`, only those will be cached.

## How is Cache Built (Internally)?

When an action like `.count()` or `.show()` triggers execution:

1. **Spark evaluates the lineage (DAG)** to compute results.
2. While computing, Spark checks:  
“Is this DataFrame marked for cache?”
3. If yes:
  - It **writes the result of each partition to the cache storage layer after computing**.
  - Spark uses **BlockManager** to store blocks in memory.
4. Subsequent actions on the same DataFrame reuse the cached blocks.

Spark **does not pre-fetch or pre-cache all partitions** upfront. Only what's accessed gets cached.

## Why only first partitions cached?

When you ran:

```
customers_df.cache()  
customers_df.count()
```

- This computed **all partitions** → all were **cached** in memory (or disk if memory was insufficient).

But:

```
customers_df.cache()  
customers_df.show(5)
```

- This only accessed **1 or 2 partitions** (enough to show 5 rows).
- So, only those partitions got cached.
- Remaining partitions are still **uncached**.

## Can I Cache Specific Partitions?

Not directly. Caching works **at the DataFrame level**.

But you can simulate this

```
df_filtered = customers_df.filter("some_column = some_value").cache()  
df_filtered.count()
```

Now only partitions that match the filter are accessed/cached.

## Where is Cache Stored?

**Spark Cache Storage Layers (in order of priority):**

Storage Level	Memory	Disk	Notes
<b>MEMORY_AND_DISK (default)</b>	✓	✓	Cached in memory, spilled to disk if full
<b>MEMORY_ONLY</b>	✓	✗	Fastest but risky if memory overflows
<b>DISK_ONLY</b>	✗	✓	Safe but slower

Spark uses **local disk** for caching because:

- It's much faster to access than HDFS.
- BlockManager stores RDD/DataFrame blocks locally for reuse.

## Where Can You See Cached Data?

In **Spark UI** (usually at <http://<driver-node>:4040>):

- Go to the **Storage** tab.
- You'll see the cached DataFrame:
  - How many partitions are cached.
  - Memory used.
  - Disk used.
  - Storage level.

## When NOT to Use Caching?

Scenario	Why Not Cache?
<b>One-time operation</b>	Caching adds overhead, no benefit
<b>DataFrame used in only one action</b>	Wasted memory
<b>Too large to fit in memory</b>	Might cause <b>GC pressure</b> , disk I/O
<b>You need fresh results every time</b>	Cache stores old state unless refreshed

## Example: Faster Second .show()

```
tail_df.cache()  
tail_df.show(5) # Triggers cache creation for needed partitions  
tail_df.show(5) # Reuses cached data – very fast
```

- The first .show(5) computes and caches 1-2 partitions.
- The second .show(5) reads from cache → fast execution.

## Cache Built at Stages

Each **stage in the DAG** corresponds to a logical operation (e.g., filter, sort). Example:

```
tail_df = customers_df.orderBy('customer_id', ascending=False)  
tail_df.cache()  
tail_df.show(5)
```

Execution flow

1. **Stage 1:** Load CSV, parse into DataFrame.
2. **Stage 2:** Apply orderBy → triggers shuffle and repartition.
3. While executing show(5), Spark:
  - Computes sorted partitions.
  - **Stores those partitions** in the BlockManager (RAM or disk).
4. .show() uses only required partitions.

## Unpersisting

```
customers_df.unpersist()  
• Manually frees up memory/disk blocks held by cached data.
```

\*\*\*If we use filter() to get ids between 40000 to 100000, which may in partition 3-4, it will read all the partitions and cache all the partitions.

## Caching on Spark SQL tables

### Why Cache a Table in Spark SQL?

- Avoid recomputing results for repeated SQL queries
- Improve performance on interactive or exploratory workloads
- Enable reuse of results in complex multi-step SQL workflows (like BI dashboards or ML pipelines)

### Syntax to Cache a Table

You can cache both temporary and permanent tables.

```
-- Cache a table
CACHE TABLE table_name;

-- Or with PySpark
spark.sql("CACHE TABLE table_name")

-- Cache a query result as a temporary view
spark.sql("""
    CACHE TABLE high_value_customers AS
    SELECT * FROM customers WHERE total_purchase > 10000
""")
```

Or if you're using a DataFrame:

```
df.createOrReplaceTempView("my_view")
spark.sql("CACHE TABLE my_view")
```

To uncache:

```
UNCACHE TABLE table_name;
```

### Where Is the Cached Table Stored?

- Cached tables are stored **in memory (RAM)** on executors using **in-memory columnar format**.
- If memory is insufficient, Spark **drops partitions** (LRU) and recomputes them on access.
- Spark does **not** persist table cache to disk unless you use `.persist()` with a disk-based storage level.

### Internals – How Caching Works for Spark SQL Tables

When you run `CACHE TABLE`:

1. Spark **registers** the table in its **in-memory catalog** as cached.
2. Spark doesn't immediately cache the data—it does **lazy caching**.
3. The table is actually **cached on first access** (first action like `SELECT`, `SHOW`, etc.).
4. The **logical plan** is stored and its physical results are cached on partitions when executed.

## Example: Caching SQL Table vs Not Caching

```
# Load and register table
df = spark.read.csv("/tmp/customers.csv", header=True)
df.createOrReplaceTempView("customers")

# Execute query without caching
spark.sql("SELECT COUNT(*) FROM customers").show() # Slow

# Cache the table
spark.sql("CACHE TABLE customers")

# Trigger cache
spark.sql("SELECT COUNT(*) FROM customers").show() # Triggers caching and stores

# Second call – now fast!
spark.sql("SELECT COUNT(*) FROM customers").show()
```

## Check Cached Tables and Their Details

```
# List cached tables
spark.catalog.isCached("customers") # Returns True/False

# List all cached tables
spark.catalog.listTables()

# Clear all cached tables
spark.catalog.clearCache()
```

## Things to Remember / Best Practices

Aspect	Explanation
⚡ When to use	Repeated access to large tables, dashboards, joins, aggregations
🧠 When not to use	One-time access, huge tables (won't fit in memory), unstable workloads
✗ Risk	Caching very large tables can lead to memory eviction, spilling, or OOM
♻️ Replace cache	After updating table content, always call UNCACHE TABLE then CACHE TABLE again
_PARTITION_AWARENESS	Caching happens <b>per partition</b> . Spark only materializes the partitions that are touched
🚫 Cache invalidation	Caching <b>does not auto-refresh</b> if the underlying data/table is updated
💾 No disk fallback	Unlike persist(StorageLevel.DISK_ONLY), CACHE uses MEMORY_AND_DISK by default but disk is fallback, not persisted

## Advanced Usage: Cache Temp View from SQL

```
# Create cached view
spark.sql("""
    CREATE OR REPLACE TEMP VIEW active_customers
    AS SELECT * FROM customers WHERE is_active = 'True'
""")
spark.sql("CACHE TABLE active_customers")

# Use it in another query
spark.sql("""
    SELECT state, COUNT(*)
    FROM active_customers
    GROUP BY state
""").show()
```

## DAG Behavior with SQL Caching

In the **Spark DAG**, caching a SQL table acts like a **checkpoint node**. Once computed, its stage is **not recomputed** in future actions. You'll see the DAG skip earlier stages for the same table if it's cached.

Aspect	Lazy Caching (default)	Eager Caching (manual trigger)
<b>Definition</b>	Caching is <b>registered</b> , but data isn't computed or stored until needed	Data is <b>cached immediately</b> by triggering an action
<b>Trigger</b>	First action (e.g., .show(), .count()) causes data to be cached	You explicitly call .cache() or CACHE TABLE <b>followed by</b> an action
<b>Example</b>	df.cache() – does nothing until you call df.count() or similar	df.cache(); df.count() — triggers actual caching
<b>Performance on 1st Access</b>	Initial action is <b>slow</b> (includes computation + caching)	Initial action already done, so subsequent actions are fast
<b>Subsequent Actions</b>	Faster than first, as data is already cached	Fast (already in memory)
<b>Control</b>	More <b>declarative</b> (let Spark decide when to cache)	More <b>imperative</b> (you choose when caching happens)
<b>When to Use</b>	When you're not sure whether the data will be used again	When you <b>know</b> you'll reuse the result repeatedly
<b>DAG Behavior</b>	Cache stage is injected after 1st action	Cache stage is <b>injected immediately</b> into DAG after action
<b>Storage Level</b>	Default: MEMORY_AND_DISK	Same (MEMORY_AND_DISK) unless explicitly persisted with custom level
<b>Resource Usage</b>	Can lead to recomputation if cache not yet triggered	Ensures memory usage is justified (already cached when used)

Sql table -> eager caching

Dataframe -> lazy

# Spark Architecture

At its core, Apache Spark is a **distributed data processing engine** based on **master-slave architecture**, where:

- **Driver Program:** Coordinates execution (your main Spark app).
- **Cluster Manager:** Allocates resources (CPU, memory) to Spark applications.
- **Executors:** Run the code and return results to the driver.
- **Tasks:** Units of work sent to executors.
- **Jobs → Stages → Tasks:** Spark breaks transformations into this hierarchy.

## Spark Run Modes (Deployment Modes)

### 1. Local Mode

- **Use Case:** Development or testing on your laptop.
- **No cluster manager needed.**
- Example:
- `spark-submit --master local[*] app.py`

### 2. Standalone Cluster Mode

- **Built-in cluster manager** provided by Spark.
- Simple setup for small to medium deployments.
- You manually start:
  - One **Master Node**
  - Multiple **Worker Nodes**
- Spark submits jobs to the Master, which assigns them to Workers.

#### Submit Example:

```
spark-submit --master spark://<master-host>:7077 app.py
```

### 3. YARN (Hadoop Cluster Manager)

- **Hadoop-based resource manager.**
- Ideal for large-scale production deployments.
- Spark can run **inside YARN containers**.
- YARN handles resource isolation and multi-tenancy.

## Modes in YARN

Mode	Description
Client	Driver runs on local machine (your laptop or edge node).
Cluster	Driver runs inside the YARN cluster (recommended for production).

Submit Example:

```
spark-submit --master yarn --deploy-mode cluster app.py
```

## 4. Apache Mesos

- General-purpose cluster manager.
- Allows Spark, Hadoop, Kafka, etc., to share the same resources.
- Less popular now, largely replaced by Kubernetes.

## 5. Kubernetes

- **Modern container-based orchestration** platform.
- Spark supports running on Kubernetes clusters using Docker images.
- Offers **scalability, containerization, and auto-healing**.

Submit Example:

```
spark-submit --master k8s://https://<k8s-master>:6443 --deploy-mode cluster ...
```

## Comparison Table of Cluster Managers

Feature	Standalone	YARN	Mesos	Kubernetes
Built-in	✓ Yes	✗ No	✗ No	✗ No
Setup Complexity	Easy	Medium	Medium	High
Scalability	Medium	High	High	Very High
Multi-tenancy	Basic	Advanced (via Hadoop queues)	Advanced	Advanced (via namespaces)
Isolation	Limited	Strong	Strong	Very Strong (Docker-based)
Best for	Simple Spark-only workloads	Hadoop ecosystem integration	Multi-framework environments	Cloud-native Spark environments

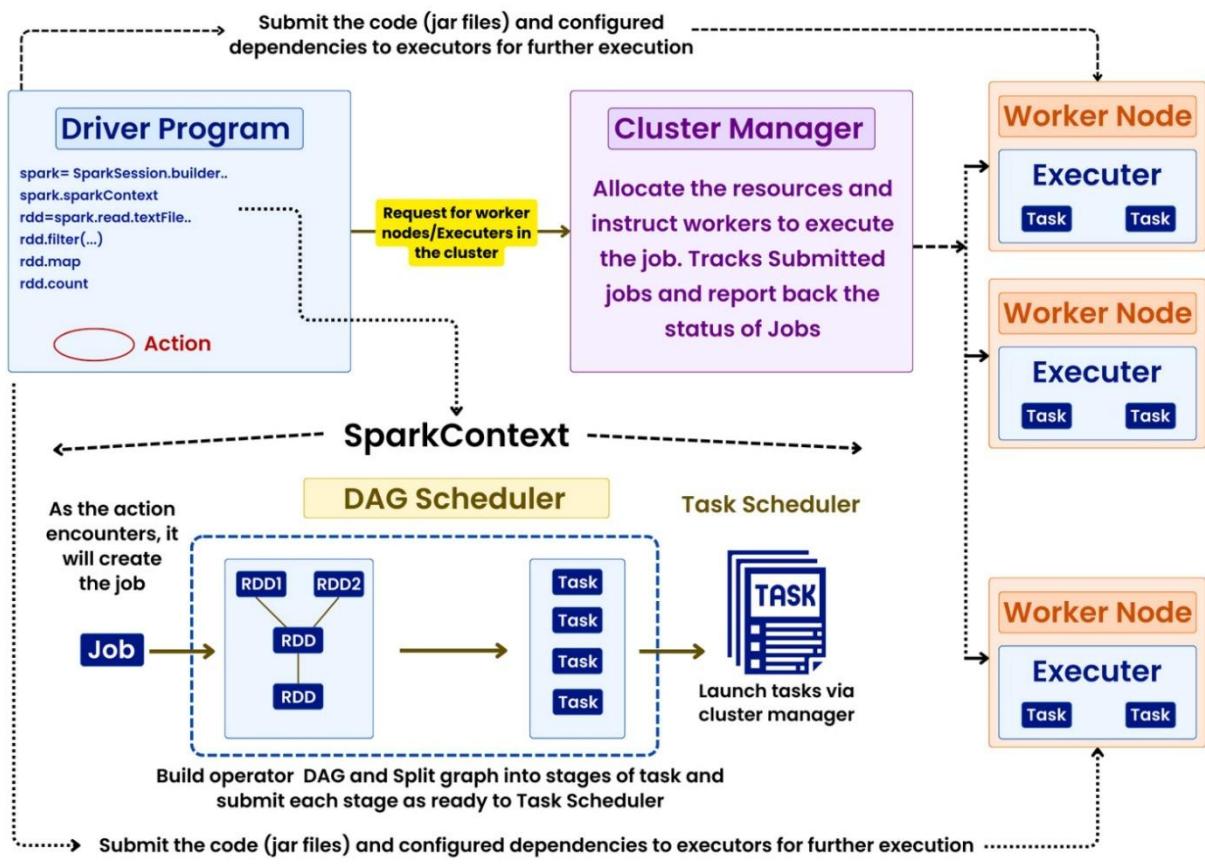
## When to Use What?

Use Case	Recommended Mode
Local development	Local[*]
Small internal Spark clusters	Standalone
Hadoop-based production workloads	YARN (Cluster mode)
Containerized/cloud-native pipelines	Kubernetes



## Internals of Job Execution in Spark

Created by :  Rocky Bhatia [in](#)



## What Makes Spark Fast?

Feature	How It Helps
<b>1. In-memory computation</b>	Avoids slow disk I/O by storing intermediate results in memory.
<b>2. DAG execution engine</b>	Optimizes the execution plan as a Directed Acyclic Graph (DAG).
<b>3. Lazy Evaluation</b>	Builds logical plan first, optimizes, then executes only when needed.
<b>4. Parallel processing (RDD partitioning)</b>	Splits data into partitions and distributes across executors for parallel work.
<b>5. Pipeline transformations</b>	Chains narrow operations to avoid intermediate disk writes.
<b>6. Columnar format (DataFrames/Datasets)</b>	Optimized memory layout and CPU cache usage.
<b>7. Predicate Pushdown and Vectorized I/O</b>	Reduces the amount of data read from storage.

### Example: $2^8 = 256$ — A Computational View

Let's use this small example to compare **MapReduce vs Spark** execution model:

#### 👉 Imagine You Want to Calculate:

result =  $2^8 = 2 * 2 * 2 * 2 * 2 * 2 * 2 * 2$

#### 🔴 In MapReduce:

- **Each multiplication is a separate map task**, and after each task, the intermediate result is written to disk (HDFS).
- For example:
  - $2 * 2 \rightarrow$  write to HDFS
  - read from HDFS  $\rightarrow * 2 \rightarrow$  write again
  - repeat 7 times...
- This introduces:
  - **High disk I/O latency**
  - **Serialization/Deserialization overhead**
  - **Network shuffle between map and reduce stages**

👉 Slow due to dependence on disk for each intermediate computation.

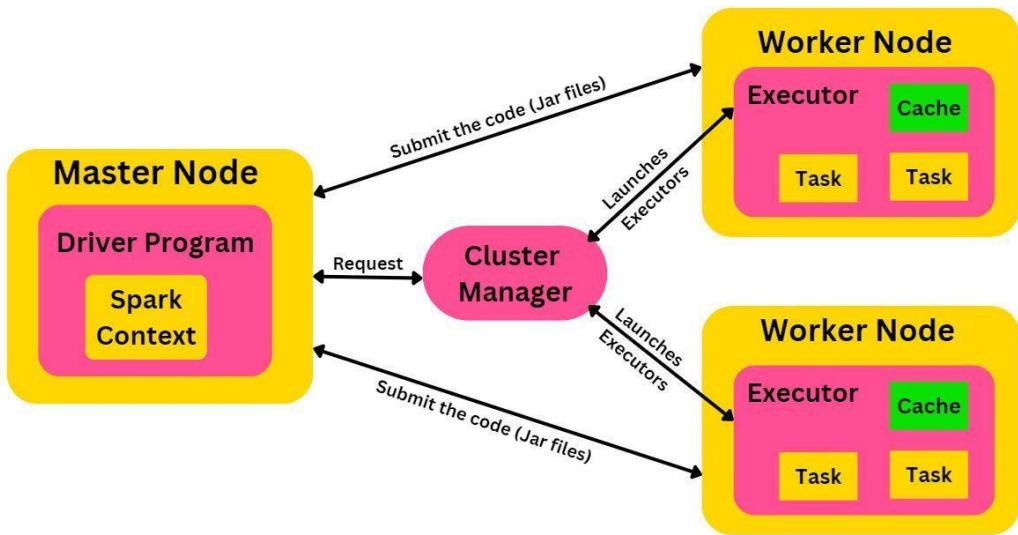
### ● In Spark:

- Spark keeps intermediate results **in memory** using **Resilient Distributed Datasets (RDDs)** or **DataFrames**.
  - All the 8 multiplications can be:
    - **Chained in a single stage** (if narrow transformation)
    - Stored in memory between wide stages
- ⚡ Faster because no disk writes until the final result. Only RAM is used.

### Why Didn't MapReduce Use In-Memory Computation Initially?

Reason	Explanation
Fault Tolerance Simplicity	MapReduce relied on disk writes between stages to recover from failures. If a node failed, HDFS still had the intermediate output.
Hardware Constraints	Early Hadoop clusters were built with commodity hardware with limited RAM, but large disk space.
Batch Processing Use Case	MapReduce was designed for batch jobs (e.g., log processing), where real-time speed wasn't the priority.
Simpler Programming Model	By isolating every step via disk, Hadoop simplified the failure boundaries but at the cost of performance.

## Spark Architecture Components



Component	Role & Function
<b>Driver Program</b>	- Entry point for your Spark application. - Contains the main method and user code. - Builds DAG (Directed Acyclic Graph). - Converts logical plan to physical execution plan. - Coordinates execution by interacting with the Cluster Manager and Executors.
<b>SparkContext</b>	- Created by the Driver. - Connects with the Cluster Manager. - Acts as the bridge between the application and Spark runtime. - Creates RDDs, accumulators, broadcast variables.
<b>Cluster Manager</b>	- Allocates resources (executors on worker nodes) to your Spark app. - Types: <b>Standalone, YARN, Mesos, or Kubernetes</b> .
<b>Executors</b>	- Run on worker nodes. - Execute tasks assigned by the driver. - Store data in memory (for caching/persisting). - Send results back to driver. - Destroyed after job completion.
<b>Task</b>	- Smallest unit of execution. - Each task processes <b>1 partition</b> of the data. - Grouped into <b>stages</b> by the DAG Scheduler.
<b>Cache / Storage</b>	- Cached data stays in memory or disk across jobs. - Reused to improve performance and avoid recomputation.

## Example Scenario

Let's walk through this code to illustrate all components in action:

```
from pyspark.sql import SparkSession

# Driver Program starts here
spark = SparkSession.builder.appName("SalesAnalysis").getOrCreate()

# SparkContext created inside SparkSession
df = spark.read.csv("hdfs:///data/sales.csv", header=True, inferSchema=True)

# Transformation (lazy)
result = df.filter("region = 'Asia'").groupBy("product").sum("revenue")

# Action (triggers DAG execution)
result.show()
```

## Execution Flow Breakdown

### 1. Job Submission

- You start the Spark application (SalesAnalysis) via a spark-submit.
- The **Driver Program** is launched on a master node.

### 2. Driver Creates SparkContext

- SparkContext connects to the chosen **Cluster Manager** (YARN/Standalone/etc.).
- Requests resources for executors.

### 3. DAG Creation

- All transformations (filter, groupBy, etc.) are **lazy**.
- Spark builds a **logical plan** → then optimizes it into a **DAG**.
- Example DAG:
- Read CSV --> Filter(region='Asia') --> GroupBy(product) --> Sum(revenue)

### 4. Divide Into Stages & Tasks

- Spark analyzes data partitions (e.g., 5 blocks → 5 partitions).
- DAG Scheduler divides DAG into **stages**:
  - Stage 1: Read + Filter (narrow transformation)
  - Stage 2: GroupBy + Aggregate (wide transformation = needs shuffle)
- Each **stage** is split into **tasks**:
  - If 5 partitions → 5 tasks per stage

## 5. Cluster Manager Allocates Executors

- Cluster Manager assigns worker nodes with **executors** for your app.
- Executors are long-running JVMs that:
  - Store data in memory
  - Run multiple tasks in parallel

## 6. Task Execution on Executors

- Tasks are **sent by Driver** to the executors.
- Each task:
  - Processes 1 partition
  - Runs filter, groupBy, and aggregation operations
  - Uses local memory for intermediate results

## 7. Intermediate Storage

- **Shuffle** occurs between wide transformations:
  - Executors write intermediate results to local disk for fault tolerance.
  - Shuffle data transferred to reducers.
- If `.cache()` or `.persist()` used → data stored in executor memory/disk

## 8. Result Collection

- Final output (`result.show()`) is collected from the executors to the Driver.
- Driver prints the tabular result to the console.

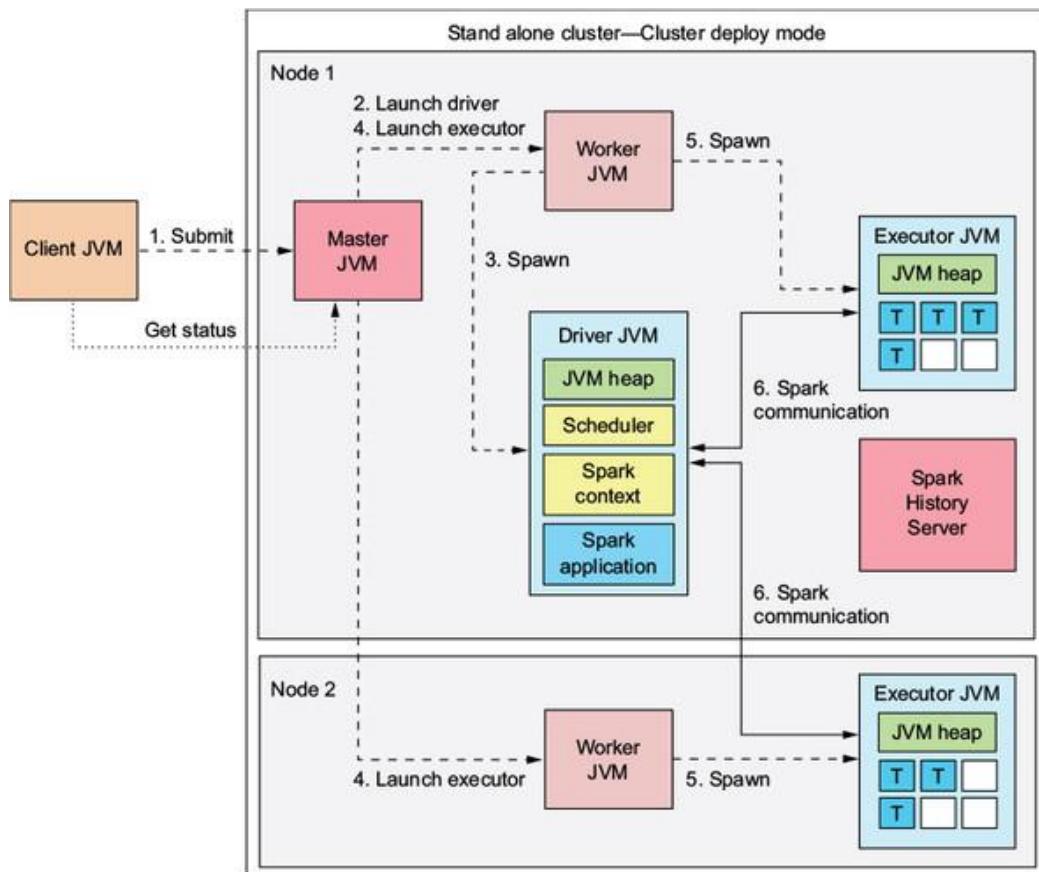
### Summary Table: Spark Component Responsibilities

Step	Component Involved	What It Does
Submit Job	Driver	Launches SparkContext, defines actions/transformations
Create SparkContext	Driver	Connects to Cluster Manager
Request Resources	Cluster Manager	Allocates Executors on Worker Nodes
Build DAG	Driver	Plans execution as DAG of stages
Divide into Stages & Tasks	DAG Scheduler (Driver)	Identifies narrow/wide ops, creates tasks
Run Tasks	Executors	Execute code, store intermediate results
Cache Data	Executors (Storage)	Holds reusable data in memory/disk
Return Result	Driver	Gathers and displays final output

## Key Concepts in Context

Concept	Meaning in Spark
<b>Job</b>	Triggered by an <b>action</b> (like <code>show()</code> , <code>collect()</code> ), consists of one or more <b>stages</b>
<b>Stage</b>	Set of transformations that can be executed together without shuffles
<b>Task</b>	Smallest unit of work; 1 per data partition
<b>Executor</b>	Executes tasks; caches/persists data if needed
<b>DAG Scheduler</b>	Builds DAG, handles stages and job flow
<b>Cluster Manager</b>	Allocates resources to applications

## Spark Standalone Architecture



**Spark Standalone** is the **built-in cluster manager** provided by Apache Spark. It allows you to run Spark in a **distributed fashion** without requiring an external cluster manager like YARN, Kubernetes, or Mesos.

Component	Description
Driver Program	Contains user code; creates SparkContext; sends tasks to executors.
Master Node	Central coordinator; manages cluster resources; launches executors on workers.
Worker Nodes	Actual machines where tasks run; executors live here and execute computations.
Executor	A JVM launched by the worker to run tasks and cache data.
Cluster Manager	Built-in in Standalone mode — the Master node itself acts as the manager.

## Spark Standalone Cluster Setup

Node Type	Roles	Notes
Master	Cluster manager	Only one active Master in HA setup
Workers	Run executors & report to Master	Can be multiple
Driver	User application	Submits job to master

## Execution Flow in Spark Standalone Mode

### 1. Application Submission

- You submit a job using:
- `./bin/spark-submit --master spark://<master-host>:7077 app.py`
- The **Driver program** starts and connects to the **Spark Master**.

### 2. Driver Registers with Master

- The **Driver (client)** registers itself with the Master.
- It requests resources to run executors.

### 3. Master Allocates Resources

- Master assigns **Worker nodes** to launch **executors** based on available cores and memory.
- Each worker spins up one or more **executors**.

### 4. Executors Start and Register with Driver

- Executors connect back to the **Driver**.
- They wait for the Driver to assign tasks.

### 5. Driver Builds DAG & Schedules Tasks

- When the first **action** (e.g., `.show()`, `.collect()`) is encountered:
  - Spark constructs a **DAG** of stages.
  - The DAG Scheduler divides it into **tasks**.

### 6. Task Execution on Executors

- Tasks are distributed to executors by the Driver.
- Executors:
  - Read data (e.g., from HDFS or local disk)
  - Process transformations
  - Write intermediate data (if shuffles are required)
  - Cache data (if specified)

- Return final output to the Driver

## 7. Driver Collects Results

- Once executors finish tasks, results are returned to the Driver.
- If .show() is used, results are displayed on the console.

## 8. Application Completion

- After the final result, the Driver signals completion.
- Executors are shut down.
- Master updates resource availability.

**Summary Table: Spark Standalone Execution Flow**

Step	Component	Activity
<b>Job Submission</b>	Driver	Submits job using spark-submit
<b>Registration with Master</b>	Driver ↔ Master	Requests resource allocation
<b>Resource Allocation</b>	Master → Workers	Assigns executors on available workers
<b>Executor Startup</b>	Workers	Executors start and register with Driver
<b>DAG Planning</b>	Driver	Transforms lazy operations into DAG and tasks
<b>Task Distribution</b>	Driver → Executors	Tasks sent to executors for processing
<b>Task Execution &amp; Caching</b>	Executors	Read, process, cache data, perform shuffles
<b>Result Return</b>	Executors → Driver	Final output sent back to Driver
<b>Shutdown</b>	Master	Executors shut down, cluster resources freed

## Why Use Standalone Mode?

☒ Pros	⚠ Cons
<b>Simple to set up</b>	Not suitable for very large clusters
<b>No dependency on external tools (YARN)</b>	No dynamic resource allocation
<b>Good for testing and small-to-mid jobs</b>	Less integration with Hadoop ecosystem

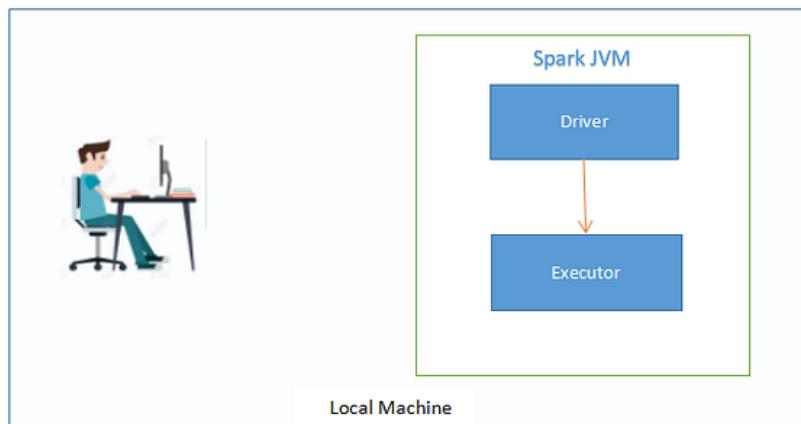
Feature	Spark Standalone	YARN (Yet Another Resource Negotiator)
<b>Cluster Manager</b>	Built-in Spark cluster manager	External, part of Hadoop ecosystem
<b>Resource Management</b>	Basic, static allocation	Advanced, dynamic allocation, fine-grained sharing of resources
<b>Scalability</b>	Moderate scalability (ideal for small to medium clusters)	Highly scalable, suitable for large production-grade clusters
<b>Driver Location</b>	Usually runs on client machine or a worker (depends on deploy mode)	Can run in client or cluster mode (cluster mode preferred for fault tolerance)
<b>Executor Management</b>	Executors launched directly by Spark Master on worker nodes	YARN NodeManagers launch and monitor executors
<b>Fault Tolerance</b>	Limited; no built-in failover for Master (unless HA is configured manually)	Strong; ResourceManager restarts applications, tracks failures
<b>Ease of Use</b>	Easy to set up; no external dependency	Requires Hadoop/YARN setup, but integrates well with existing big data stack
<b>Integration</b>	Limited (mostly with Spark components)	Deep integration with Hadoop tools (Hive, HDFS, Oozie, HBase, etc.)
<b>Application Monitoring</b>	Basic UI at <code>http://&lt;master&gt;:8080</code>	Advanced monitoring via ResourceManager UI and Spark History Server
<b>Scheduling</b>	FIFO (default), FAIR scheduling with config	Capacity Scheduler, Fair Scheduler, DRF – supports multiple queues
<b>Security</b>	Limited authentication and access control	Kerberos, ACLs, and encryption supported
<b>Cluster Sharing</b>	Less suited for multi-tenancy or sharing	Designed for multi-tenancy and resource sharing
<b>Deployment</b>	Simpler, good for dev/test setups	Better for large-scale enterprise deployments
<b>Data Locality</b>	Basic (depends on worker's locality to data)	Strong — integrates with HDFS and local nodes for optimal locality
<b>Application Recovery</b>	No recovery unless manually configured	Automatic recovery and re-execution of failed jobs/tasks
<b>Dynamic Allocation</b>	Requires additional setup	Built-in and tightly integrated
<b>Typical Use Case</b>	Dev/test, small clusters, Spark-specific workloads	Large production clusters with multiple applications and complex workflows

## Spark Deployment Modes

Apache Spark supports **multiple deployment modes** to accommodate different development and production needs. The deployment mode determines **where the driver and executors run**.

### Local Mode

Ideal for development and testing

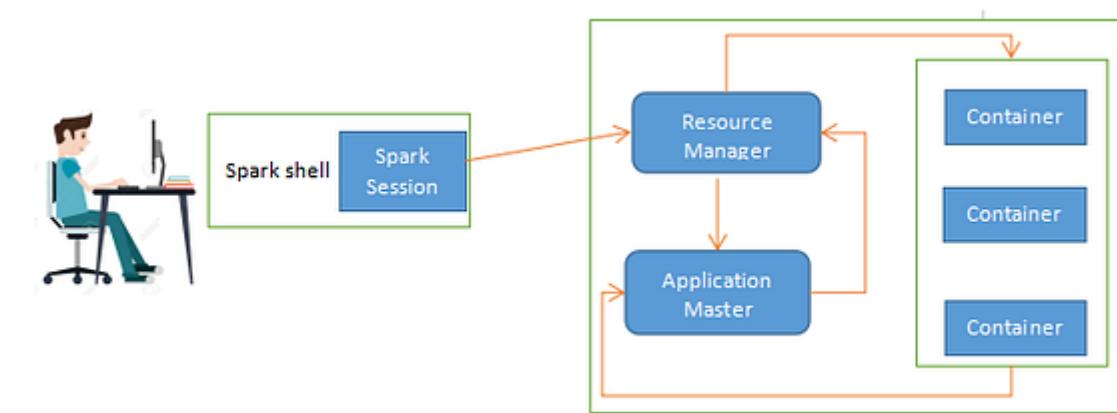


Property	Details
<b>Driver Location</b>	Runs on the <b>local machine</b>
<b>Executor Location</b>	Same as the driver (on local JVM)
<b>Cluster Manager</b>	None — runs locally without a cluster manager
<b>Usage</b>	master("local"), local[*], local[2] etc.
<b>When to Use</b>	- Prototyping- Learning/testing Spark code locally- Unit tests
<b>Limitations</b>	Not scalable, single-machine execution

Example:

```
SparkSession.builder.master("local[*]").appName("LocalTest").getOrCreate()
```

## Client Mode



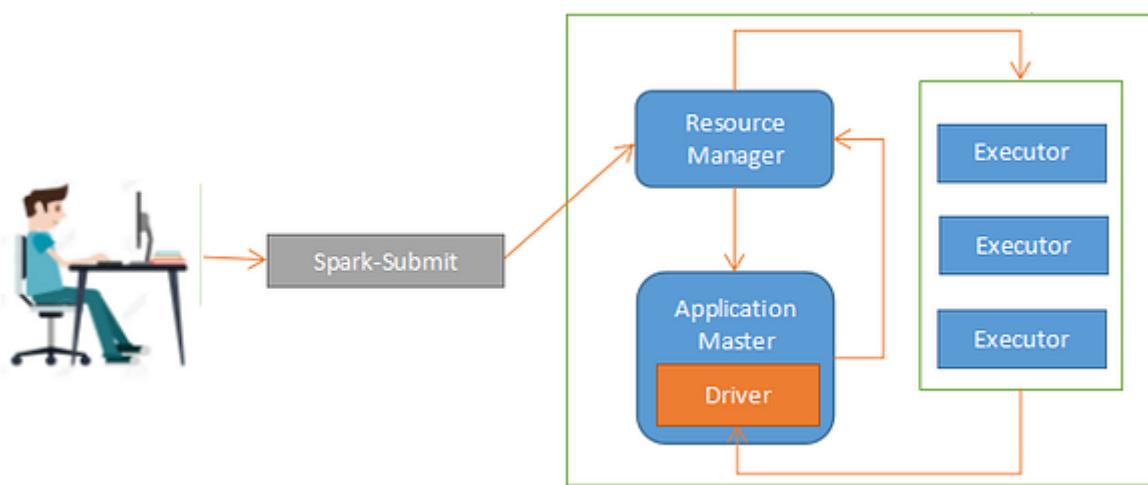
**Driver runs on submitting machine (edge node/laptop)**

Property	Details
<b>Driver Location</b>	On your local machine or edge node (i.e., where job is submitted)
<b>Executor Location</b>	On cluster nodes (e.g., YARN, Kubernetes, etc.)
<b>Cluster Manager</b>	Spark Standalone, YARN, Kubernetes, Mesos
<b>When to Use</b>	- When the client (your machine) has a stable connection to the cluster
<b>Risk</b>	- If client disconnects/crashes, the job fails
<b>Benefits</b>	Easy debugging (driver logs stay local)

**Typical Command:**

```
spark-submit --master yarn --deploy-mode client my_script.py
```

## Cluster Mode



**Production-grade, highly fault-tolerant**

Property	Details
<b>Driver Location</b>	On a worker node inside the cluster
<b>Executor Location</b>	On cluster nodes
<b>Cluster Manager</b>	Spark Standalone, YARN, Kubernetes, Mesos
<b>When to Use</b>	- Production deployments- Cloud/data center usage- Better fault tolerance
<b>Benefits</b>	- Client can disconnect after submission- Resilient to client failure
<b>Risk</b>	Slightly harder debugging (logs on cluster nodes)

**Typical Command:**

```
spark-submit --master yarn --deploy-mode cluster my_script.py
```

Aspect	Local Mode	Client Mode	Cluster Mode
<b>Driver Location</b>	Local machine	Submitting machine (laptop/edge node)	Worker node in cluster
<b>Executor Location</b>	Local machine	Cluster nodes	Cluster nodes
<b>Cluster Manager</b>	None	YARN / Standalone / Kubernetes / Mesos	YARN / Standalone / Kubernetes / Mesos
<b>Network Requirement</b>	None	<b>Stable connection</b> to cluster from client	Only for job submission; then independent
<b>Communication Latency</b>	None (all in local JVM)	Moderate — between client and cluster	Low — all inside cluster
<b>Resource Management</b>	Local machine only (manual/limited)	Managed by cluster manager	Fully managed by cluster manager
<b>Fault Tolerance</b>	Low — if machine crashes, job dies	Low — if client disconnects, job dies	High — driver & executors managed by cluster
<b>Cloud Examples</b>	Your laptop only	Submitting from laptop to EMR/YARN	Running inside GCP Dataproc / AWS EMR / AKS

# Introduction to Databricks

## What is Databricks?

**Databricks** is a **cloud-based unified data analytics platform** built on top of **Apache Spark**. It helps in processing large-scale data, developing machine learning models, and building scalable data pipelines, all in one collaborative environment.

Think of it as a managed environment that combines **big data + AI + collaborative notebooks**.

## Key Features

Feature	Description
<b>Unified Workspace</b>	Combines data engineering, data science, and machine learning workflows.
<b>Collaborative Notebooks</b>	Supports Python, SQL, Scala, and R with real-time collaboration.
<b>Auto Scaling and Clusters</b>	Auto-provisions and scales Spark clusters for workloads.
<b>Job Scheduling</b>	Schedule and monitor jobs with built-in scheduler.
<b>Delta Lake Support</b>	Supports ACID transactions on data lakes with Delta Lake.
<b>MLflow Integration</b>	Built-in tools for MLOps and model tracking.
<b>Visualization</b>	Built-in charts and dashboards in notebooks.

## Supported Cloud Platforms

- **AWS**
- **Microsoft Azure (Azure Databricks)**
- **Google Cloud Platform (GCP)**

## Databricks Architecture

1. **Control Plane** (managed by Databricks):
  - Manages the workspace, UI, job scheduling, authentication, etc.
2. **Compute Plane** (in your cloud account):
  - Spark clusters and data processing happens here.
  - Your data **stays within your VPC**.

## Core Components of Databricks

### 1. Workspace

- Main user interface where users can:
  - Create notebooks
  - Manage libraries and files
  - Browse data
  - Create jobs and clusters

### 2. Notebooks

- Interactive documents for code, results, visualizations.
- Languages: **Python, SQL, Scala, R** (multi-language supported with %magic commands like %sql, %python).

### 3. Clusters

- Set of compute resources for running notebooks/jobs.
- Types:
  - **Interactive Clusters:** For development
  - **Job Clusters:** Created/destroyed per job run

### 4. Jobs

- Scheduled or manual runs of notebooks, scripts, or workflows.

### 5. DBFS (Databricks File System)

- A distributed file system (abstraction over cloud storage).
- Accessible via /dbfs/

### 6. Delta Lake

- Open-source storage layer that brings ACID transactions to data lakes.
- Supports:
  - Time travel
  - Schema enforcement and evolution
  - Scalable metadata

## Common Use Cases

Use Case	Description
<b>ETL Pipelines</b>	Extract, transform, and load large-scale data
<b>Batch &amp; Streaming Analytics</b>	Real-time insights using Spark Structured Streaming
<b>Machine Learning</b>	Build, train, and deploy ML models using MLflow
<b>Data Lakehouse</b>	Combines data lake storage + data warehouse capabilities
<b>Business Intelligence (BI)</b>	Use with tools like Power BI, Tableau, or native dashboards

## Data Access

Databricks supports data from:

- **Cloud storage** (e.g., AWS S3, Azure Blob, GCS)
- **JDBC-compatible databases** (MySQL, PostgreSQL, etc.)
- **Delta tables**
- **Parquet, CSV, JSON, Avro files**

## Developer Tips

- Use **display()** for interactive visualizations in notebooks.
- Use **Databricks Utilities (dbutils)** for file management, widgets, secrets, etc.
- Use **%run** to modularize notebooks.
- Save jobs as **workflows** for production.

## Security and Governance

- Role-based access control (RBAC)
- Unity Catalog (for fine-grained data governance)
- Audit logs and cluster-level policies

## Tools and Integrations

- **MLflow** – model tracking and deployment
- **Delta Live Tables (DLT)** – declarative ETL pipelines
- **Unity Catalog** – centralized data catalog with governance
- **Repos** – native Git integration for version control
- **REST APIs** – for automation and CI/CD

## Databricks vs Alternatives

Feature	Databricks	EMR	Snowflake	BigQuery
Spark-native	✓	✓	✗	✗
Notebook UI	✓	✗	✗	✗
Delta Lake	✓	Partial	✗	✗
ML Integration	✓	✗	✗	✗

## Control Plane vs Data Plane

Databricks architecture separates the platform into:

### 1. Control Plane

Managed by **Databricks** (in their cloud environment).

Responsible for **managing** and **orchestrating** workloads — not for executing your data processing.

Element	Description
UI	Web-based interface to create notebooks, manage clusters, jobs, etc.
Workspace Management	User folders, notebooks, repos, file structure organization
Job Scheduler	Built-in job orchestration and scheduling engine
Cluster Manager	Interface to configure and manage clusters (creation, scaling, termination)
REST APIs	Control plane exposes APIs to control jobs, clusters, notebooks
Access Control	RBAC, credential passthrough, Unity Catalog metadata management

 **Note:** No customer data is stored in the control plane.

### 2. Data Plane

Runs **inside your cloud account** (AWS, Azure, or GCP) — where actual **computation and data access** happen.

Element	Description
Driver Node	The main node that coordinates Spark job execution (runs Spark driver program)
Executor Nodes	Worker nodes that process data, execute tasks (e.g., in parallel)
Compute Resources	Virtual machines/instances in your cloud (EC2, Azure VM, etc.)
Writer Nodes	Spark executors that write the output data
DBFS (Databricks File System)	Interface over cloud object storage (e.g., S3, ADLS) used to read/write data
Code Execution	All transformations/actions on data are executed here
Networking	Data plane connects to your VPC resources (databases, storage, APIs)

 **Data privacy:** The data plane is **fully isolated** and hosted in your own cloud account (unless you're using Databricks' serverless compute).

## Databricks File System (DBFS)

### What is DBFS?

**DBFS** is an **abstraction layer** over your cloud object storage (like AWS S3, Azure Blob, or GCP GCS), making it **look like a file system** inside Databricks. Distributed file system.

You can access files in /dbfs/ like a local file system, even though they're backed by cloud storage.

### DBFS Key Features:

Feature	Description
<b>Mountable</b>	You can mount cloud storage locations (S3, ADLS) to DBFS paths.
<b>Accessible from Notebooks</b>	Access files using dbutils.fs or /dbfs/path
<b>Versioned</b>	Managed files can be versioned (if using Delta Lake)
<b>Used for</b>	Uploading datasets, saving models, checkpointing, caching, etc.

### DBFS Zones

Zone	Purpose
/databricks-datasets/	Public datasets provided by Databricks
/tmp/	Temporary storage (node-local, not persistent)
/dbfs/	Main access point to files in DBFS
/mnt/	Mounted cloud storage locations (e.g., S3, ADLS)

# Additional Note I

## Example DataFrame Creation

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, min, max, rank, dense_rank, row_number, length
from pyspark.sql import Window

spark = SparkSession.builder.appName("BeginnerSparkDF").getOrCreate()
data = [
    (1, "Alice", "New York", "NY", "2023-05-01"),
    (2, "Bob", "Los Angeles", "CA", "2023-05-03"),
    (3, "Charlie", "New York", "NY", "2023-05-02"),
    (4, "David", "Los Angeles", "CA", "2023-05-04"),
    (5, "Eva", "Chicago", "IL", "2023-05-05"),
    (6, "Frank", "Chicago", "IL", "2023-05-01"),
]

columns = ["id", "name", "city", "state", "registration_date"]

df = spark.createDataFrame(data, columns)
df.show()
```

**Output:**

```
+---+-----+-----+-----+
| id|     name|       city|state|registration_date|
+---+-----+-----+-----+
|  1|   Alice| New York|  NY|  2023-05-01|
|  2|     Bob| Los Angeles | CA| 2023-05-03|
|  3|Charlie| New York|  NY| 2023-05-02|
|  4|  David| Los Angeles | CA| 2023-05-04|
|  5|    Eva|     Chicago| IL| 2023-05-05|
|  6|  Frank|     Chicago| IL| 2023-05-01|
+---+-----+-----+-----+
```

## Using .withColumn()

```
df1 = df.withColumn("name_length", length(col("name")))
df1.show()
```

**Output:**

```
+---+-----+-----+-----+-----+
| id|     name|       city|state|registration_date|name_length|
+---+-----+-----+-----+-----+
|  1|   Alice| New York|  NY|  2023-05-01|      7|
|  2|     Bob| Los Angeles | CA| 2023-05-03|     10|
|  3|Charlie| New York|  NY| 2023-05-02|      7|
|  4|  David| Los Angeles | CA| 2023-05-04|     10|
|  5|    Eva|     Chicago| IL| 2023-05-05|      7|
|  6|  Frank|     Chicago| IL| 2023-05-01|      7|
+---+-----+-----+-----+-----+
```

`.withColumn()` either **creates a new column or replaces an existing one**.

Here, we're adding "name\_length" to store the number of letters in each customer's name.

We use:

- `length(col("name"))` → counts characters in the name column.
- `col()` → tells Spark which column we're referring to.

## Getting Unique Values

```
unique_cities = df.select("city").distinct().collect()  
print(unique_cities)      # Full list of unique cities  
print(unique_cities[0][0]) # First unique city
```

**Output:**

```
[Row(city='Chicago'), Row(city='Los Angeles'), Row(city='New York')]  
Chicago
```

- `.select("city")` → picks only the city column.
- `.distinct()` → keeps only **unique values**, removing duplicates.
- `.collect()` → brings the results from Spark into Python as a list.
- `[0][0]` → gets the **first city name** from that list.

## Pivoting Data

```
pivot_df = df.groupBy("state").pivot("city").count()  
pivot_df.show()
```

**Output**

```
+-----+-----+-----+  
|state|Chicago|Los Angeles|New York|  
+-----+-----+-----+  
|   IL|      2|        null|      null|  
|   NY|    null|        null|       2|  
|   CA|    null|         2|      null|  
+-----+-----+-----+
```

Pivoting turns **unique values in a column into new columns**.

Here:

- `groupBy("state")` → groups the table by state.
- `.pivot("city")` → turns each city name into a column.
- `.count()` → counts how many rows fall into each state-city combination.  
null means there was no data for that combination.

## Using Window Functions

```
window_spec = Window.partitionBy('state').orderBy(col('registration_date').desc())\n\ndf1 = df1.withColumn('rank', rank().over(window_spec)) \\ \n    .withColumn('dense_rank', dense_rank().over(window_spec)) \\ \n    .withColumn('row_number', row_number().over(window_spec))\n\ndf1.show()
```

### Output

+-----+-----+-----+-----+-----+-----+-----+-----+	id   name   city   state   registration_date   name_length   rank   dense_rank   row_number	+-----+-----+-----+-----+-----+-----+-----+-----+
5  Eva   Chicago   IL   2023-05-05   3   1   1   1		
6  Frank   Chicago   IL   2023-05-01   5   2   2   2		
4  David   Los Angeles   CA   2023-05-04   5   1   1   1		
2  Bob   Los Angeles   CA   2023-05-03   3   2   2   2		
3  Charlie   New York   NY   2023-05-02   7   1   1   1		
1  Alice   New York   NY   2023-05-01   5   2   2   2		

A **Window** lets you calculate values **within groups of rows** without collapsing them into one row.

- partitionBy('state') → split data by state.
- orderBy(registration\_date desc) → order newest first inside each state.

Functions:

- rank()** → assigns ranking numbers (ties skip numbers: 1, 2, 2, 4).
- dense\_rank()** → ties still get same rank, but no number gaps (1, 2, 2, 3).
- row\_number()** → gives each row a unique position, even if values are the same.

## Aggregation — Oldest & Newest Customer per City

```
df1.groupBy('city') \\ \n    .agg(\n        min('registration_date').alias('oldest'),\n        max('registration_date').alias('newest'))\n    ) \\ \n    .show()
```

### Output:

+-----+-----+-----+-----+	city   oldest   newest	+-----+-----+-----+
Los Angeles   2023-05-03   2023-05-04		
Chicago   2023-05-01   2023-05-05		
New York   2023-05-01   2023-05-02		

- `.groupBy('city')` → groups rows for each city.
- `.agg()` → lets us run multiple aggregation functions in one go.
- `min('registration_date')` → finds the earliest date in each city.
- `max('registration_date')` → finds the latest date in each city.

### Summary Table

Function	Purpose	Example
<code>withColumn</code>	Add or update a column	<code>df.withColumn("new", col("old")*2)</code>
<code>distinct</code>	Remove duplicates	<code>df.select("city").distinct()</code>
<code>pivot</code>	Convert unique values into columns	<code>groupBy().pivot().count()</code>
<code>Window</code>	Perform ranking or calculations within groups	<code>rank().over(window_spec)</code>
<code>agg()</code>	Aggregate data with multiple functions	min, max, avg, etc.

# Spark Project 1

```
# Create the session
from pyspark.sql import SparkSession
from pyspark.sql.functions import *

spark = SparkSession.builder.appName("CustomerDataProcessing").getOrCreate()
spark
```

## SparkSession - hive

### SparkContext

#### [Spark UI](#)

#### Version

v3.5.3

#### Master

yarn

#### AppName

PySparkShell

```
df1 = spark.read.format('csv').option("header", 'true').load("/tmp/customers_1mb.csv")
```

```
df1.show(5)
[Stage 1:>                                         (0 + 1) / 1]
+-----+-----+-----+-----+-----+
|customer_id|      name|     city|      state|country|registration_date|is_active|
+-----+-----+-----+-----+-----+
|      0|Customer_0|      Pune|Maharashtra|  India|  2023-06-29| False|
|      1|Customer_1|Bangalore| Tamil Nadu|  India| 2023-12-07| True|
|      2|Customer_2|Hyderabad|   Gujarat|  India| 2023-10-27| True|
|      3|Customer_3|Bangalore| Karnataka|  India| 2023-10-17| False|
|      4|Customer_4|Ahmedabad| Karnataka|  India| 2023-03-14| False|
+-----+-----+-----+-----+-----+
only showing top 5 rows
```

```
df1.printSchema()
```

```
root
|-- customer_id: string (nullable = true)
|-- name: string (nullable = true)
|-- city: string (nullable = true)
|-- state: string (nullable = true)
|-- country: string (nullable = true)
|-- registration_date: string (nullable = true)
|-- is_active: string (nullable = true)
```

```
df1 = df1.withColumn('registration_date',to_date(col('registration_date'),'yyyy-MM-dd'))\
    .withColumn('is_active', col('is_active').cast('boolean'))
```

```
df1.show(5)
```

```
+-----+-----+-----+-----+-----+
|customer_id|      name|     city|      state|country|registration_date|is_active|
+-----+-----+-----+-----+-----+
|      0|Customer_0|      Pune|Maharashtra|  India|  2023-06-29| false|
|      1|Customer_1|Bangalore| Tamil Nadu|  India| 2023-12-07| true|
|      2|Customer_2|Hyderabad|   Gujarat|  India| 2023-10-27| true|
|      3|Customer_3|Bangalore| Karnataka|  India| 2023-10-17| false|
|      4|Customer_4|Ahmedabad| Karnataka|  India| 2023-03-14| false|
+-----+-----+-----+-----+-----+
only showing top 5 rows
```

```

df1.printSchema()
root
|-- customer_id: string (nullable = true)
|-- name: string (nullable = true)
|-- city: string (nullable = true)
|-- state: string (nullable = true)
|-- country: string (nullable = true)
|-- registration_date: date (nullable = true)
|-- is_active: boolean (nullable = true)
# Fill the empty values
df1 = df1.fillna({'city':'Unknown','state':'Unknown','country':'Unknown'})

```

```

# Extract registration date and month
df1 = df1.withColumn('registration_year',year(col('registration_date')))\n        .withColumn('registration_month',month(col('registration_date')))
df1.show(5)
+-----+-----+-----+-----+-----+-----+-----+-----+
|customer_id|    name|     city|   state|country|registration_date|is_active|registration_year|registration_month|
+-----+-----+-----+-----+-----+-----+-----+-----+
|      0|Customer_0|      Pune|Maharashtra|   India| 2023-06-29|    false|       2023|             6|
|      1|Customer_1|Bangalore|Tamil Nadu|   India| 2023-12-07|     true|       2023|            12|
|      2|Customer_2|Hyderabad|   Gujarat|   India| 2023-10-27|     true|       2023|             10|
|      3|Customer_3|Bangalore|Karnataka|   India| 2023-10-17|    false|       2023|             10|
|      4|Customer_4|Ahmedabad|Karnataka|   India| 2023-03-14|    false|       2023|              3|
+-----+-----+-----+-----+-----+-----+-----+-----+
only showing top 5 rows

```

```

# Get the unique values
unique_cities = df1.select(countDistinct('city')).collect()
print(unique_cities[0][0])

unique_states = df1.select(countDistinct('state')).collect()
print(unique_states[0][0])

unique_countries = df1.select(countDistinct('country')).collect()
print(unique_countries[0][0])
8
7
1
df1.groupBy('city').count().orderBy(col('count').desc()).show()

```

```

df1.groupBy('city').count().orderBy(col('count').desc()).show()
df1.groupBy('city').count().orderBy(col('count').desc()).show()
+-----+-----+
|     city|count|
+-----+-----+
|      Pune| 2243|
|Hyderabad| 2242|
|   Kolkata| 2223|
|Bangalore| 2211|
|      Delhi| 2200|
|Ahmedabad| 2198|
|   Chennai| 2194|
|     Mumbai| 2142|
+-----+-----+

```

```
df1.groupBy('state','country').count().orderBy(col('count').desc()).show()
```

state	country	count
Delhi	India	2578
Gujarat	India	2543
Tamil Nadu	India	2536
Telangana	India	2520
West Bengal	India	2503
Maharashtra	India	2490
Karnataka	India	2483

```
# Pivot table -> get a count of active and inactive users per state
```

```
df1.groupBy('state').pivot('is_active').count().show()
```

state	false	true
Gujarat	1211	1332
Delhi	1356	1222
Karnataka	1207	1276
Telangana	1294	1226
Maharashtra	1260	1230
Tamil Nadu	1284	1252
West Bengal	1306	1197

```
from pyspark.sql import Window
```

```
from pyspark.sql.functions import col, rank, dense_rank, row_number
```

```
window_spec = Window.partitionBy('state').orderBy(col('registration_date').desc())
```

```
df1 = df1.withColumn('rank',rank().over(window_spec))\n    .withColumn('dense_rank',dense_rank().over(window_spec))\n    .withColumn('row_number',row_number().over(window_spec))
```

```
df1.show(5)
```

customer_id	name	city	state	country	registration_date	is_active	registration_year	registration_month	rank	dense_rank	row_number
61	Customer_61	Hyderabad	Delhi	India	2023-12-31	false	2023		12	1	1
501	Customer_501	Mumbai	Delhi	India	2023-12-31	false	2023		12	1	1
2763	Customer_2763	Pune	Delhi	India	2023-12-31	true	2023		12	1	1
12858	Customer_12858	Ahmedabad	Delhi	India	2023-12-31	true	2023		12	1	1
13570	Customer_13570	Bangalore	Delhi	India	2023-12-31	false	2023		12	1	1

only showing top 5 rows

```
# Get the recent customers
```

```
df_recent_customers = df1.filter(col('registration_date') >= lit('2023-07-01'))
```

```
df_recent_customers.show(10)
```

customer_id	name	city	state	country	registration_date	is_active	registration_year	registration_month	rank	dense_rank	row_number
61	Customer_61	Hyderabad	Delhi	India	2023-12-31	false	2023		12	1	1
501	Customer_501	Mumbai	Delhi	India	2023-12-31	false	2023		12	1	1
2763	Customer_2763	Pune	Delhi	India	2023-12-31	true	2023		12	1	1
12858	Customer_12858	Ahmedabad	Delhi	India	2023-12-31	true	2023		12	1	1
13570	Customer_13570	Bangalore	Delhi	India	2023-12-31	false	2023		12	1	1
14970	Customer_14970	Chennai	Delhi	India	2023-12-31	true	2023		12	1	1
16447	Customer_16447	Ahmedabad	Delhi	India	2023-12-31	false	2023		12	1	1
16709	Customer_16709	Chennai	Delhi	India	2023-12-31	false	2023		12	1	1
17129	Customer_17129	Chennai	Delhi	India	2023-12-31	true	2023		12	1	1
250	Customer_250	Pune	Delhi	India	2023-12-30	true	2023		12	10	2

only showing top 10 rows

```
df_recent_customers.count()
```

9025

```

# Get the oldest and newest customer per city
df1.groupBy('city').agg(min('registration_date').alias('oldest'),max('registration_date').alias('newest')).show()
+-----+-----+
|   city|   oldest|   newest|
+-----+-----+
|   Delhi|2023-01-01|2023-12-31|
|Kolkata|2023-01-01|2023-12-31|
|Hyderabad|2023-01-01|2023-12-31|
|Bangalore|2023-01-01|2023-12-31|
|Ahmedabad|2023-01-01|2023-12-31|
|Chennai|2023-01-01|2023-12-31|
|Mumbai|2023-01-01|2023-12-31|
|Pune|2023-01-01|2023-12-31|
+-----+-----+
output_path = "/tmp/processed_customers_1mb.csv"
df1.write.mode('overwrite').parquet(output_path)
! hadoop fs -ls -h /tmp/
Found 8 items
-rw-r--r--  2 mahelapandukabandara hadoop      10.0 M 2025-08-01 07:29 /tmp/customers_10mb.csv
-rw-r--r--  2 mahelapandukabandara hadoop    160.7 M 2025-08-01 07:30 /tmp/customers_150mb.csv
-rw-r--r--  2 mahelapandukabandara hadoop      1.0 M 2025-08-01 07:26 /tmp/customers_1mb.csv
-rw-r--r--  2 mahelapandukabandara hadoop   327.4 M 2025-08-01 08:48 /tmp/customers_350mb.csv
drwxrwxrwt - hdfs          hadoop      0 2025-06-30 09:52 /tmp/hadoop-yarn
drwx-wx-wx - hive           hadoop      0 2025-06-30 09:53 /tmp/hive
drwxr-xr-x - root          hadoop      0 2025-08-11 17:06 /tmp/processed_customers_1mb
drwxr-xr-x - root          hadoop      0 2025-08-11 17:15 /tmp/processed_customers_1mb.csv

orders_df =
spark.read.format('csv').option("header", 'true').option('inferSchema',True).load("/tmp/orders.csv")
orders_df.show(5)
+-----+-----+-----+-----+
|order_id|customer_id|order_date|total_amount|status|
+-----+-----+-----+-----+
|      0|     3692|2024-09-03|547.7160076008001| Shipped|
|      1|    11055|2024-08-10|577.8942599188381| Pending|
|      2|     6963|2024-08-22|484.2085562764487| Pending|
|      3|    13268|2024-09-01|366.3286882431848|Cancelled|
|      4|     1131|2024-08-09|896.9588380686909| Pending|
+-----+-----+-----+-----+
only showing top 5 rows
orders_df.printSchema()
root
 |-- order_id: string (nullable = true)
 |-- customer_id: string (nullable = true)
 |-- order_date: string (nullable = true)
 |-- total_amount: string (nullable = true)
 |-- status: string (nullable = true)

# Analysis
customer_orders_df = df1.join(orders_df, "customer_id", "inner")
customer_orders_df.show(5)
+-----+-----+-----+-----+-----+-----+-----+-----+
|customer_id|   name|   city|   state|country|registration_date|is_active|order_id|order_date|total_amount|status|
+-----+-----+-----+-----+-----+-----+-----+-----+
|      2|Customer_2|Hyderabad|Gujarat|India|2023-10-27| True| 10691|2024-06-16|215.94174086119202| Pending|
|      2|Customer_2|Hyderabad|Gujarat|India|2023-10-27| True| 2859|2024-10-12|345.01571719332037|Cancelled|
|      3|Customer_3|Bangalore|Karnataka|India|2023-10-17| False| 8728|2024-07-19| 939.6745764247852|Cancelled|
|      4|Customer_4|Ahmedabad|Karnataka|India|2023-03-14| False| 17286|2024-03-23|19.279629205930526|Cancelled|
|      4|Customer_4|Ahmedabad|Karnataka|India|2023-03-14| False| 2228|2024-11-23| 512.2135003327533|Delivered|
+-----+-----+-----+-----+-----+-----+-----+-----+
only showing top 5 rows
# Total order per customer
customers_orders_count = customer_orders_df.groupBy("customer_id").count()
customers_orders_count.show(5)

```

```

+-----+-----+
|customer_id|count|
+-----+-----+
|      296|    2|
|    13192|    3|
|   16250|    2|
|    9583|    2|
|   13610|    1|
+-----+-----+
only showing top 5 rows
customers_total_spend =
customer_orders_df.groupBy("customer_id").agg(sum('total_amount')).orderBy(col('sum(total_amount)').desc())
customers_total_spend.show(10)
+-----+-----+
|customer_id| sum(total_amount)|
+-----+-----+
|     3336| 4362.550733141537|
|     3884| 4187.99763145619|
|    16020| 3967.2692112582276|
|    14372| 3961.787139557334|
|   14933| 3828.5841072418348|
|     7566| 3647.119115720654|
|   10559| 3548.8378633460234|
|   11776| 3438.36692751212|
|   11449| 3396.060974816134|
|     5425| 3389.162933156913|
+-----+-----+
only showing top 10 rows
customers_avg_spend =
customer_orders_df.groupBy("customer_id").agg(avg('total_amount')).orderBy(col('avg(total_amount)').desc())
customers_avg_spend.show(10)
+-----+-----+
|customer_id|avg(total_amount)|
+-----+-----+
|    11854| 999.864258397557|
|      46| 999.592553819927|
|   17590| 999.5726342625253|
|   11587| 999.5595016039513|
|    6816| 999.4348902885968|
|    9648| 999.421469440536|
|   15486| 999.0348855189945|
|   10980| 998.9071094160754|
|    1711| 998.7736900580057|
|    6333| 998.6394502434505|
+-----+-----+
only showing top 10 rows
# Order by status
order_status_count = customer_orders_df.groupBy("status").count()
order_status_count.show(5)
+-----+-----+
|  status|count|
+-----+-----+
|Cancelled| 4469|
|Delivered| 4341|
|  Shipped| 4386|
| Pending| 4457|
+-----+-----+

```

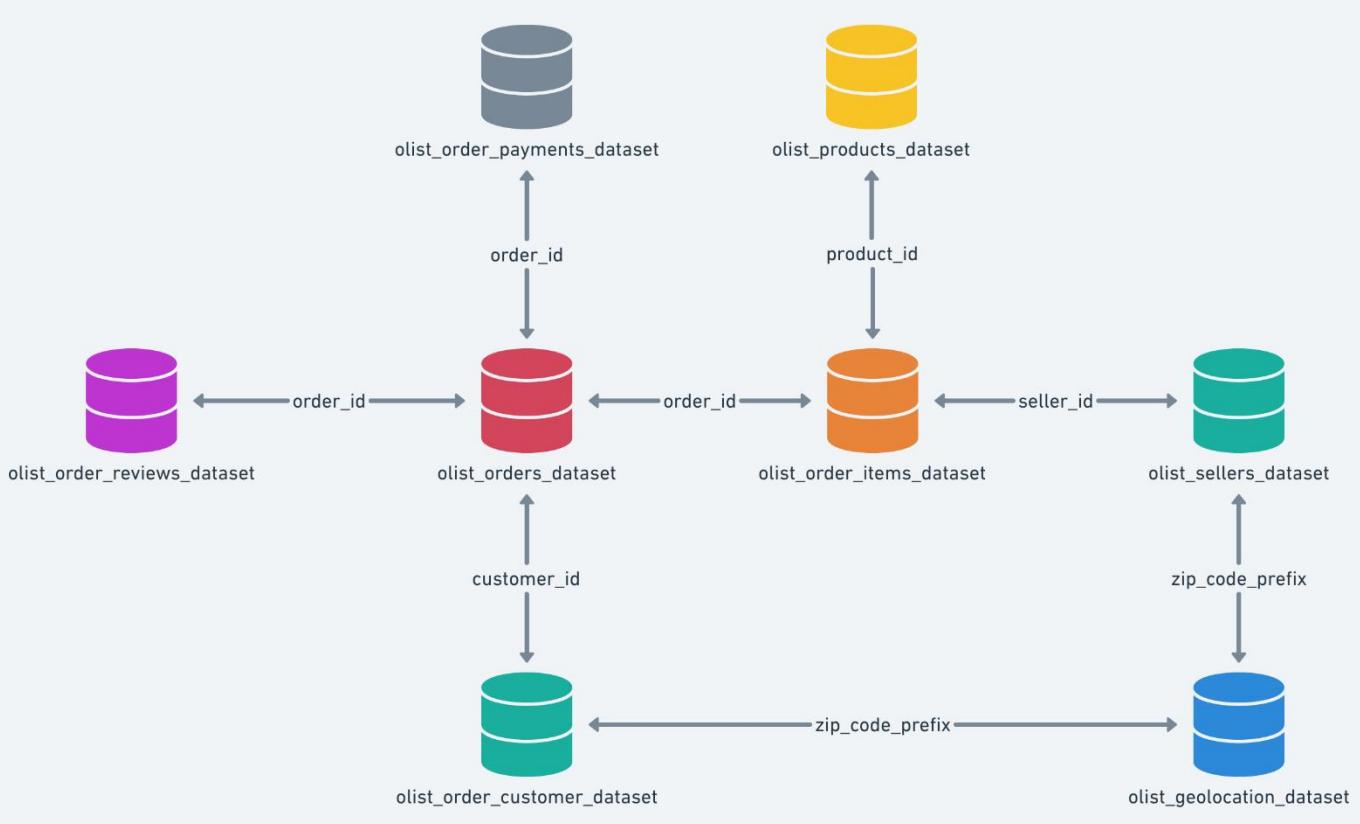
```

# Order by month
orders_by_month =
customer_orders_df.withColumn('order_month',month(col('order_date'))).groupBy('order_month')
  .count().orderBy('order_month')
orders_by_month.show()
+-----+-----+
|order_month|count|
+-----+-----+
|      1| 1499|
|      2| 1368|
|      3| 1539|
|      4| 1457|
|      5| 1518|
|      6| 1455|
|      7| 1517|
|      8| 1472|
|      9| 1446|
|     10| 1513|
|     11| 1426|
|     12| 1443|
+-----+-----+
window_spec = Window.orderBy(col('sum(total_amount)').desc())

ranked_customers =
customers_total_spend.withColumn('dense_rank',dense_rank().over(window_spec))
ranked_customers.show(10)
+-----+-----+-----+
|customer_id| sum(total_amount)|dense_rank|
+-----+-----+-----+
|    3336| 4362.550733141537|        1|
|    3884| 4187.99763145619|        2|
| 16020| 3967.2692112582276|        3|
| 14372| 3961.787139557334|        4|
| 14933| 3828.5841072418348|        5|
|   7566| 3647.119115720654|        6|
| 10559| 3548.8378633460234|        7|
| 11776| 3438.36692751212|        8|
| 11449| 3396.060974816134|        9|
|   5425| 3389.162933156913|       10|
+-----+-----+-----+
only showing top 10 rows
# Finding customers with hifgh frequency and low spent
customers_spend_vs_orders =
customers_orders_count.join(customers_total_spend,'customer_id','inner')\
  .orderBy(col('count').desc(),col('sum(total_amount)'))
customers_spend_vs_orders.show(10)
+-----+-----+-----+
|customer_id|count| sum(total_amount)|
+-----+-----+-----+
|   11776|    7| 3438.36692751212|
|    5160|    6| 1656.737343311546|
|    4294|    6| 1821.603928366352|
|    3243|    6| 2860.1827303387754|
| 14838|    6| 2894.355602564058|
| 13034|    6| 3195.0224622202268|
|   7566|    6| 3647.119115720654|
|    3884|    6| 4187.99763145619|
|    3336|    6| 4362.550733141537|
|    5002|    5| 1017.6476860103272|
+-----+-----+-----+
only showing top 10 rows

```

# Spark Project 2 : Brazilian E-Commerce Dataset



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

**SparkSession - hive**

**SparkContext**

[Spark UI](#)

**Version**

v3.5.3

**Master**

yarn

**AppName**

PySparkShell

```
!hadoop fs -ls /data/olist/
Found 9 items
-rw-r--r-- 2 mahelapandukabandara hadoop 9033957 2025-08-27 19:45 /data/olist/olist_customers_dataset.csv
-rw-r--r-- 2 mahelapandukabandara hadoop 61273883 2025-08-27 19:45 /data/olist/olist_geolocation_dataset.csv
-rw-r--r-- 2 mahelapandukabandara hadoop 15438671 2025-08-27 19:45 /data/olist/olist_order_items_dataset.csv
-rw-r--r-- 2 mahelapandukabandara hadoop 5777138 2025-08-27 19:45 /data/olist/olist_order_payments_dataset.csv
-rw-r--r-- 2 mahelapandukabandara hadoop 14451670 2025-08-27 19:45 /data/olist/olist_order_reviews_dataset.csv
-rw-r--r-- 2 mahelapandukabandara hadoop 17654914 2025-08-27 19:45 /data/olist/olist_orders_dataset.csv
-rw-r--r-- 2 mahelapandukabandara hadoop 2379446 2025-08-27 19:45 /data/olist/olist_products_dataset.csv
-rw-r--r-- 2 mahelapandukabandara hadoop 174703 2025-08-27 19:45 /data/olist/olist_sellers_dataset.csv
-rw-r--r-- 2 mahelapandukabandara hadoop 2613 2025-08-27 19:45 /data/olist/product_category_name_translation.csv
```

```
customer_df = spark.read.csv(hdfs_path + 'olist_customers_dataset.csv')
customer_df.show(5)
```

```
customers_df = spark.read.csv("/data/olist/olist_customers_dataset.csv", header=True, inferSchema=True)
geolocation_df = spark.read.csv("/data/olist/olist_geolocation_dataset.csv", header=True, inferSchema=True)
order_items_df = spark.read.csv("/data/olist/olist_order_items_dataset.csv", header=True, inferSchema=True)
order_payments_df = spark.read.csv("/data/olist/olist_order_payments_dataset.csv", header=True, inferSchema=True)
order_reviews_df = spark.read.csv("/data/olist/olist_order_reviews_dataset.csv", header=True, inferSchema=True)
orders_df = spark.read.csv("/data/olist/olist_orders_dataset.csv", header=True, inferSchema=True)
products_df = spark.read.csv("/data/olist/olist_products_dataset.csv", header=True, inferSchema=True)
sellers_df = spark.read.csv("/data/olist/olist_sellers_dataset.csv", header=True, inferSchema=True)
product_category_name_translation_df = spark.read.csv("/data/olist/product_category_name_translation.csv",
header=True, inferSchema=True)
```

## Data Exploration

```
customers_df.printSchema()
```

```
order_items_df.printSchema()
```

```
# Check for Data Leakage
print(f'Customers : {customers_df.count()} rows')
```

```
print(f'Orders : {orders_df.count()} rows')
```

## Check for Null

```
from pyspark.sql.functions import *
# unnecessary imports make things slower to boot up and testing would take longer
```

```
customers_df.select([count(when(col(c).isNull(),1)).alias(c) \
for c in customers_df.columns]).show()
```

```
# Duplicate Values
customers_df.groupBy('customer_id').count().filter('count>1').show()
```

```
customers_df.groupBy('customer_state').count().orderBy('count',ascending=False).show()
```

```
orders_df.groupBy('order_status').count().orderBy('count',ascending=False).show()
```

```
orders_df.show()
```

```
order_payments_df.show()
```

```
order_payments_df.groupBy('payment_type').count().orderBy('count',ascending=False).show()
```

```
order_items_df.show(5)
```

```
# top products
from pyspark.sql.functions import sum
top_products = order_items_df.groupBy('product_id').agg(sum('price').alias('total_sales'))
top_products.orderBy('total_sales', ascending = False).show(20)
```

## Data Cleaning and Transformation

Steps in data cleaning and transformation

- Identify issues -> Missing values , duplicates , invalid data
- Handle missing data -> Drop or fill Null values , impute
- Standardize formats -> Normalize, standardize date time
- Data Type Correction -> Correct data type for each column
- De-Duplication
- Data Transformation
- Store Clean Data -> In HDFS, parquet

```
from pyspark.sql.functions import *
# Identify Missing Values
def missing_values(df,df_name):
    print(f'Missing values in {df_name}')
    df.select([count(when(col(c).isNull(),1)).alias(c) for c in df.columns]).show(5)
```

```
missing_values(customer_df,"customer_df")
```

```
missing_values(order_payments_df,"order_payments_df")
```

## Handling missing values

```
orders_df_cleaned = orders_df.na.drop(subset =
['order_id','customer_id','order_status'])
# Other columns missing values can be filled.
# These columns are unique and cannot be estimated. So dropping is the best option
orders_df_cleaned.show()
```

```
#Fill with static value
orders_df_cleaned = orders_df.fillna({'order_delivered_customer_date':'9999-12-31'})
orders_df_cleaned.show(10)
```

```
# Impute missing values
from pyspark.ml.feature import Imputer
# Fill Null values in payment_value column with its median
imputer = Imputer(inputCols = ['payment_value'], outputCols =
['payment_value_imputed']).setStrategy('median')
order_payments_df_cleaned = imputer.fit(order_payments_df).transform(order_payments_df)
order_payments_df_cleaned.show()
```

## Standardizing the format

```
def print_schema(df,df_name):
    print(f'Schema in {df_name} :')
    df.printSchema()
```

```
orders_df.show(5)
```

```
print_schema(orders_df, 'orders')
```

```
orders_df_cleaned =
orders_df.withColumn('order_purchase_timestamp',to_date(col('order_purchase_timestamp')))
print_schema(orders_df_cleaned, 'orders_cleaned')
```

```
orders_df_cleaned.show(5) # Changed YYYY-MM-DD HH:MM:SS to YYYY-MM-DD
```

```
# change boleto to Bank Transfer , card and credit_card to Credit Card
order_payments_df_cleaned = order_payments_df.withColumn('payment_type',when(col('payment_type')=='boleto','Bank Transfer')\
.when(col('payment_type')=='credit_card','Credit Card').otherwise('other'))
order_payments_df_cleaned.show(10)
```

## Remove Duplicates

```
customers_df_cleaned = customers_df.dropDuplicates(['customer_id'])
customers_df_cleaned.show(5)
```

## Data Transformation

```
order_with_details = orders_df.join(order_items_df,'order_id','left')\
    .join(order_payments_df,'order_id','left')\
    .join(customers_df,'customer_id','left')
order_with_details.show(5)
```

```
# Find total order value
order_with_total_values = order_with_details.groupBy('order_id')\
    .agg(sum('payment_value').alias('total_amount')).orderBy('total_amount',ascending=False)
order_with_total_values.show(10)
```

```
#Quantiles
quantiles = order_items_df.approxQuantile('price',[0.01,0.99],0.0)
quantiles
order_items_df.select('price').summary().show()

# Remove outliers
low_cutoff,high_cutoff = quantiles[0],quantiles[1]

order_item_df_filtered = order_items_df.filter((col('price')>=low_cutoff) &
(col('price')<=high_cutoff))
order_item_df_filtered.select('price').summary().show()

products_cleaned = products_df.withColumn('product_size_category',
when(col('product_weight_g')<500,'small')\
.when(col('product_weight_g').between(500,2000),'medium').otherwise('large'))
products_cleaned.select('product_weight_g','product_size_category').show(5)
```

## Data Integration and Aggregation

```
# Cache frequently used data for better performance
olist_orders_df.cache()

order_items_joined_df = orders_df.join(order_items_df,'order_id','inner')
order_items_product_df = order_items_joined_df.join(products_df,'product_id','inner')
order_items_product_seller_df = order_items_product_df.join(sellers_df,'seller_id','inner')
full_orders_df = order_items_product_seller_df.join(customers_df,'customer_id','inner')
full_orders_df = full_orders_df.join(geolocation_df,full_orders_df.customer_zip_code_prefix ==geolocation_df.geolocation_zip_code_prefix , 'inner')
full_orders_df= full_orders_df.join(broadcast(order_reviews_df),'order_id','left')
full_orders_df = full_orders_df.join(order_payments_df,'order_id','left')
full_orders_df.cache()
seller_revenue_df = full_orders_df.groupBy('seller_id').agg(sum('price'))
seller_revenue_df.show(5)
```

# Spark Optimization: Broadcasting, Caching

## The Join Operation in Spark

In Spark, joining two DataFrames is a common but often **expensive operation** because:

- Data may need to be **shuffled across nodes** (network transfer).
- Large datasets may not fit into memory efficiently.
- If both datasets are large, joins can slow down due to shuffles and spill to disk.

### Example

```
full_orders_df = full_orders_df.join(broadcast(order_reviews_df), 'order_id', 'left')
```

Here:

- full\_orders\_df → Large DataFrame (e.g., millions of customer orders).
- order\_reviews\_df → Smaller DataFrame (e.g., reviews for some orders).
- 'order\_id' → The common column to join on.
- 'left' → Left join ensures all orders remain, even if no matching review exists.

## Broadcasting in Spark

### ◊ What is Broadcast Join?

- If one DataFrame is **small enough** (fits into memory), Spark can **broadcast** it to all worker nodes.
- Instead of shuffling both datasets, the smaller one is **replicated to each executor**, and join happens locally with partitions of the larger dataset.

### ◊ Why Use broadcast()?

- **Avoids shuffle** of the large dataset.
- Faster joins since small dataset is already in memory.
- Good for **star-schema joins** (large fact table + small dimension table).

### ◊ Example

```
from pyspark.sql.functions import broadcast
```

```
optimized_df = full_orders_df.join(broadcast(order_reviews_df), 'order_id', 'left')
```

Here, order\_reviews\_df is small (say, few MBs), so Spark sends it to every executor. The large full\_orders\_df doesn't shuffle → performance gain.

### ◊ When to Use:

- Dimension tables, lookup tables, or small reference data.
- Dataset fits within **broadcast threshold** (default: 10 MB in Spark, configurable via spark.sql.autoBroadcastJoinThreshold).

## Caching in Spark

### ◊ What is Caching?

- Spark uses **lazy evaluation**: transformations aren't executed until an action (count(), collect(), write) is triggered.
- If the same DataFrame/RDD is reused multiple times, Spark recomputes it each time.
- **Caching** (or persist()) stores intermediate results in memory (or disk if needed).

### ◊ Example:

```
full_orders_df.cache()
```

or

```
full_orders_df.persist(StorageLevel.MEMORY_AND_DISK)
```

#### ◆ Benefits

- Speeds up iterative algorithms and repeated queries.
- Prevents re-computation of expensive transformations.
- Useful in ETL pipelines, ML training, and interactive queries.

## Optimization Summary

Technique	What it Does	When to Use	Benefit
Broadcast Join	Sends small DataFrame to all executors	When joining large + small dataset	Avoids shuffle, faster join
Caching	Stores results in memory/disk	When reusing same DataFrame multiple times	Avoids recomputation, speeds queries

## Best Practices

- Use broadcast() explicitly for small lookup/reference tables.
- Monitor with **Spark UI** → check if broadcast is used (look for "BroadcastHashJoin").
- Tune spark.sql.autoBroadcastJoinThreshold if needed.
- Cache only when reusing a dataset; otherwise, caching wastes memory.
- Partition data properly in HDFS for parallelism.

# Spark Session Builder Configs

Config / Option	Example Code	Explanation
<b>App Name</b>	<code>.appName("MyApp")</code>	Sets the name of your Spark application (shows up in Spark UI). Helpful for identifying jobs in Spark UI or YARN/cluster manager.
<b>Master URL</b>	<code>.master("local[*]")</code>	Defines the cluster master: <ul style="list-style-type: none"> <li>• "local[*]" → local mode using all cores</li> <li>• "yarn" → Hadoop YARN cluster</li> <li>• "spark://host:port" → Spark standalone cluster.</li> </ul>
<b>Executor Memory</b>	<code>.config("spark.executor.memory", "4g")</code>	Amount of RAM allocated per executor. Too low = OOM, too high = waste/less parallelism.
<b>Driver Memory</b>	<code>.config("spark.driver.memory", "2g")</code>	RAM allocated to the driver. Needs to be high enough to handle collected results and metadata.
<b>Executor Cores</b>	<code>.config("spark.executor.cores", "4")</code>	Number of CPU cores used per executor. More cores → more tasks in parallel. Needs balance with number of executors.
<b>Dynamic Allocation</b>	<code>.config("spark.dynamicAllocation.enabled","true")</code>	Spark can add/remove executors dynamically based on workload. Useful in shared clusters to save resources.
<b>Shuffle Partitions</b>	<code>.config("spark.sql.shuffle.partitions","200")</code>	Number of partitions for shuffle ops (joins, groupBy, etc). Default <b>200</b> , but often too high for small data → tune lower.
<b>Auto Broadcast Join Threshold</b>	<code>.config("spark.sql.autoBroadcastJoinThreshold", 50*1024*1024)</code>	Maximum table size (in bytes) that Spark will <b>auto-broadcast</b> in a join. Default = <b>10 MB</b> . <ul style="list-style-type: none"> <li>• Higher = faster joins for small dimension tables</li> <li>• -1 disables auto-broadcast.</li> </ul>
<b>Max Partition Bytes</b>	<code>.config("spark.sql.files.maxPartitionBytes", 134217728)</code>	(Default: <b>128 MB</b> ) Maximum number of bytes to pack into a single partition when reading files. Controls parallelism of file scans. Smaller → more partitions (higher parallelism), larger → fewer partitions (less overhead but possible skew).
<b>Open Cost In Bytes</b>	<code>.config("spark.sql.files.openCostInBytes", 4194304)</code>	(Default: <b>4 MB</b> ) Estimated cost of opening a file. Spark uses this to decide how to split small files into partitions. A higher value → fewer small files grouped together. Useful for optimizing <b>small files problem</b> .
<b>Max Result Size</b>	<code>.config("spark.driver.maxResultSize","1g")</code>	(Default: <b>1g</b> ) Maximum result size the driver can collect (per action). Prevents driver OOM. <ul style="list-style-type: none"> <li>• "0" = unlimited</li> </ul>

		(dangerous) • Tune based on driver memory.
<b>Memory Fraction</b>	.config("spark.memory.fraction", "0.6")	(Default: <b>0.6</b> ) Fraction of JVM heap dedicated to Spark execution + storage. Remaining memory is for user data structures, RDD metadata, etc.
<b>Memory Storage Fraction</b>	.config("spark.memory.storageFraction", "0.5")	(Default: <b>0.5</b> ) Fraction of Spark memory reserved for storage (cached RDDs, broadcast vars). Remaining goes to execution (shuffles, joins, sorts). Increasing this helps caching but may hurt shuffle performance.
<b>Checkpoint Directory</b>	.config("spark.checkpoint.dir","hdfs://checkpoints")	Directory used for storing checkpoints (mainly in streaming or long lineage DAGs).
<b>Warehouse Dir</b>	.config("spark.sql.warehouse.dir","/user/hive/warehouse")	Default directory for managed Spark SQL/Hive tables.
<b>Kryo Serializer</b>	.config("spark.serializer","org.apache.spark.serializer.KryoSerializer")	Faster, more compact serializer than Java's default. Recommended for production workloads.

# Spark Optimization A–Z (Beginner Friendly)

Apache Spark is great for big data, but **unoptimized queries** can be slow and resource-heavy.  
Here are **five core optimizations** every beginner should know:

## Bucketing

### ◊ What is Bucketing?

- Bucketing **organizes data into fixed “buckets” (files)** based on the hash of a column.
- Example: If you bucket by user\_id into 8 buckets → all rows with the same user\_id always go into the **same bucket file**.

### ◊ Why Use Bucketing?

- Makes **joins and aggregations faster** because Spark doesn't need to reshuffle all data.
- Useful when joining **two very large tables** on the same key.

### ◊ Example

```
# Write a DataFrame bucketed by user_id
df.write.bucketBy(8, "user_id").sortBy("user_id").saveAsTable("bucketed_users")

# Later join with another bucketed table on the same column
orders.write.bucketBy(8, "user_id").saveAsTable("bucketed_orders")
```

Now, joining bucketed\_users and bucketed\_orders avoids shuffle → faster.

**Best for:** Large tables frequently joined on the same column.

**Limit:** Buckets are fixed at write time → less flexible than repartition.

## Handling Data Skew

### ◊ What is Data Skew?

- **Skew = uneven distribution of data** across partitions.
- Example: If customer\_id = 123 appears in **80% of rows**, one partition gets overloaded → job slows down.

### ◊ Why It's a Problem?

- Some executors finish fast, but one executor (handling the skewed key) becomes a **straggler**.
- Leads to long-running tasks and wasted resources.

### ◊ Techniques to Handle Skew

#### 1. Salting (Random Prefixing)

Add a random number to skewed keys to spread them across partitions.

```
from pyspark.sql.functions import concat_ws, rand
df = df.withColumn("salted_key", concat_ws("_", df.customer_id, (rand()*10).cast("int")))
```

#### 2. Skew Join Hint (Spark ≥ 3.0)

Let Spark handle skew automatically:

```
result = df1.join(df2.hint("skew"), "id")
```

#### 3. Broadcast Join

If one table is small, broadcast it instead of shuffling.

```
result = large_df.join(broadcast(small_df), "id")
```

**Best for:** Joins/aggregations where some keys dominate.

**Watch out:** Salting may require "unsalting" later to recombine results.

## Predicate Pushdown

### ◊ What is Predicate Pushdown?

- Predicate = **filter condition** (e.g., WHERE order\_date > '2025-01-01').
- Spark can **push filters down to the storage layer** (Parquet/ORC/JDBC) → only read relevant data.

### ◊ Example

```
# Spark pushes the filter to Parquet reader
```

```
df = spark.read.parquet("hdfs://orders").filter("order_date > '2025-01-01'")
```

Instead of reading entire table → Spark only loads blocks with matching dates.

**Best for:** Large Parquet/ORC/Delta tables.

**Not supported** for CSV/JSON (they must be fully read).

## Column Pruning

### ◊ What is Column Pruning?

- Instead of reading **all columns**, Spark only reads the columns you actually need.
- Saves **I/O, memory, and CPU**.

### ◊ Example:

```
# Instead of selecting all columns
```

```
df = spark.read.parquet("hdfs://orders").select("order_id", "customer_id")
```

If Parquet has 50 columns but you only need 2 → Spark only reads 2.

**Best for:** Wide tables with many columns.

**Works best** with columnar formats (Parquet, ORC, Delta).

## Adaptive Query Execution (AQE)

### ◊ What is AQE?

- Introduced in **Spark 3.0**.
- Spark adjusts the **query plan dynamically at runtime** (instead of sticking to the static plan).

### ◊ What AQE Does

#### 1. Coalesces Shuffle Partitions

- If Spark created 200 shuffle partitions but only 20 are non-empty → AQE reduces them automatically.

#### 2. Handles Skew Joins

- Detects skew and splits skewed partitions into smaller ones.

#### 3. Changes Join Strategy

- If Spark planned a Sort-Merge Join but detects small data → switches to Broadcast Join at runtime.

### ◊ Enable AQE

```
spark.conf.set("spark.sql.adaptive.enabled", True)
```

**Benefit:** Smarter queries without manual tuning.

**Note:** AQE works best with Spark SQL & DataFrame API (not raw RDDs).

## Summary Cheat Sheet

Optimization	What It Does	Beginner Example	Benefit
<b>Bucketing</b>	Pre-divides data into fixed hash buckets	<code>bucketBy(8, "user_id")</code>	Avoids shuffle in joins
<b>Data Skew Handling</b>	Fixes uneven data distribution	Salting, skew hints	Prevents stragglers
<b>Predicate Pushdown</b>	Pushes filters to storage layer	<code>.filter("order_date &gt; '2025-01-01'")</code>	Reads less data
<b>Column Pruning</b>	Reads only needed columns	<code>.select("id", "name")</code>	Less I/O & memory
<b>AQE</b>	Adaptive query optimization at runtime	<code>spark.conf.set("spark.sql.adaptive.enabled", True)</code>	Smarter, faster queries

## GCP vs AWS vs Azure – Service Comparison Table

Category	AWS	Azure	GCP
<b>Compute</b>	EC2 (Elastic Compute Cloud)	Virtual Machines (VMs)	Compute Engine
<b>Serverless Compute</b>	AWS Lambda	Azure Functions	Cloud Functions
<b>Containers (K8s)</b>	Amazon EKS	Azure Kubernetes Service (AKS)	Google Kubernetes Engine (GKE)
<b>App Deployment (PaaS)</b>	AWS Elastic Beanstalk	Azure App Service	App Engine
<b>Storage (Object)</b>	Amazon S3	Azure Blob Storage	Google Cloud Storage
<b>Block Storage</b>	Amazon EBS	Azure Disk Storage	Persistent Disks (GCE)
<b>File Storage</b>	Amazon EFS	Azure Files	Filestore
<b>Relational Database</b>	Amazon RDS	Azure SQL Database	Cloud SQL
<b>NoSQL Database</b>	Amazon DynamoDB	Azure Cosmos DB	Cloud Firestore / Datastore
<b>Data Warehousing</b>	Amazon Redshift	Azure Synapse Analytics	BigQuery
<b>Big Data Processing</b>	Amazon EMR	Azure HDInsight	Cloud Dataproc
<b>Stream Processing</b>	Kinesis	Azure Stream Analytics	Cloud Dataflow / Pub/Sub
<b>Message Queues</b>	Amazon SQS	Azure Queue Storage / Service Bus	Pub/Sub
<b>Monitoring</b>	Amazon CloudWatch	Azure Monitor	Cloud Monitoring (Stackdriver)
<b>Logging</b>	AWS CloudTrail + CloudWatch	Azure Log Analytics	Cloud Logging
<b>Identity &amp; Access</b>	IAM	Azure Active Directory (AAD)	IAM + Cloud Identity

AI/ML Platform	SageMaker	Azure ML Studio	Vertex AI
Data Integration (ETL)	AWS Glue	Azure Data Factory	Cloud Data Fusion / Dataflow
DevOps & CI/CD	CodePipeline, CodeBuild	Azure DevOps, Pipelines	Cloud Build, Cloud Deploy
DNS & Networking	Route 53, VPC	Azure DNS, VNet	Cloud DNS, VPC
CDN	CloudFront	Azure CDN	Cloud CDN
Edge/IoT Services	AWS IoT Core	Azure IoT Hub	Cloud IoT Core
Cost Management	AWS Cost Explorer	Azure Cost Management	GCP Billing + Cost Tools

## ✓ Highlights & Strengths

Provider	Strengths
AWS	Most mature and feature-rich, largest ecosystem, global presence
Azure	Strong enterprise integration (especially Microsoft stack), Active Directory
GCP	Best for analytics, AI/ML (BigQuery, Vertex AI), deep integration with Kubernetes

## 💡 Common Service Naming Confusion

Service Type	AWS	Azure	GCP
Object Storage	S3	Blob Storage	Cloud Storage
Serverless	Lambda	Azure Functions	Cloud Functions
SQL Database	RDS	SQL DB	Cloud SQL
NoSQL	DynamoDB	Cosmos DB	Firestore
Big Data	EMR	HDInsight	Dataproc
Warehousing	Redshift	Synapse	BigQuery

## 🧠 Final Notes

- **AWS:** Great for **breadth of services**, performance, and ecosystem scale
- **Azure:** Ideal for **Microsoft-heavy enterprises** and hybrid cloud
- **GCP:** Best for **developers, data science, analytics, and Kubernetes-native apps**

Additional:

# Spark SQL

## Spark SQL Basics

Apache Spark SQL is a module in Spark for **structured data processing**, combining the power of SQL with Spark's distributed computation engine. It allows you to query data using **DataFrame API** or **SQL queries**.

### SparkSession

#### What is SparkSession?

- **SparkSession** is the **entry point** for using Spark SQL and DataFrames.
- It replaces older **SQLContext** and **HiveContext** APIs.
- Through SparkSession, you can:
  - Read data from files or databases
  - Create DataFrames
  - Execute SQL queries

#### Creating SparkSession

```
from pyspark.sql import SparkSession

# Create a SparkSession
spark = SparkSession.builder \
    .appName("SparkSQL_Basics") \
    .master("local[*]") \   # use all available CPU cores
    .getOrCreate()
```

- **appName**: Name of your Spark application
- **master**: Cluster type. `local[*]` runs Spark locally using all cores.
- **getOrCreate()**: Returns existing `SparkSession` if it exists, else creates a new one.

#### Configuring SparkSession

```
spark = SparkSession.builder \
    .appName("SparkSQL_Basics") \
    .config("spark.sql.shuffle.partitions", "10") \  # set default shuffle partitions
    .config("spark.executor.memory", "2g") \          # executor memory
    .getOrCreate()

• spark.sql.shuffle.partitions → number of partitions for shuffle operations
• spark.executor.memory → memory for each executor
```

## DataFrame API vs SQL API

Feature	DataFrame API	SQL API
Syntax	Python/Scala/Java	SQL queries (string-based)
Use Case	Programmatic transformations	Quick queries, reporting
Example	df.filter(df.age > 30)	spark.sql("SELECT * FROM df WHERE age > 30")
Flexibility	Full control with functions	Limited to SQL syntax

 You can mix both APIs: register DataFrame as a temp view and query with SQL.

## Loading Data into Spark SQL

Spark SQL can read data from **local or HDFS**, supporting multiple formats

### ◆ CSV

```
df = spark.read.csv("data/orders.csv", header=True, inferSchema=True)
```

- header=True → first row as column names
- inferSchema=True → automatically detect data types

### ◆ JSON

```
df = spark.read.json("data/orders.json")
```

### ◆ Parquet (columnar format)

```
df = spark.read.parquet("data/orders.parquet")
```

### ◆ ORC

```
df = spark.read.orc("data/orders.orc")
```

### ◆ JDBC (database)

```
df = spark.read.format("jdbc") \
.option("url", "jdbc:mysql://localhost:3306/salesdb") \
.option("dbtable", "orders") \
.option("user", "root") \
.option("password", "password") \
.load()
```

## DataFrame Operations

Once data is loaded, **DataFrames** allow powerful transformations:

- ◆ **Select columns**

```
df.select("order_id", "customer_id").show(5)
```

- ◆ **Filter rows**

```
df.filter(df.order_amount > 100).show()  
# OR using SQL expressions
```

```
df.filter("order_amount > 100").show()
```

- ◆ **Group By & Aggregation**

```
from pyspark.sql.functions import sum, avg  
  
df.groupBy("customer_id").agg(  
    sum("order_amount").alias("total_amount"),  
    avg("order_amount").alias("avg_amount")  
).show()
```

- ◆ **Join DataFrames**

```
df_orders = spark.read.parquet("orders.parquet")  
df_customers = spark.read.parquet("customers.parquet")  
  
df_joined = df_orders.join(df_customers, df_orders.customer_id ==  
df_customers.customer_id, "inner")  
df_joined.show()
```

## Viewing and Exploring Data

- ◆ **Show top rows**

```
vdf.show(5) # display first 5 rows
```

- ◆ **Schema of DataFrame**

```
df.printSchema()
```

- ◆ **Describe data**

```
df.describe().show()  
• Gives count, mean, stddev, min, max for numeric columns
```

- ◆ **Count rows**

```
df.count()
```

- ◆ **Columns and data types**

```
df.columns  
df.dtypes
```

## Putting It Together: Example

```
from pyspark.sql import SparkSession  
from pyspark.sql.functions import sum, avg  
  
spark = SparkSession.builder.appName("SparkSQL_Basics").master("local[*]").getOrCreate()  
  
# Load CSV data  
orders = spark.read.csv("data/orders.csv", header=True, inferSchema=True)  
  
# Select columns  
orders.select("order_id", "customer_id", "order_amount").show(5)  
  
# Filter high-value orders  
high_orders = orders.filter(orders.order_amount > 500)  
  
# Aggregate total and average order amount by customer  
agg_orders = high_orders.groupBy("customer_id").agg(  
    sum("order_amount").alias("total_amount"),  
    avg("order_amount").alias("avg_amount")  
)  
  
agg_orders.show()  
  
# Count total orders  
print("Total orders:", orders.count())  
  
# View schema  
orders.printSchema()
```

# Spark SQL Data Types

In Spark SQL, **data types define the kind of data stored in columns** of a DataFrame. Choosing the right data type helps with **performance, accuracy, and compatibility**.

## Primitive Data Types

Primitive types are **basic data types** used for single values in columns.

Data Type	Description	Example
<b>StringType</b>	Text/string data	"Alice", "Bob"
<b>IntegerType</b>	32-bit signed integer	100, -25
<b>LongType</b>	64-bit signed integer	1234567890
<b>DoubleType</b>	Double-precision floating point number	12.5, 3.1415
<b>BooleanType</b>	True/False values	True, False
<b>FloatType</b>	Single-precision floating point	3.14
<b>DateType</b>	Calendar date (YYYY-MM-DD)	"2025-08-30"
<b>TimestampType</b>	Date & time (YYYY-MM-DD HH:MM:SS)	"2025-08-30 14:00:00"

## Example: Using Primitive Types

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType, IntegerType, DoubleType

spark = SparkSession.builder.appName("DataTypesExample").getOrCreate()

data = [("Alice", 30, 2500.50), ("Bob", 25, 1800.75)]
schema = StructType([
    StructField("name", StringType(), True),
    StructField("age", IntegerType(), True),
    StructField("salary", DoubleType(), True)
])

df = spark.createDataFrame(data, schema)
df.printSchema()
df.show()
```

Output:

root

|-- name: string (nullable = true)

|-- age: integer (nullable = true)

|-- salary: double (nullable = true)

```
+---+---+-----+
| name|age| salary|
+---+---+-----+
|Alice| 30| 2500.5|
| Bob| 25| 1800.75|
+---+---+-----+
```

## Complex Data Types

Complex types allow **nested or multi-valued data** in a single column.

### 2.1 ArrayType

- Stores an **array (list) of values** in a single column.
- All elements must have the **same data type**.

```
from pyspark.sql.types import ArrayType

data = [("Alice", [85, 90, 95]), ("Bob", [70, 75, 80])]
schema = StructType([
    StructField("name", StringType(), True),
    StructField("marks", ArrayType(IntegerType()), True)
])
df = spark.createDataFrame(data, schema)
df.show()
```

Output:

```
+---+-----+
| name|      marks|
+---+-----+
|Alice| [85, 90, 95]|
| Bob| [70, 75, 80]|
+---+-----+
```

### 2.2 MapType

- Stores **key-value pairs** in a single column.
- Keys and values can have **different data types**.

```
from pyspark.sql.types import MapType

data = [("Alice", {"math": 90, "eng": 95}), ("Bob", {"math": 75, "eng": 80})}
schema = StructType([
    StructField("name", StringType(), True),
    StructField("marks_map", MapType(StringType(), IntegerType()), True)
])
df = spark.createDataFrame(data, schema)
df.show(truncate=False)
```

Output:

```
+---+-----+
|name |marks_map      |
+---+-----+
|Alice|{math -> 90, eng -> 95}|
|Bob  |{math -> 75, eng -> 80}|
+---+-----+
```

## 2.3 StructType

- Allows **nested schema** (a column can have multiple subfields).
- Useful for hierarchical data like JSON.

```
from pyspark.sql.types import StructType, StructField, StringType, IntegerType

address_schema = StructType([
    StructField("city", StringType(), True),
    StructField("zip", IntegerType(), True)
])

data = [("Alice", ("New York", 10001)), ("Bob", ("LA", 90001))]
schema = StructType([
    StructField("name", StringType(), True),
    StructField("address", address_schema, True)
])
df = spark.createDataFrame(data, schema)
df.show(truncate=False)
```

Output:

```
+---+-----+
|name |address      |
+---+-----+
|Alice|{New York, 10001}|
|Bob  |{LA, 90001}|
+---+-----+
```

## Schema Inference and Manual Schema Specification

### Schema Inference

- Spark can **automatically detect schema** when loading data.
- Works well for CSV, JSON, Parquet, ORC.

```
df = spark.read.csv("data/orders.csv", header=True, inferSchema=True)  
df.printSchema()
```

✓ Pros:

- Quick and easy

⚠ Cons:

- Can be **slower for very large datasets**
- May **infer wrong type** (e.g., String instead of Integer)

### Manual Schema Specification

- Recommended for **accuracy and performance**.
- Define schema using StructType and StructField.

```
from pyspark.sql.types import StructType, StructField, StringType, IntegerType  
  
schema = StructType([  
    StructField("order_id", StringType(), True),  
    StructField("customer_id", StringType(), True),  
    StructField("amount", IntegerType(), True)  
)  
  
df = spark.read.csv("data/orders.csv", schema=schema, header=True)  
df.printSchema()
```

✓ Pros

- Ensures **correct types**
- Faster because Spark **doesn't infer schema**
- Necessary when loading data from **external sources like CSV or JSON**

## Spark SQL – Working with SQL Queries [Transformations]

Spark SQL allows querying data either through **SQL queries** or the **DataFrame API**. Both approaches are equivalent, but the DataFrame API is more **programmatic and Pythonic**, while SQL is **declarative**.

### Sample DataFrame

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, row_number, rank, lead, lag
from pyspark.sql.window import Window

spark = SparkSession.builder.appName("SQL_vs_DF").getOrCreate()

data = [
    (1, "Alice", 100, "2025-08-01"),
    (2, "Bob", 200, "2025-08-02"),
    (3, "Alice", 150, "2025-08-03"),
    (4, "Bob", 50, "2025-08-04"),
    (5, "Charlie", 300, "2025-08-05")
]

columns = ["order_id", "customer_id", "amount", "order_date"]
df = spark.createDataFrame(data, columns)
df.show()
```

**Output:**

```
+-----+-----+-----+
|order_id|customer_id|amount|order_date|
+-----+-----+-----+
|      1|      Alice|   100|2025-08-01|
|      2|        Bob|   200|2025-08-02|
|      3|      Alice|   150|2025-08-03|
|      4|        Bob|    50|2025-08-04|
|      5|    Charlie|   300|2025-08-05|
+-----+-----+-----+
```

### Creating Temporary Views

```
df.createOrReplaceTempView("orders")
```

- Now SQL queries can be executed on "orders".

## SQL Queries vs DataFrame API

### SELECT & WHERE

#### SQL

```
SELECT order_id, customer_id, amount
FROM orders
WHERE amount > 100
```

#### DataFrame API:

```
df.filter(col("amount") > 100).select("order_id", "customer_id", "amount").show()
```

#### Output

```
+-----+-----+
|order_id|customer_id|amount|
+-----+-----+
|      2|        Bob|    200|
|      3|      Alice|    150|
|      5|   Charlie|    300|
+-----+-----+
```

### GROUP BY & HAVING

#### SQL:

```
SELECT customer_id, SUM(amount) AS total_amount
FROM orders
GROUP BY customer_id
HAVING SUM(amount) > 200
```

#### DataFrame API:

```
from pyspark.sql.functions import sum

df.groupBy("customer_id") \
  .agg(sum("amount").alias("total_amount")) \
  .filter(col("total_amount") > 200) \
  .show()
```

#### Output:

```
+-----+
|customer_id|total_amount|
+-----+
|      Alice|      250|
|   Charlie|      300|
+-----+
```

## ORDER BY

**SQL:**

```
SELECT * FROM orders ORDER BY amount DESC
```

**DataFrame API:**

```
df.orderBy(col("amount").desc()).show()
```

**Output:**

```
+-----+-----+-----+
|order_id|customer_id|amount|order_date|
+-----+-----+-----+
|      5|    Charlie|   300|2025-08-05|
|      2|        Bob|   200|2025-08-02|
|      3|     Alice|   150|2025-08-03|
|      1|     Alice|   100|2025-08-01|
|      4|        Bob|    50|2025-08-04|
+-----+-----+-----+
```

## DISTINCT

**SQL:**

```
SELECT DISTINCT customer_id FROM orders
```

**DataFrame API:**

```
df.select("customer_id").distinct().show()
```

**Output:**

```
+-----+
|customer_id|
+-----+
|    Alice|
|    Bob|
|  Charlie|
+-----+
```

## LIMIT

### SQL:

```
SELECT * FROM orders LIMIT 2
```

### DataFrame API:

```
df.limit(2).show()
```

### Output:

```
+-----+-----+-----+
|order_id|customer_id|amount|order_date|
+-----+-----+-----+
|      1|     Alice|   100|2025-08-01|
|      2|      Bob|   200|2025-08-02|
+-----+-----+-----+
```

## Window Functions (ROW\_NUMBER, RANK, LEAD, LAG)

### SQL Example:

```
SELECT customer_id, order_id, amount,
       ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY amount DESC) AS row_num,
       RANK() OVER (PARTITION BY customer_id ORDER BY amount DESC) AS rank
FROM orders
```

### DataFrame API Equivalent:

```
window_spec = Window.partitionBy("customer_id").orderBy(col("amount").desc())

df.withColumn("row_num", row_number().over(window_spec)) \
  .withColumn("rank", rank().over(window_spec)) \
  .show()
```

### Output:

```
+-----+-----+-----+-----+
|customer_id|order_id|amount|row_num|rank|
+-----+-----+-----+-----+
|     Alice|      3|   150|      1|    1|
|     Alice|      1|   100|      2|    2|
|      Bob|      2|   200|      1|    1|
|      Bob|      4|    50|      2|    2|
| Charlie|      5|   300|      1|    1|
+-----+-----+-----+-----+
```

## Subqueries & CTEs

### SQL with CTE:

```
WITH high_orders AS (
    SELECT * FROM orders WHERE amount > 100
)
SELECT customer_id, COUNT(*) AS num_orders
FROM high_orders
GROUP BY customer_id
```

### DataFrame API Equivalent:

```
high_orders = df.filter(col("amount") > 100)
high_orders.groupBy("customer_id").count().alias("num_orders").show()
```

### Output:

```
+-----+-----+
|customer_id|count|
+-----+-----+
|      Alice|     1|
|       Bob|     1|
|   Charlie|     1|
+-----+-----+
```

## Joins (with another sample customers DataFrame)

```
customers_data = [
    ("Alice", "NY"),
    ("Bob", "LA"),
    ("Charlie", "SF")
]
customers_df = spark.createDataFrame(customers_data, ["customer_id", "city"])

# Inner Join
df.join(customers_df, on="customer_id", how="inner").show()
```

### Output:

```
+-----+-----+-----+-----+
|customer_id|order_id|amount|order_date|city|
+-----+-----+-----+-----+
|      Alice|     1|   100|2025-08-01|   NY|
|      Alice|     3|   150|2025-08-03|   NY|
|       Bob|     2|   200|2025-08-02|   LA|
|       Bob|     4|    50|2025-08-04|   LA|
|   Charlie|     5|   300|2025-08-05|   SF|
+-----+-----+-----+-----+
```

- Supports inner, left, right, full\_outer joins.

# Spark SQL Functions

Spark SQL provides a rich set of built-in functions for string manipulation, date/time operations, mathematical calculations, conditional logic, and aggregation. You can also create **User-Defined Functions (UDFs)** for custom logic.

## Built-in Functions

### 1.1 String Functions

Function	Description	Example
<b>concat</b>	Concatenate multiple columns or strings	concat(col("first_name"), lit(" "), col("last_name"))
<b>substring</b>	Extract substring from a string	substring(col("name"), 1, 3) → first 3 chars
<b>lower</b>	Convert string to lowercase	lower(col("name")) → "alice"
<b>upper</b>	Convert string to uppercase	upper(col("name")) → "ALICE"

```
from pyspark.sql.functions import concat, col, lit, substring, lower, upper

df.select(
    concat(col("first_name"), lit(" "), col("last_name")).alias("full_name"),
    substring(col("first_name"), 1, 3).alias("first3"),
    lower(col("last_name")).alias("lname_lower"),
    upper(col("last_name")).alias("lname_upper")
).show()
```

### 1.2 Date & Time Functions

Function	Description	Example
<b>current_date</b>	Returns current date	current_date() → "2025-08-30"
<b>datediff</b>	Difference between two dates	datediff(col("end"), col("start"))
<b>date_add</b>	Add days to a date	date_add(col("start"), 10)
<b>year</b>	Extract year from date	year(col("date"))
<b>month</b>	Extract month from date	month(col("date"))

```
from pyspark.sql.functions import current_date, datediff, date_add, year, month

df.select(
    current_date().alias("today"),
    datediff(col("end_date"), col("start_date")).alias("days_diff"),
    date_add(col("start_date"), 10).alias("date_plus_10"),
    year(col("start_date")).alias("start_year"),
    month(col("start_date")).alias("start_month")
).show()
```

## 1.3 Mathematical Functions

Function	Description	Example
<b>round</b>	Round to nearest integer	round(col("amount"), 2)
<b>ceil</b>	Round up	ceil(col("amount"))
<b>floor</b>	Round down	floor(col("amount"))
<b>abs</b>	Absolute value	abs(col("amount"))

```
from pyspark.sql.functions import round, ceil, floor, abs

df.select(
    round(col("amount"), 2).alias("rounded"),
    ceil(col("amount")).alias("ceiled"),
    floor(col("amount")).alias("floored"),
    abs(col("amount")).alias("absolute")
).show()
```

## 1.4 Conditional Functions

Function	Description	Example
<b>when</b>	Conditional expressions (like IF)	when(col("amount") > 100, "High").otherwise("Low")
<b>coalesce</b>	First non-null value	coalesce(col("col1"), col("col2"), lit(0))
<b>ifnull</b>	Replace NULL with a value	ifnull(col("amount"), 0)
<b>nvl</b>	Similar to ifnull	nvl(col("amount"), 0)

```
from pyspark.sql.functions import when, coalesce, lit, ifnull, nvl

df.select(
    when(col("amount") > 100, "High").otherwise("Low").alias("amount_level"),
    coalesce(col("amount"), lit(0)).alias("amount_nonnull"),
    ifnull(col("amount"), 0).alias("amount_ifnull"),
    nvl(col("amount"), 0).alias("amount_nvl")
).show()
```

## 1.5 Aggregation Functions

Function	Description	Example
<b>sum</b>	Sum of values	sum(col("amount"))
<b>avg</b>	Average of values	avg(col("amount"))
<b>count</b>	Count of rows	count(col("order_id"))
<b>max</b>	Maximum value	max(col("amount"))
<b>min</b>	Minimum value	min(col("amount"))

```
from pyspark.sql.functions import sum, avg, count, max, min

df.groupBy("customer_id").agg(
    sum("amount").alias("total_amount"),
    avg("amount").alias("avg_amount"),
    count("order_id").alias("num_orders"),
    max("amount").alias("max_amount"),
    min("amount").alias("min_amount")
).show()
```

## User-Defined Functions (UDFs)

- Sometimes **built-in functions are not enough**.
- You can create **custom functions** and register them as **UDFs**.

### 2.1 Creating UDFs

```
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType

# Python function
def categorize_amount(amount):
    if amount > 100:
        return "High"
    else:
        return "Low"

# Register as UDF
categorize_udf = udf(categorize_amount, StringType())
```

### 2.2 Using UDFs in DataFrame API

```
df.select(
    "customer_id", "amount",
    categorize_udf(col("amount")).alias("category")
).show()
```

### 2.3 Registering UDFs for SQL Queries

```
spark.udf.register("categorize_amount_udf", categorize_amount, StringType())

spark.sql("""
    SELECT customer_id, amount, categorize_amount_udf(amount) AS category
    FROM orders
""").show()
```

 Now the UDF can be used **directly in SQL queries**.

# Spark SQL Window Functions

Window functions allow you to **perform calculations across a set of rows related to the current row**. Unlike aggregations, they **do not collapse rows**; each row retains its identity while additional computed columns are added.

## Introduction

- **Window functions** are useful for analytics like:
  - Running totals
  - Ranking rows within a group
  - Comparing current row with previous/next rows (lag/lead)
- They operate **within a “window” of rows** defined by **PARTITION BY** and **ORDER BY**.

## Syntax

```
<window_function>() OVER (
    PARTITION BY column1, column2 ...
    ORDER BY column3 [ASC|DESC]
)
```

- PARTITION BY → groups rows for the function (optional)
- ORDER BY → defines the order within each partition
- Functions do **not aggregate all rows**; they add **new columns** to the original DataFrame

## Common Window Functions

Function	Description
<b>ROW_NUMBER()</b>	Assigns a sequential number per partition
<b>RANK()</b>	Assigns rank, with ties getting same rank
<b>DENSE_RANK()</b>	Like RANK() but no gaps for ties
<b>LEAD(col, n)</b>	Accesses the value of the next n rows
<b>LAG(col, n)</b>	Accesses the value of the previous n rows
<b>SUM(col)</b>	Running total per partition
<b>AVG(col)</b>	Rolling average per partition

## Sample DataFrame

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import row_number, rank, lead, lag, sum
from pyspark.sql.window import Window
from pyspark.sql.functions import col

spark = SparkSession.builder.appName("WindowFunctions").getOrCreate()

data = [
    (1, "Alice", 100),
    (2, "Alice", 150),
    (3, "Bob", 200),
    (4, "Bob", 50),
    (5, "Charlie", 300)
]

columns = ["order_id", "customer_id", "amount"]
df = spark.createDataFrame(data, columns)
df.show()
```

**Output:**

```
+-----+-----+-----+
|order_id|customer_id|amount|
+-----+-----+-----+
|      1|      Alice|    100|
|      2|      Alice|    150|
|      3|        Bob|    200|
|      4|        Bob|     50|
|      5|    Charlie|    300|
+-----+-----+-----+
```

## Example 1 – ROW\_NUMBER & RANK

```
window_spec = Window.partitionBy("customer_id").orderBy(col("amount").desc())

df.withColumn("row_num", row_number().over(window_spec)) \
  .withColumn("rank", rank().over(window_spec)) \
  .show()
```

**Output:**

```
+-----+-----+-----+-----+
|order_id|customer_id|amount|row_num|rank|
+-----+-----+-----+-----+
|      2|      Alice|    150|      1|    1|
|      1|      Alice|    100|      2|    2|
|      3|        Bob|    200|      1|    1|
|      4|        Bob|     50|      2|    2|
|      5|    Charlie|    300|      1|    1|
+-----+-----+-----+-----+
```

- ROW\_NUMBER() → unique sequential number per partition
- RANK() → same amount gets same rank, with possible gaps

## Example 2 – LEAD & LAG

```
df.withColumn("prev_amount", lag("amount", 1).over(window_spec)) \  
  .withColumn("next_amount", lead("amount", 1).over(window_spec)) \  
  .show()
```

Output:

order_id	customer_id	amount	prev_amount	next_amount
2	Alice	150	null	100
1	Alice	100	150	null
3	Bob	200	null	50
4	Bob	50	200	null
5	Charlie	300	null	null

- lag → previous row's value
- lead → next row's value

## Example 3 – Running Total

```
df.withColumn("running_total", sum("amount").over(window_spec)) \  
  .show()
```

Output:

order_id	customer_id	amount	running_total
2	Alice	150	150
1	Alice	100	250
3	Bob	200	200
4	Bob	50	250
5	Charlie	300	300

- Useful for **cumulative sums** or **running aggregates** per customer/partition.

# Spark SQL Optimization Techniques

Optimizing Spark SQL queries is crucial for **reducing execution time, memory usage, and improving scalability**. Spark provides multiple techniques to optimize queries both **programmatically** and via **query planning**.

## Caching and Persistence

- **Caching** stores DataFrames in memory for **repeated use**, avoiding recomputation.
- **Persistence** can store DataFrames in memory, disk, or both, depending on the storage level.

```
# Cache DataFrame in memory
df.cache()

# Persist DataFrame in memory + disk
from pyspark import StorageLevel
df.persist(StorageLevel.MEMORY_AND_DISK)
# Trigger action to materialize cache
df.count()
```

Use when the same DataFrame is **used multiple times**.

## Broadcast Joins

- Useful when **joining a large DataFrame with a small one**.
- Spark sends the **small DataFrame to all nodes**, reducing shuffles.

```
from pyspark.sql.functions import broadcast
df_large.join(broadcast(df_small), "customer_id", "inner").show()
```

Reduces **network shuffle**, improves **join performance**.

## Partition Pruning

- Spark **reads only relevant partitions** instead of the entire dataset.
- Works with **partitioned tables**.

```
# Assuming orders are partitioned by 'year'
df.filter("year = 2025").show()
```

- Only **2025 partition** is scanned → faster query.

## Bucketing

- Bucketing **pre-sorts data into buckets** based on a column (like customer\_id).
- Optimizes **joins and aggregations**.

```
# Save DataFrame as bucketed table
df.write.bucketBy(4, "customer_id").sortBy("amount").saveAsTable("bucketed_orders")
```

Joins on bucketed columns **avoid shuffle**.

## Handling Skewed Data

- **Data skew** occurs when some keys have **many more rows** than others → performance bottleneck.
- Techniques to handle skew:
  1. **Salting** → Add a random prefix to keys and join on the salted key.

```
from pyspark.sql.functions import concat, lit, rand
df_skewed = df.withColumn("salted_key", concat(col("customer_id"), lit("_"),
(rand()*10).cast("int"))))
```

2. **Broadcast the smaller table** if possible.
3. **Repartition** skewed keys.

```
df.repartition(10, "customer_id")
```

## Predicate Pushdown

- **Filters are pushed down** to the data source level (like Parquet, ORC).
- Only **necessary rows are read**.

```
df = spark.read.parquet("orders.parquet").filter("amount > 100")
```

Improves **I/O efficiency**, reduces **data scan**.

## Column Pruning

- Reads **only required columns** instead of the entire row.
- Automatically applied for **column selection**.

```
df.select("order_id", "amount").show()
```

Less **data transfer**, faster processing.

## Adaptive Query Execution (AQE)

- Spark 3.x feature. Dynamically **optimizes query plan** at runtime.
- Optimizations:
  1. **Dynamic partition coalescing** → small files merged.
  2. **Skew join optimization** → automatically splits skewed keys.
  3. **Optimized shuffle join selection** → chooses broadcast vs sort-merge.

```
spark.conf.set("spark.sql.adaptive.enabled", True)
```

## Monitoring with Spark UI

- Spark UI shows:
  - **Stages & tasks**
  - **Shuffle read/write**
  - **Skewed partitions**
  - **Executor memory usage**
- Access: <http://<driver-node>:4040>

## Cost-Based Optimizer (CBO) Hints

- Spark can **optimize queries using table statistics**.
- **Hints** guide optimizer:

```
# Broadcast hint
df_large.join(df_small.hint("broadcast"), "customer_id").show()

# Shuffle hash hint
df_large.join(df_small.hint("shuffle_hash"), "customer_id").show()
```

- CBO uses **data size, column stats** to generate an **efficient execution plan**.

## Summary Table

Optimization Technique	Purpose / Benefit
<b>Caching &amp; Persistence</b>	Reuse DataFrames without recomputation
<b>Broadcast Joins</b>	Reduce shuffle for small-large joins
<b>Partition Pruning</b>	Read only relevant partitions
<b>Bucketing</b>	Optimize joins/aggregations on bucketed columns
<b>Handling Skew</b>	Avoid bottlenecks from uneven key distribution
<b>Predicate Pushdown</b>	Filter at data source, reduce I/O
<b>Column Pruning</b>	Read only required columns
<b>AQE</b>	Dynamically optimize query at runtime
<b>Spark UI</b>	Monitor jobs, stages, tasks, memory, skew
<b>CBO &amp; Hints</b>	Guide optimizer for efficient query plan

# Spark SQL – Time Series & Date Processing

Working with **dates**, **timestamps**, and **time series data** is common in analytics. Spark SQL provides **built-in functions** and **window operations** to handle time-based data efficiently.

## Date and Timestamp Functions

Spark SQL supports multiple **date and timestamp functions**:

Function	Description	Example
<code>current_date()</code>	Returns current date	2025-08-30
<code>current_timestamp()</code>	Returns current timestamp	2025-08-30 21:30:45
<code>date_add(col, n)</code>	Add n days to a date	<code>date_add("2025-08-01", 10) → 2025-08-11</code>
<code>date_sub(col, n)</code>	Subtract n days	<code>date_sub("2025-08-11", 5) → 2025-08-06</code>
<code>datediff(end, start)</code>	Difference in days between two dates	<code>datediff("2025-08-10", "2025-08-01") → 9</code>
<code>to_date(col)</code>	Converts string/timestamp to date	<code>to_date("2025-08-30 21:30:45") → 2025-08-30</code>
<code>to_timestamp(col)</code>	Converts string/date to timestamp	<code>to_timestamp("2025-08-30 21:30:45")</code>
<code>from_unixtime(col)</code>	Converts Unix epoch seconds to timestamp	<code>from_unixtime(1693375800) → 2025-08-30 21:30:00</code>

## Handling Time Zones

- Spark SQL **supports time zones** for timestamps.
- Use `spark.conf` to set session time zone:

```
spark.conf.set("spark.sql.session.timeZone", "UTC")
```

- Converting timestamps:

```
from pyspark.sql.functions import to_utc_timestamp, to_local_timestamp
```

```
df.withColumn("utc_time", to_utc_timestamp("order_ts", "Asia/Kolkata")) \
  .withColumn("local_time", to_local_timestamp("utc_time", "Asia/Kolkata")) \
  .show()
```

 Ensures accurate **time series calculations across regions**.

## Extracting Date Parts

- Common for **grouping, filtering, and aggregation:**

Function	Description
<b>year(col)</b>	Extract year
<b>month(col)</b>	Extract month
<b>day(col)</b>	Extract day
<b>hour(col)</b>	Extract hour
<b>minute(col)</b>	Extract minute
<b>second(col)</b>	Extract second

```
from pyspark.sql.functions import year, month, dayofmonth, hour

df.withColumn("year", year("order_date")) \
    .withColumn("month", month("order_date")) \
    .withColumn("day", dayofmonth("order_date")) \
    .show()
```

### Output

```
+-----+-----+-----+-----+-----+
|order_id|customer_id|amount|order_date|year|month|day|
+-----+-----+-----+-----+-----+
|      1|     Alice|   100|2025-08-01|2025|     8|   1|
|      2|      Bob|   200|2025-08-02|2025|     8|   2|
|      3|     Alice|   150|2025-08-03|2025|     8|   3|
|      4|      Bob|    50|2025-08-04|2025|     8|   4|
|      5| Charlie|   300|2025-08-05|2025|     8|   5|
+-----+-----+-----+-----+-----+
```

## Window Aggregations on Time Series

- Useful for **running totals, moving averages, ranking within time periods.**
- Use **window functions** with **time-based partitioning**.

```
from pyspark.sql.window import Window
from pyspark.sql.functions import sum, col

window_spec =
Window.partitionBy("customer_id").orderBy("order_date").rowsBetween(Window.unboundedPreceding, Window.currentRow)

df.withColumn("running_total", sum("amount").over(window_spec)) \
    .show()
```

## Output:

order_id	customer_id	amount	order_date	running_total
1	Alice	100	2025-08-01	100
3	Alice	150	2025-08-03	250
2	Bob	200	2025-08-02	200
4	Bob	50	2025-08-04	250
5	Charlie	300	2025-08-05	300

- `rowsBetween(Window.unboundedPreceding, Window.currentRow)` → cumulative sum up to the current row.

## Spark SQL – Aggregations and Grouping

Aggregation and grouping are essential for **summarizing data**, **calculating totals**, and performing **analytics**. Spark SQL provides **flexible APIs** for both **DataFrame operations** and **SQL queries**.

### Basic Aggregations with `groupBy` and `agg`

- `groupBy` → groups rows based on one or more columns.
- `agg` → applies aggregation functions like sum, avg, count, max, min.

```
from pyspark.sql.functions import sum, avg, count, max, min

df.groupBy("customer_id") \
  .agg(
    sum("amount").alias("total_amount"),
    avg("amount").alias("avg_amount"),
    count("order_id").alias("num_orders"),
    max("amount").alias("max_amount"),
    min("amount").alias("min_amount")
  ).show()
```

## Output:

customer_id	total_amount	avg_amount	num_orders	max_amount	min_amount
Alice	250	125.0	2	150	100
Bob	250	125.0	2	200	50
Charlie	300	300.0	1	300	300

## Multi-Column Grouping

- Group by **more than one column** for detailed summaries.

```
df.groupBy("customer_id", "order_date") \  
    .agg(sum("amount").alias("daily_total")) \  
    .show()
```

Output:

customer_id	order_date	daily_total
Alice	2025-08-01	100
Alice	2025-08-03	150
Bob	2025-08-02	200
Bob	2025-08-04	50
Charlie	2025-08-05	300

## Rollups and Cubes

- **Rollup** → hierarchical aggregation (e.g., subtotal + grand total).
- **Cube** → all combinations of grouping columns (multi-dimensional aggregation).

### 3.1 Rollup Example

```
df.rollup("customer_id", "order_date") \  
    .agg(sum("amount").alias("total_amount")) \  
    .orderBy("customer_id", "order_date") \  
    .show()
```

Output:

customer_id	order_date	total_amount
Alice	2025-08-01	100
Alice	2025-08-03	150
Alice	null	250
Bob	2025-08-02	200
Bob	2025-08-04	50
Bob	null	250
Charlie	2025-08-05	300
Charlie	null	300
null	null	800

- null in rollup → **subtotal / grand total**.

### 3.2 Cube Example

```
df.cube("customer_id", "order_date") \  
.agg(sum("amount").alias("total_amount")) \  
.orderBy("customer_id", "order_date") \  
.show()
```

- Cube gives **all combinations** of customer and date including totals.

### Pivot Tables

- **Pivot** turns **rows into columns**, useful for creating **cross-tab reports**.

```
df.groupBy("customer_id") \  
.pivot("order_date") \  
.sum("amount") \  
.show()
```

Output:

customer_id	2025-08-01	2025-08-02	2025-08-03	2025-08-04	2025-08-05
Alice	100	null	150	null	null
Bob	null	200	null	50	null
Charlie	null	null	null	null	300

-  Useful for **time-series summaries** or **cross-tab analytics**.

# Spark SQL Best Practices

Efficient Spark SQL usage can **dramatically improve performance**, reduce **execution time**, and save **cluster resources**.

## Use Columnar Formats (Parquet, ORC)

- Columnar formats **store data by column**, enabling **predicate pushdown** and **column pruning**.
- Preferred for **analytical workloads**.

```
# Save DataFrame as Parquet
df.write.parquet("orders.parquet")

# Read Parquet (only selected columns are read)
df_parquet = spark.read.parquet("orders.parquet").select("order_id", "amount")
```

Improves **I/O efficiency** and **query speed**.

## Partition Large Datasets for Parallelism

- **Partitioning** allows Spark to process **data in parallel** across nodes.
- Partition by **high-cardinality columns** used in filters or joins.

```
# Save DataFrame partitioned by year
df.write.partitionBy("year").parquet("orders_partitioned")
  • Use partition pruning to only read relevant partitions.
```

```
df_filtered = spark.read.parquet("orders_partitioned").filter("year = 2025")
```

Reduces **data scan** and **shuffle**.

## Avoid Wide Transformations When Possible

- **Wide transformations** (e.g., groupBy, join, distinct) require **shuffle** across nodes → expensive.
- Prefer **narrow transformations** (filter, select, map) where possible.

```
# Narrow transformation
df.filter(col("amount") > 100).select("order_id", "amount")
```

Reduces **network overhead** and **execution time**.

## Use AQE for Dynamic Optimizations

- **Adaptive Query Execution (AQE)** dynamically adjusts query plan at runtime.
- Automatically:
  - Handles skewed joins
  - Coalesces small partitions
  - Chooses optimal join strategy

```
spark.conf.set("spark.sql.adaptive.enabled", True)
```

Automatically improves **performance without manual tuning**.

## Minimize Shuffles

- **Shuffle** occurs when data moves across partitions (e.g., groupBy, join).
- Reduce shuffle by:
  - **Repartitioning wisely** (repartition or coalesce)
  - **Broadcast joins** for small tables
  - **Avoid unnecessary wide transformations**

```
df_large.join(broadcast(df_small), "customer_id")
```

Reduces **network I/O**, improves **execution speed**.

## Use Broadcast Joins for Small Tables

- Small tables (<~10MB) can be **broadcasted** to all executors.
- Avoids **expensive shuffle join**.

```
from pyspark.sql.functions import broadcast  
  
df_large.join(broadcast(df_small), "customer_id", "inner")
```

Efficient for **star-schema joins** in data warehouses.

## Use Caching Wisely

- **Cache/persist** DataFrames used **multiple times**.
- Avoid caching very large DataFrames unless necessary.

```
df.cache() # Stores in memory  
df.count() # Trigger cache materialization
```

- Remove cache when not needed:

```
df.unpersist()
```

Saves **computation** for repeated queries.

## Summary Table of Best Practices

Practice	Benefit
<b>Columnar formats (Parquet/ORC)</b>	Efficient I/O, predicate pushdown, column pruning
<b>Partitioning</b>	Parallelism, reduced scan, faster queries
<b>Avoid wide transformations</b>	Minimize shuffles, reduce execution time
<b>Use AQE</b>	Automatic runtime optimization
<b>Minimize shuffles</b>	Less network I/O, faster joins and aggregations
<b>Broadcast joins for small tables</b>	Efficient joins without shuffle
<b>Cache wisely</b>	Save computation for repeated queries

# Hive

## Introduction to Apache Hive

### What is Hive?

Apache Hive is a **data warehouse tool** built on top of Hadoop.

- It lets you query and analyze **large datasets** stored in HDFS (Hadoop Distributed File System) using a language similar to SQL, called **HQL (Hive Query Language)**.
- Hive translates HQL queries into **MapReduce, Tez, or Spark jobs**, so you don't need to write low-level code.[Otherwise needed to write java codes for map reduce]
- It is designed mainly for **batch processing** of structured and semi-structured data.

👉 In short: Hive = **SQL for Big Data**.

### History and Evolution

- **2007** → Created by **Facebook** to handle massive log data stored in Hadoop.
- **2008** → Contributed to the Apache Hadoop ecosystem as an open-source project.
- **2010** → Became a **Top-Level Project in Apache**.
- **Now** → Hive supports multiple execution engines (MapReduce, Tez, Spark) and is widely used in big data analytics and ETL.

### When to Use Hive?

#### ✓ Best for (Batch Use Cases)

- Data summarization, analytics, and reporting.
- ETL (Extract, Transform, Load) pipelines.
- Processing large log files, clickstream data, and transactional data.
- Building data warehouses on Hadoop.

#### ✗ Not ideal for (Real-Time Use Cases)

- Low-latency queries.
- Real-time dashboards.
- Use cases needing millisecond responses (→ better with **HBase, Presto, or Spark SQL**).

## Hive vs RDBMS (Differences)

Feature	Hive	RDBMS
Query Language	HiveQL (similar to SQL)	SQL
Underlying Storage	HDFS (Hadoop)	Traditional DB files (disk-based)
Execution Engine	MapReduce / Tez / Spark (batch)	Native query engine (row-based)
Speed	Slow (batch, high latency)	Fast (low latency)
Use Case	Data warehousing, analytics, ETL	OLTP (transactions), OLAP (reporting)
Transactions	Limited ACID (insert/update/delete supported since Hive 0.14)	Full ACID
Schema	Schema-on-Read (applied when querying)	Schema-on-Write (strict schema enforcement)

👉 Key Idea: Hive is **not a replacement** for RDBMS. It's built for **scalability over speed**.

## Hive Architecture

Hive sits **on top of Hadoop** and provides a SQL-like interface.

When you run a query in Hive

### 1. Hive Driver

- Accepts the query (HQL).
- Checks syntax and passes it to the compiler.
- Manages the lifecycle of the query.

### 2. Compiler

- Parses the query.
- Generates a logical plan.
- Optimizes the query.
- Converts it into execution plans (DAGs of MapReduce/Tez/Spark jobs).

### 3. Metastore

- Stores metadata about Hive tables (schema, columns, data types, partition info).
- Uses RDBMS (like MySQL, Derby, or Postgres) internally for metadata.
- Essential for query planning and optimization.

#### 4. Execution Engine

- Executes the compiled query plan.
- Engines supported:
  - **MapReduce** → Traditional, fault-tolerant, but slow.
  - **Tez** → DAG-based, faster than MapReduce.
  - **Spark** → In-memory, fastest option for Hive queries.

#### Simplified Flow of a Hive Query

1. User runs:
2. SELECT customer\_id, COUNT(\*)
3. FROM sales
4. WHERE country = 'US'
5. GROUP BY customer\_id;
6. Query goes to **Driver** → **Compiler** → **Metastore** → **Execution Engine**.
7. Execution Engine runs the job (MapReduce/Tez/Spark).
8. Results returned to the user.

## Hive vs Spark

- **Hive** → SQL-like data warehouse on top of Hadoop (HDFS). Converts HiveQL → MapReduce/Tez/Spark jobs. Best for **batch analytics**.
- **Spark** → General-purpose **distributed computing engine**. Provides APIs in **Scala, Python, Java, R**. Used for **batch, streaming, ML, and graph processing**.

## Architecture

Feature	Hive	Spark
Type	Data Warehouse on Hadoop	Distributed Data Processing Engine
Query Language	HiveQL (SQL-like)	Spark SQL (SQL) + APIs (RDD, DataFrame, Dataset)
Underlying Storage	HDFS	Works with HDFS, S3, HBase, Cassandra, etc.
Execution Engine	MapReduce, Tez, or Spark	Spark Core (in-memory execution)
Processing	Disk-based (slower)	In-memory (faster)

## Performance

- **Hive:**
  - Traditionally runs on **MapReduce** → **disk I/O heavy** → **slow**.
  - Tez and Spark engines improve speed, but still mostly **batch-oriented**.
- **Spark:**
  - **In-memory computing** makes it **10x-100x faster** than MapReduce-based Hive.
  - Optimized with **Catalyst Optimizer** and **Tungsten execution engine**.
  - Supports **interactive queries + real-time streaming**.

## Use Cases

Use Case	Hive	Spark
ETL (Extract-Transform-Load)	<input checked="" type="checkbox"/> Yes (batch pipelines)	<input checked="" type="checkbox"/> Yes (faster, scalable)
Data Warehousing	<input checked="" type="checkbox"/> Strong	<input checked="" type="checkbox"/> Strong
Real-time Processing	<input checked="" type="checkbox"/> No	<input checked="" type="checkbox"/> Yes (Spark Streaming, Structured Streaming)
Machine Learning	<input checked="" type="checkbox"/> Limited	<input checked="" type="checkbox"/> Yes (MLlib)
Graph Processing	<input checked="" type="checkbox"/> No	<input checked="" type="checkbox"/> Yes (GraphX)
Ad-hoc Interactive Queries	Limited	Excellent

## Latency

- **Hive** → High latency (minutes to hours for big jobs).
- **Spark** → Low latency (interactive queries in seconds).

## Ecosystem

- **Hive** integrates well with Hadoop ecosystem (Pig, HBase, Oozie, Sqoop).
- **Spark** has its own ecosystem:
  - **Spark SQL** (structured data)
  - **Spark Streaming** (real-time data)
  - **MLlib** (machine learning)
  - **GraphX** (graph computation)

## Example

### Hive Query (ETL)

```
INSERT OVERWRITE TABLE sales_summary
SELECT country, COUNT(*) as total_orders
FROM orders
GROUP BY country;
```

### Spark Equivalent (Python)

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("SalesSummary").getOrCreate()
orders = spark.read.csv("orders.csv", header=True, inferSchema=True)
summary = orders.groupBy("country").count()
summary.write.mode("overwrite").saveAsTable("sales_summary")
```

**Summary Table**

Feature	Hive	Spark
<b>Processing Type</b>	Batch (slow)	Batch + Streaming (fast)
<b>Execution Engine</b>	MapReduce/Tez/Spark	Spark Core (in-memory)
<b>Latency</b>	High	Low
<b>Real-time</b>	✗ No	✓ Yes
<b>Machine Learning</b>	✗ Limited	✓ MLLib
<b>Ease of Use</b>	Easy (SQL-like)	Medium (APIs + SQL)
<b>Best For</b>	Data Warehousing, ETL, Large Batch Jobs	Real-time analytics, ML, fast ETL, interactive queries

# How Hive Makes Big Data Processing Easier

## The Challenge of Big Data

Before Hive

- Data in **HDFS** (Hadoop Distributed File System) is **raw** → logs, CSV, JSON, Parquet, ORC.
- To analyze it, you had to write **complex Java MapReduce programs** (hundreds of lines).
- Not beginner-friendly, not fast for analysts.

👉 **Problem:** Analysts knew SQL, not Java.

## The Solution: Hive

- Hive provides a **SQL-like interface (HiveQL)** on top of Hadoop.
- You can query **petabytes of data** in HDFS just like SQL.
- Hive automatically translates SQL → **MapReduce/Tez/Spark jobs** in the backend.

✓ **You don't need to write MapReduce manually.**

✓ **Familiar SQL syntax → faster adoption.**

## Hive Architecture (Simplified Flow)

1. User submits **HiveQL query**
2. **Driver + Compiler** → Converts query into execution plan
3. **Execution Engine (MR/Tez/Spark)** → Runs distributed job
4. Reads/Writes from **HDFS**
5. **Metastore** → Stores metadata (tables, partitions, schema)

## Example – Without Hive (using raw MapReduce in Java)

Suppose you want to **count the number of orders per country** from a huge dataset (orders.csv).

👉 **In MapReduce (Java)** = 200+ lines of boilerplate code:

- Mapper → read each line, emit (country, 1)
- Reducer → aggregate counts
- Handle HDFS input/output

This is **hard for analysts**.

## Example – With Hive

👉 Same problem, but with **Hive SQL**.

### Step 1: Load data into Hive

```
-- Create table
CREATE TABLE orders (
    order_id STRING,
    customer_id STRING,
    country STRING,
    amount DOUBLE
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;

-- Load data from HDFS into Hive
LOAD DATA INPATH '/data/orders.csv' INTO TABLE orders;
```

### Step 2: Run query (Big Data analytics made easy 🎯)

```
-- Count orders by country
SELECT country, COUNT(*) as total_orders
FROM orders
GROUP BY country;
```

Hive automatically:

- Parses SQL → execution plan
- Runs it as **MapReduce (or Tez/Spark)** jobs
- Outputs aggregated results

## Behind the Scenes

That **single SQL query** above in Hive:

- Creates **Map tasks** → scan data, map each row to (country, 1)
- Creates **Reduce tasks** → aggregate counts per country
- Writes result back to HDFS

👉 Without Hive: you'd write 200+ lines of Java.

👉 With Hive: **just 3 lines of SQL**.

## Common Questions & Misconceptions about Hive

“Hive is a database just like MySQL or PostgreSQL.”

- ◆ **Reality:** Hive is NOT a database system like MySQL/Postgres. It's a **data warehouse framework** built on top of Hadoop.

### Comparison: Hive vs RDBMS

Feature	Hive (Data Warehouse)	MySQL/Postgres (RDBMS)
<b>Storage</b>	Data stored in <b>HDFS</b> (Hadoop Distributed File System)	Data stored in <b>disk files (local storage or SAN/NAS)</b>
<b>Query Language</b>	HiveQL (SQL-like, but limited to analytical queries)	SQL (full, transactional + analytical)
<b>Schema</b>	<b>Schema on Read</b> (schema applied when querying)	<b>Schema on Write</b> (schema enforced before inserting data)
<b>Query Processing</b>	Queries converted into <b>MapReduce/Tez/Spark jobs</b> → batch execution	Queries executed directly by database engine (real-time, row-level)
<b>Transactions</b>	Designed for <b>batch-level operations</b> , limited ACID (works only at partition or batch level)	Full <b>ACID transactions</b> , supports row-level insert/update/delete
<b>Speed</b>	Optimized for <b>scanning large datasets (petabytes)</b> ; slower latency (seconds/minutes)	Optimized for <b>row-level fast transactions</b> ; low latency (milliseconds)
<b>Use Case</b>	<b>Big Data analytics</b> (reports, aggregations, batch jobs)	<b>OLTP</b> (Online Transaction Processing) – small updates, fast queries

👉 **Conclusion:** Hive = Analytics on Big Data; MySQL/Postgres = Fast real-time OLTP.

**“Hive can perform row-level transactions like SQL databases.”**

- ◆ **Reality:** ❌ Hive is **NOT meant for row-level transactions.**
  - Hive was designed for **batch processing**, not OLTP.
  - Hive supports **ACID transactions** only for certain storage formats (like ORC), but:
    - Updates/deletes happen at **partition or file level**, not efficient for single rows.
    - It **rewrites the whole partition/file** internally when you update.

👉 So, if you try:

```
UPDATE orders SET amount = 100 WHERE order_id = '123';
```

- Hive will **not just update one row** — it will **rewrite the entire file/partition** where that row lives.
- This is inefficient for high-frequency row updates.

✓ Best for: **batch inserts and aggregations**

✗ Not good for: **frequent row-level updates/deletes**

**“Hive works only on HDFS.”**

- ◆ **Reality:** Mostly true, but Hive can also connect to other storage systems via connectors.
  - By default, Hive works on **HDFS (Hadoop Distributed File System)**.
  - But Hive can also query data from:
    - **Amazon S3**
    - **Azure Data Lake**
    - **Google Cloud Storage**
    - **HBase (NoSQL)**
    - Any storage supported by Hadoop InputFormat.

👉 Hive is **storage-agnostic** as long as the data can be presented via Hadoop’s filesystem API.

**“Hive is a real-time query engine like MySQL.”**

◆ Reality: ❌ Hive is **NOT real-time**.

- Hive queries run as **batch jobs** (MapReduce/Tez/Spark).
- They take **seconds to minutes**, depending on dataset size.
- Not suitable for **real-time dashboards or OLTP queries**.

👉 For **real-time Big Data queries**, you'd use:

- **Apache Impala**
- **Apache Drill**
- **Presto/Trino**
- **Spark SQL (with caching)**

**“Hive stores data itself.”**

◆ Reality: ❌ Hive does **NOT store data**.

- Hive only stores **metadata** (table schema, partitions, locations) in the **Metastore (usually MySQL/Postgres/Derby)**.
- The actual data lives in **HDFS (or cloud storage like S3)**.

👉 Think of Hive as a **SQL translator** for Hadoop.

**“Hive is fast for all queries.”**

◆ Reality: ❌ Hive is **not optimized for small queries**.

- Hive shines when scanning **huge datasets (GB → PB)**.
- For small datasets, RDBMS like MySQL/Postgres/SQLite are much faster.
- Hive query overhead = launching a distributed job → slow startup time.

👉 Example:

- **SELECT COUNT(\*) FROM orders;** on 1GB dataset → MySQL finishes in seconds, Hive may take 1+ minute.
- But on **10TB dataset**, MySQL will choke, while Hive will still scale.

**“Hive can replace OLTP databases.”**

- ◆ **Reality:** ❌ Hive is **NOT a replacement** for OLTP systems like MySQL/Postgres.
  - Hive = **OLAP (Online Analytical Processing)**
  - MySQL/Postgres = **OLTP (Online Transaction Processing)**

👉 Correct usage:

- Use Hive for **reporting, aggregations, data warehousing.**
- Use MySQL/Postgres for **applications, transactions, user-facing systems.**

**“Hive only works with MapReduce.”**

- ◆ **Reality:** ❌ Hive started with **MapReduce**, but now it supports:
  - **Tez** (faster DAG execution engine)
  - **Spark** (in-memory, faster queries)

👉 Modern Hive is often run on **Spark SQL** backend for speed.

# Connecting to Hive – CLI & Beeline

## Hive CLI (Old way – hive command)

Earlier, Hive was accessed using the **Hive command-line interface (CLI)**.

It's still available, but now **deprecated** in favor of **Beeline** (for better security & JDBC support).

### Steps to connect via Hive CLI

```
# Start Hive CLI  
hive
```

👉 Once inside, you can run HiveQL commands:

```
SHOW DATABASES;  
USE default;  
SHOW TABLES;  
SELECT * FROM orders LIMIT 10;
```

📌 **Limitation:** Hive CLI connects directly to HiveServer2's backend processes, not recommended for production.

## Hive Beeline (Modern & Recommended)

Beeline is a **JDBC client** that connects to **HiveServer2**.

It supports **multi-user connections**, authentication, and is production-ready.

### Start HiveServer2 (must be running first)

On the server node:

```
hiveserver2
```

This launches the HiveServer2 service, which listens for Beeline/JDBC clients.

### Connecting via Beeline

```
# Open Beeline shell  
beeline  
  
# Connect to HiveServer2  
!connect jdbc:hive2://localhost:10000  
  ◇ If authentication is enabled:  
!connect jdbc:hive2://localhost:10000 username password
```

## Example Session in Beeline

```
SHOW DATABASES;  
USE retail_db;  
SHOW TABLES;  
SELECT COUNT(*) FROM orders;
```

👉 Output looks similar to MySQL/Postgres CLI, but queries are executed on top of **HDFS + MapReduce/Tez/Spark**.

## Difference Between Hive CLI & Beeline

Feature	Hive CLI  (old)	Beeline  (recommended)
Connection type	Direct to Hive backend	JDBC via HiveServer2
Security	Weak (no auth)	Supports LDAP/Kerberos
Multi-user support		
Production use		
Future support	Deprecated	Actively supported

## Hive with CSV File

### Prepare a Sample CSV File

Let's say we have `orders.csv` like this:

```
order_id,order_date,customer_id,order_status  
1,2023-01-01,101,COMPLETE  
2,2023-01-02,102,PENDING  
3,2023-01-03,103,COMPLETE  
4,2023-01-04,104,CANCELLED  
5,2023-01-05,105,PROCESSING
```

Save it to your **local system** first.

### Upload CSV to HDFS

Hive works on top of **HDFS**, so we need to put the file there:

```
# Make directory in HDFS  
hdfs dfs -mkdir -p /user/hive/warehouse/orders_data  
  
# Upload the CSV file  
hdfs dfs -put orders.csv /user/hive/warehouse/orders_data/
```

## Start HiveServer2

On your server:

```
hiveserver2
```

Keep it running in the background.

## Connect with Beeline

```
beeline
!connect jdbc:hive2://localhost:10000
(Use username/password if enabled.)
```

## Create a Database (Optional)

```
CREATE DATABASE retail_db;
USE retail_db;
```

## Create a Hive Table for the CSV

We define table schema mapping to CSV columns.

```
CREATE TABLE orders (
    order_id INT,
    order_date STRING,
    customer_id INT,
    order_status STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;
```

👉 Explanation:

- ROW FORMAT DELIMITED → tells Hive it's CSV-like.
- FIELDS TERMINATED BY ',' → columns are separated by commas.
- STORED AS TEXTFILE → stored in plain text format.

## Load CSV Data into Hive

```
LOAD DATA INPATH '/user/hive/warehouse/orders_data/orders.csv'
INTO TABLE orders;
```

📌 **Note:** This doesn't copy the file, it just moves it inside Hive's warehouse directory.

## Run Queries in Hive

Now you can query like SQL:

```
-- Check data
SELECT * FROM orders LIMIT 5;

-- Count total rows
SELECT COUNT(*) FROM orders;

-- Find all COMPLETE orders
SELECT * FROM orders WHERE order_status = 'COMPLETE';

-- Number of orders per status
SELECT order_status, COUNT(*)
FROM orders
GROUP BY order_status;
```

## Output Example

order_id	order_date	customer_id	order_status
1	2023-01-01	101	COMPLETE
2	2023-01-02	102	PENDING
3	2023-01-03	103	COMPLETE
...			

## Optimizations (Optional)

- Store data as **Parquet/ORC** instead of CSV for better performance

```
CREATE TABLE orders_parquet (
    order_id INT,
    order_date STRING,
    customer_id INT,
    order_status STRING
)
STORED AS PARQUET;
```

- Insert data into Parquet table:

```
INSERT INTO orders_parquet SELECT * FROM orders;
```

👉 Queries on orders\_parquet will be much faster.

# Hive Metadata

## What is Hive Metadata?

- **Definition**  
Hive metadata is information *about the tables and databases* in Hive — not the actual data, but the **schema, structure, and statistics**.
- Stored in the **Hive Metastore** (usually backed by an RDBMS like MySQL, Derby, or PostgreSQL).
- **Examples of metadata Hive stores:**
  - Databases (names, locations in HDFS)
  - Tables (columns, datatypes, partition info, bucket info)
  - Views
  - Indexes
  - Storage format (TEXTFILE, PARQUET, ORC, AVRO...)
  - File location in HDFS
  - Table statistics (number of rows, file size, partitions count)

👉 Metadata allows Hive to translate SQL-like queries (HiveQL) into execution plans on Hadoop/Spark.

## Why is Metadata Important?

1. **Separation of schema and data** → Data is stored in HDFS, schema/definitions in Metastore.
2. **SQL-like experience** → Hive knows how to interpret raw HDFS files.
3. **Query optimization** → Hive uses metadata (like partition info) to skip unnecessary data.
4. **Integration** → Other tools (Spark, Presto, Impala, etc.) can also use Hive Metastore.

## Hive Metastore

- **Embedded Mode** → Uses Derby (single-user, local testing).
- **Local/Remote Mode** → Uses external DB like MySQL/Postgres, supports multi-user clusters.

## Components:

- **Metastore service** → API layer to fetch metadata.
- **Underlying RDBMS** → Actually stores the schema.
- **Hive Clients** → Beeline, Hive CLI, Spark SQL, etc.

## Accessing Hive Metadata

### (a) From Hive Queries

You can query metadata using **DESCRIBE** commands:

```
-- List databases
SHOW DATABASES;

-- List tables in a database
SHOW TABLES IN retail_db;

-- Describe schema of a table
DESCRIBE orders;

-- Extended metadata (location, input/output format, SerDe info)
DESCRIBE EXTENDED orders;

-- More detailed info including statistics
DESCRIBE FORMATTED orders;

-- Partition info
SHOW PARTITIONS orders;

-- Show table properties
SHOW TBLPROPERTIES orders;
```

### (b) From Beeline / Hive CLI

```
!tables      -- show available tables
!columns orders -- list columns of "orders" table
```

### (c) From Metastore Database (Direct SQL)

If using MySQL/Postgres as the metastore, you can directly query system tables:

```
-- In MySQL metastore
USE hive_metastore;

-- List all Hive tables
SELECT TBL_NAME, DB_ID FROM TBLS;

-- Find columns of a table
SELECT COLUMN_NAME, TYPE_NAME FROM COLUMNS_V2 WHERE CD_ID=123;

-- Check database metadata
SELECT * FROM DBS;
```

👉 Typically, Hadoop admins do this for debugging.

## (d) From Spark / External Tools

Hive Metastore is shared — Spark can access it too:

```
# Spark accessing Hive metadata
spark.sql("SHOW DATABASES").show()
spark.sql("SHOW TABLES IN retail_db").show()
spark.sql("DESCRIBE FORMATTED orders").show(truncate=False)
```

### ◆ 5. Example: Metadata in Action

Suppose we created:

```
CREATE TABLE orders (
    order_id INT,
    order_date STRING,
    customer_id INT,
    order_status STRING
)
PARTITIONED BY (year STRING, month STRING)
STORED AS PARQUET;
```

- **Data** goes to → HDFS: /user/hive/warehouse/retail\_db.db/orders/
- **Metadata** goes to → Metastore DB:
  - orders table name stored in TBLS
  - Columns in COLUMNS\_V2
  - Partition info in PARTITIONS
  - File format info in SDS (storage descriptors)

When you query:

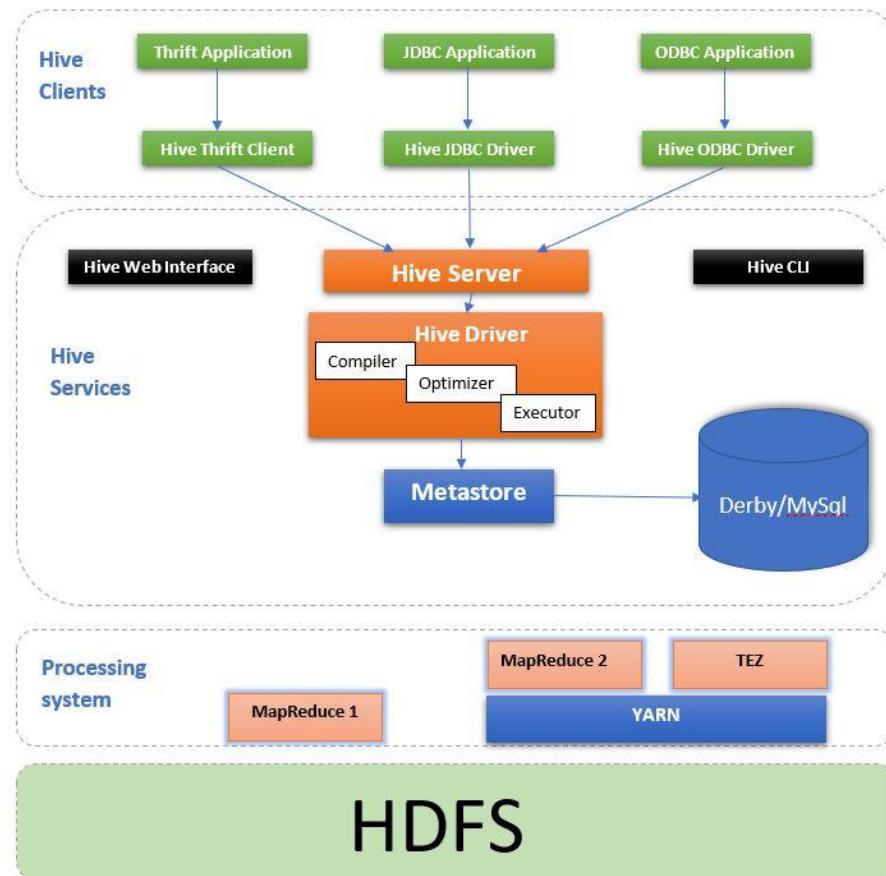
```
SELECT COUNT(*) FROM orders WHERE year='2023' AND month='01';
```

Hive checks **metadata first** to see which partition to read → avoids scanning entire dataset.

## Best Practices for Metadata

- Use **external Metastore (MySQL/Postgres)** for production (multi-user).
- Enable **automatic statistics collection** (ANALYZE TABLE ... COMPUTE STATISTICS).
- Regularly **back up Metastore DB** (critical for cluster recovery).
- Be mindful of **partition explosion** → too many partitions will overload metadata.

# Hive architecture



## Hive Clients in Hive Architecture

Hive has a layered architecture. The **top layer** is **Hive Clients**, which are the entry points for users/developers to interact with Hive.

### What are Hive Clients?

- Hive clients are the **interfaces/tools** through which users **submit queries** to Hive.
- They **send HiveQL (SQL-like queries)** to Hive → Hive Compiler → Metastore → Execution Engine → Hadoop/Spark.
- They do **not process data directly**; they only **communicate** with HiveServer2.

### Types of Hive Clients

#### 1. Hive Command Line Interface (CLI) (Legacy)

Old interface (now mostly replaced by Beeline). Lets users run HiveQL queries directly from terminal.

Example:

Example:

```
hive
hive> SHOW DATABASES;
hive> SELECT * FROM employees LIMIT 5;
```

- Limitation → Only worked with embedded/local mode. Deprecated in new Hadoop distributions.

## Beeline (Recommended CLI)

- A JDBC client that connects to **HiveServer2**.
- Lightweight, secure, and supports **multi-user** access.
- Example:

```
beeline -u jdbc:hive2://localhost:10000 -n hadoop_user -p password
0: jdbc:hive2://localhost:10000> SHOW TABLES;
```

- Preferred over old Hive CLI.

## JDBC/ODBC Clients

- Used for **BI Tools** (Tableau, Power BI, Superset) to connect to Hive.
- JDBC = Java-based API.
- ODBC = Standard API for Windows/BI tools.
- Example JDBC string:

```
jdbc:hive2://hive-server:10000/default
```

## WebHCat (Templeton) – REST API Client

- Allows **programmatic access** to Hive using **REST APIs**.
- Useful for web apps, automation, or when tools can't use JDBC/ODBC.
- Example:

```
curl -s -d execute="SHOW TABLES" \
      -d user.name=hadoop_user \
      http://localhost:50111.templeton/v1/hive
```

## Programmatic Clients (Java, Python, etc.)

- Developers can connect to Hive programmatically using APIs:

### Java (JDBC Example)

```
Connection con = DriverManager.getConnection(
    "jdbc:hive2://localhost:10000/default", "hadoop_user", "password");
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM employees");
```

## Python (PyHive Example)

```
from pyhive import hive

conn = hive.Connection(host='localhost', port=10000, username='hadoop_user')
cursor = conn.cursor()
cursor.execute("SELECT * FROM employees LIMIT 5")
for row in cursor.fetchall():
    print(row)
```

## How Hive Clients Fit in Hive Architecture

### Flow

User → Hive Client (CLI/Beeline/JDBC/ODBC/API) → HiveServer2 → Driver → Compiler → Metastore → Execution Engine → Hadoop/Spark

- Hive clients = **entry gate**.
- HiveServer2 = **manager** that accepts client connections.
- Execution Engine (MR/Tez/Spark) = **worker** that runs queries on cluster.

## Hive Server & Hive Driver

### HiveServer (HS2 – HiveServer2)

#### What is HiveServer?

- HiveServer2 (HS2) is the **service layer** that accepts connections from **Hive Clients** (Beeline, JDBC, ODBC, BI tools, REST API).
- It is a **Thrift-based service** → allows remote clients to submit queries.
- Acts like a **gateway** between the user and Hive execution engine.

#### Functions of HiveServer2

1. **Multi-user support** → multiple clients can connect simultaneously.
2. **Authentication & Security** → supports Kerberos, LDAP, etc.
3. **Session Management** → handles queries from multiple users.
4. **Communication Layer** → provides JDBC/ODBC access for BI tools.

## Example

Start HiveServer2 in Hadoop cluster:

```
hiveserver2
```

Connect using Beeline:

```
beeline -u jdbc:hive2://localhost:10000 -n hadoop_user
```

## Without HiveServer2

- Hive CLI could only run in **local embedded mode** (single user).
- With HS2 → Hive becomes **multi-user and enterprise-ready**.

## Hive Driver

### What is Hive Driver?

- Hive Driver is part of Hive's **query processing engine**.
- It **manages query execution lifecycle**:
  1. Accepts the query from client.
  2. Sends it to **compiler**.
  3. Monitors query execution.
  4. Returns results back to client.

### Components inside Hive Driver

#### 1. Parser

- Checks **syntax** of HiveQL query.
- Example: If you write
- SELEC name FROM employees;

Parser throws an error (SELEC instead of SELECT).

#### 2. Semantic Analyzer

- Validates **tables, columns, and schema** using **Metastore**.
- Example: If table employees does not exist → error.

#### 3. Query Optimizer

- Rewrites the query for better performance.
- Example:
- SELECT \* FROM employees WHERE age > 30;

→ Optimizer may push the filter ( $age > 30$ ) down to the data source.

#### 4. Execution Plan Generator

- Converts HiveQL into **execution plan (DAG of tasks)** → submitted to **Execution Engine** (MapReduce/Tez/Spark).

#### Workflow of Hive Driver

 Suppose a user runs:

```
SELECT department, COUNT(*)  
FROM employees  
WHERE salary > 50000  
GROUP BY department;
```

 The flow is:

- Hive Client (Beeline)** sends query → HiveServer2.
- Hive Driver** receives query.
- Parser** → checks SQL syntax.
- Semantic Analyzer** → checks table employees in Metastore.
- Optimizer** → rewrites query for efficiency.
- Execution Plan Generator** → generates MapReduce/Tez/Spark tasks.
- Execution Engine** runs tasks on Hadoop cluster.
- Driver collects results** and sends them back to HiveServer2 → Client.

#### HiveServer vs Hive Driver

Feature	HiveServer2 (HS2)	Hive Driver
<b>Role</b>	Entry point for clients	Manages query lifecycle
<b>Location</b>	Runs as a service (separate process)	Internal to Hive execution engine
<b>Function</b>	Accepts queries via JDBC/ODBC/Beeline	Parses, analyzes, optimizes, and executes
<b>Focus</b>	Communication & sessions	Query processing & execution management

## Hive Clients, Hive Server, and Hive Driver

Hive is built as a **data warehouse system on Hadoop/Spark**. Its components work together to **accept queries, process them, and return results**.

### Hive Clients

#### Properties:

- **User-facing tools** → interface for users to run Hive queries (HiveQL).
- Provide **connectivity** to Hive Server (direct CLI, Beeline, JDBC, ODBC, or Web UI).
- Can run in **embedded/local mode** or connect to **HiveServer2** for distributed/multi-user mode.
- Support **query submission, metadata browsing, and data retrieval**.

#### Examples of Hive Clients:

1. **Hive CLI (deprecated)** – Directly talks to Driver (single-user).
2. **Beeline (recommended)** – Connects to HiveServer2 (multi-user).
3. **JDBC/ODBC clients** – BI tools like Tableau, Power BI, etc.
4. **Web-based clients** – Hue, Ambari, etc.

#### Behavior

- Clients **submit queries** written in HiveQL.
- They **do not process data** → they only act as a **gateway for queries**.
- If connected to HiveServer2, they rely on **Thrift/JDBC/ODBC protocols** for communication.

#### Example:

```
beeline -u jdbc:hive2://localhost:10000 -n hadoop_user
```

Here, Beeline is the **client** → sends queries to HiveServer2.

## 2. Hive Server (HiveServer2)

### Properties:

- Runs as a **long-running service**.
- Provides **multi-user, concurrent query execution**.
- Implements **authentication & security** (Kerberos, LDAP, PAM).
- Provides **session management** (each client gets its own session).
- Uses **Thrift/JDBC/ODBC APIs** → allows external applications (Java, Python, BI tools) to connect.

### Behavior:

- Accepts HiveQL queries from clients (Beeline, JDBC, ODBC).
- Passes queries to **Hive Driver** for execution.
- Handles **user sessions** and **resource isolation**.
- Ensures that multiple queries from different users can run at the same time.

### Example

```
hiveserver2
```

Starts HiveServer2 → then clients (Beeline/JDBC) connect to it.

## 3. Hive Driver

### Properties:

- Internal component of Hive execution engine.
- Responsible for **managing the query lifecycle**.
- Contains **Parser, Semantic Analyzer, Optimizer, Execution Plan Generator**.
- Acts as a **coordinator** between query compilation and execution.
- Not directly accessed by users → always invoked through HiveServer2 or Hive CLI.

### Behavior:

1. **Accepts query** from HiveServer2.
2. **Parses query** → checks syntax.
3. **Analyzes semantics** → checks schema and metadata from Metastore.
4. **Optimizes query** → rewrites query for efficiency.
5. **Generates execution plan** (MapReduce, Tez, or Spark jobs).
6. **Submits execution plan** to Execution Engine.

7. **Monitors query execution** and collects results.

8. **Sends results back** to HiveServer2 → client.

### Example Flow

```
SELECT department, COUNT(*)  
FROM employees  
WHERE salary > 50000  
GROUP BY department;
```

👉 Hive Driver ensures:

- Syntax is correct.
- Table & column exist.
- Filter (salary > 50000) is optimized.
- MapReduce/Tez/Spark jobs are generated and executed.

### Comparison Table

Component	Properties	Behavior
Hive Clients	Interfaces for users (CLI, Beeline, JDBC, ODBC, Web UI).	Submit HiveQL queries, fetch results. Don't process data.
Hive Server	Thrift/JDBC/ODBC service. Multi-user, session, authentication.	Accepts queries from clients, forwards to Driver, manages sessions.
Hive Driver	Core query manager (Parser, Analyzer, Optimizer, Plan Generator).	Validates, optimizes, and executes queries. Returns results to Hive Server → Clients.

# Hive Compiler and Hive Optimizer

Hive's execution flow:

Hive Client → HiveServer2 → Hive Driver → Compiler → Optimizer → Execution Engine  
(MapReduce/Tez/Spark) → HDFS

The **Compiler and Optimizer** are inside the **Hive Driver**. They transform HiveQL queries into an **efficient execution plan**.

## Hive Compiler

### What is Hive Compiler?

- The **compiler converts HiveQL queries into an execution plan** that can be run on Hadoop/Spark.
- It ensures that the query is **syntactically and semantically correct**.

### Components of Hive Compiler

#### 1. Parser

- Checks **syntax** of HiveQL query.
- Example:  
`SELEC * FROM orders;`  
→ Parser will throw an **error** (SELEC is invalid).

#### 2. Semantic Analyzer

- Checks **metadata correctness**.
- Confirms tables, columns, partitions exist by querying **Hive Metastore**.
- Checks **types** of columns for operations like SUM, COUNT.
- Example:  
`SELECT order_id, total FROM orders;`  
→ If total column doesn't exist, semantic analyzer will raise an error.

#### 3. Query Plan Generator

- Converts HiveQL into a **Directed Acyclic Graph (DAG)** of tasks.
- Each task represents **MapReduce/Tez/Spark jobs**.
- Example: GROUP BY → Map task to emit (key, value), Reduce task to aggregate.

## Behavior of Hive Compiler

- Validates query for **syntax & semantics**.
- Generates **logical query plan** (tasks + dependencies).
- Prepares the plan for the **optimizer** to improve performance.

## Hive Optimizer

### What is Hive Optimizer?

- Optimizer **rewrites and improves the query plan** for better performance.
- Minimizes **data scanned**, reduces **shuffle**, and improves **execution time**.

### Optimization Techniques in Hive

#### 1. Predicate Pushdown

- Filters are pushed to **scan phase** → read only necessary rows.
- Example:

```
SELECT * FROM orders WHERE order_status='COMPLETE';  
→ Only scans rows with order_status='COMPLETE'.
```

#### 2. Column Pruning

- Reads **only required columns** instead of full table.
- Example:

```
SELECT order_id, customer_id FROM orders;  
→ Only these two columns are read from HDFS.
```

#### 3. Partition Pruning

- Only relevant **partitions** are scanned.
  - Example:
- ```
SELECT * FROM orders WHERE year='2023';  
→ Hive scans only year=2023 partition.
```

#### 4. Join Optimization

- Determines **best join strategy**: MapJoin (Broadcast Join), Sort-Merge Join, etc.
- Small tables can be **broadcasted** to all mappers to avoid shuffles.

#### 5. Bucketing

- Helps **optimize joins and aggregations**.
- Hive knows data is pre-shuffled into buckets → can skip unnecessary scans.

## 6. Merge and Combine Steps

- Combines small map outputs before shuffle → reduces network traffic.

## 7. Adaptive Query Execution (AQE)

- Optimizer can **adjust execution plan dynamically** based on runtime statistics.
- Example: dynamically switch join strategy if table sizes are different than expected.

## Behavior of Hive Optimizer

- Takes the **logical plan** from Compiler.
- Applies **rewrite rules and heuristics** to reduce processing time.
- Outputs **optimized execution plan** ready for Execution Engine.

## Workflow Example

Query:

```
SELECT customer_id, COUNT(*)  
FROM orders  
WHERE year='2023' AND order_status='COMPLETE'  
GROUP BY customer_id;
```

### Hive Compiler

- Parses syntax
- Checks table orders, columns customer\_id, order\_status, year
- Generates logical DAG → Map + Reduce tasks

### Hive Optimizer

- Pushes filters → only year='2023' partitions + order\_status='COMPLETE' rows
- Column pruning → only customer\_id column read
- Combines small map outputs
- Generates optimized plan for Execution Engine

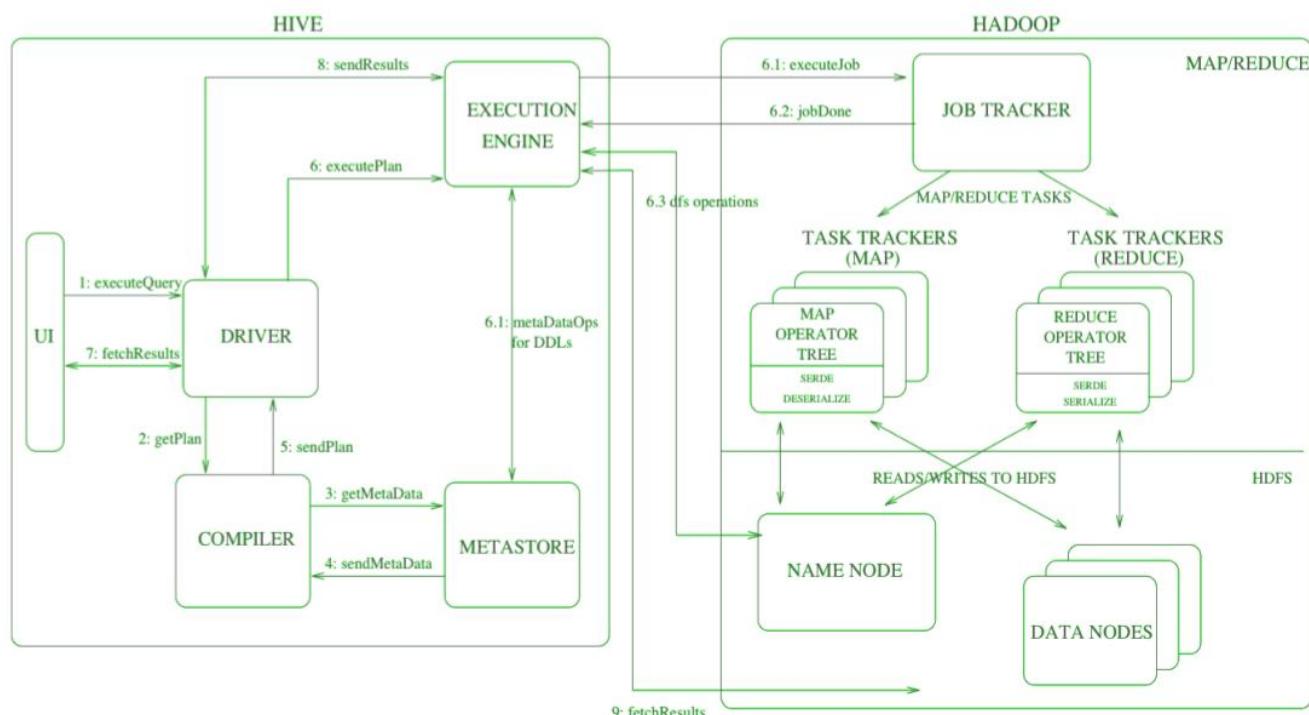
**Execution Engine** (Tez/MapReduce/Spark) → runs jobs efficiently on HDFS.

## Summary

| Component      | Role                                              | Key Behavior                                                         |
|----------------|---------------------------------------------------|----------------------------------------------------------------------|
| Hive Compiler  | Validates HiveQL and generates logical query plan | Syntax check, semantic check, DAG generation                         |
| Hive Optimizer | Improves query plan for efficiency                | Predicate pushdown, column/partition pruning, join optimization, AQE |
| Together       | Convert SQL → optimized execution plan            | Ensures queries run efficiently on Hadoop/Spark                      |

## Hive Query Flow

When you run a query in Hive (via Beeline, JDBC, or Hive CLI), it **doesn't immediately access HDFS**. Instead, it passes through **several layers** inside Hive before execution. Understanding this flow is critical for optimization, debugging, and designing big data workflows.



## User Submits Query via Hive Client

### Clients:

- **Beeline** (recommended CLI)
- Hive CLI (deprecated)
- JDBC/ODBC connections (BI tools, Python, Java apps)
- Web UI (Hue, Ambari)

### Example Query:

```
SELECT customer_id, COUNT(*)
FROM orders
WHERE year='2023' AND order_status='COMPLETE'
GROUP BY customer_id;
```

### What happens

- The client sends the HiveQL query to **HiveServer2** (HS2) via **JDBC/ODBC/Thrift protocol**.
- The client itself **does not process data**; it just passes the query and receives results.

## HiveServer2 Receives Query

### HiveServer2 Properties:

- Multi-user, session-based service.
- Manages authentication (Kerberos, LDAP).
- Forwards queries to **Hive Driver** for execution.

### Behavior:

- Creates a **session** for each user.
- Receives query text and session info.
- Forwards query to **Hive Driver**.

## Hive Driver – Query Lifecycle Manager

**Hive Driver** is the **core component** that manages the lifecycle of a HiveQL query.

### Steps:

1. **Accept query** from HiveServer2.
2. **Send to Compiler** → parse and validate.
3. **Monitor execution** → track progress of tasks.
4. **Return results** back to HiveServer2 → Client.

## Hive Compiler – Parse, Analyze, Generate Plan

### Compiler Responsibilities:

#### 1. Parser

- Checks **syntax correctness** of query.
- Example: SELEC \* FROM orders; → Syntax error.

#### 2. Semantic Analyzer

- Checks **tables, columns, data types** using **Hive Metastore**.
- Ensures all references exist and are valid.

#### 3. Logical Plan Generator

- Converts query into a **DAG (Directed Acyclic Graph)** of tasks.
- Tasks could be **MapReduce, Tez, or Spark jobs** depending on execution engine.

### Output:

- **Logical execution plan** (not yet optimized).

## Hive Optimizer – Improve Performance

### Optimizer Responsibilities:

- Applies **rules and techniques** to improve execution:
  1. **Predicate Pushdown** → filter early in scan phase.
  2. **Column Pruning** → read only required columns.
  3. **Partition Pruning** → scan only relevant partitions.
  4. **Join Optimizations** → broadcast small tables, choose join strategies.
  5. **Bucketing & Sorting** → reduce shuffle and improve joins.
  6. **Adaptive Query Execution (AQE)** → dynamically optimize based on runtime stats.

### Example Optimization

```
SELECT customer_id, COUNT(*) FROM orders
WHERE year='2023' AND order_status='COMPLETE'
GROUP BY customer_id;
```

- Only scans year=2023 partition → **reduces data read**.
- Only reads customer\_id column → **reduces IO**.

### Output

- **Optimized physical execution plan** ready for the execution engine.

## Execution Engine – Run on Hadoop/Spark

### Options:

- **MapReduce** (default in older Hive versions)
- **Tez** (faster DAG-based engine)
- **Spark** (in-memory execution, faster queries)

### Behavior:

- Executes tasks generated by compiler + optimizer.
- Reads data from **HDFS** (or other storage like S3, HBase).
- Performs **Map → Shuffle → Reduce** operations (or DAG tasks in Tez/Spark).
- Writes intermediate results and final output.

### Example Execution:

- Map phase → Reads relevant rows/columns → emits (customer\_id, 1)
- Reduce phase → Aggregates counts by customer\_id → outputs results.

## Hive Metastore Interaction

### During query processing:

- Compiler/Optimizer queries **Hive Metastore** for:
  - Table definitions
  - Column types
  - Partition locations
  - Storage format (TEXTFILE, ORC, PARQUET)
- Metastore **does not store data**, only metadata.

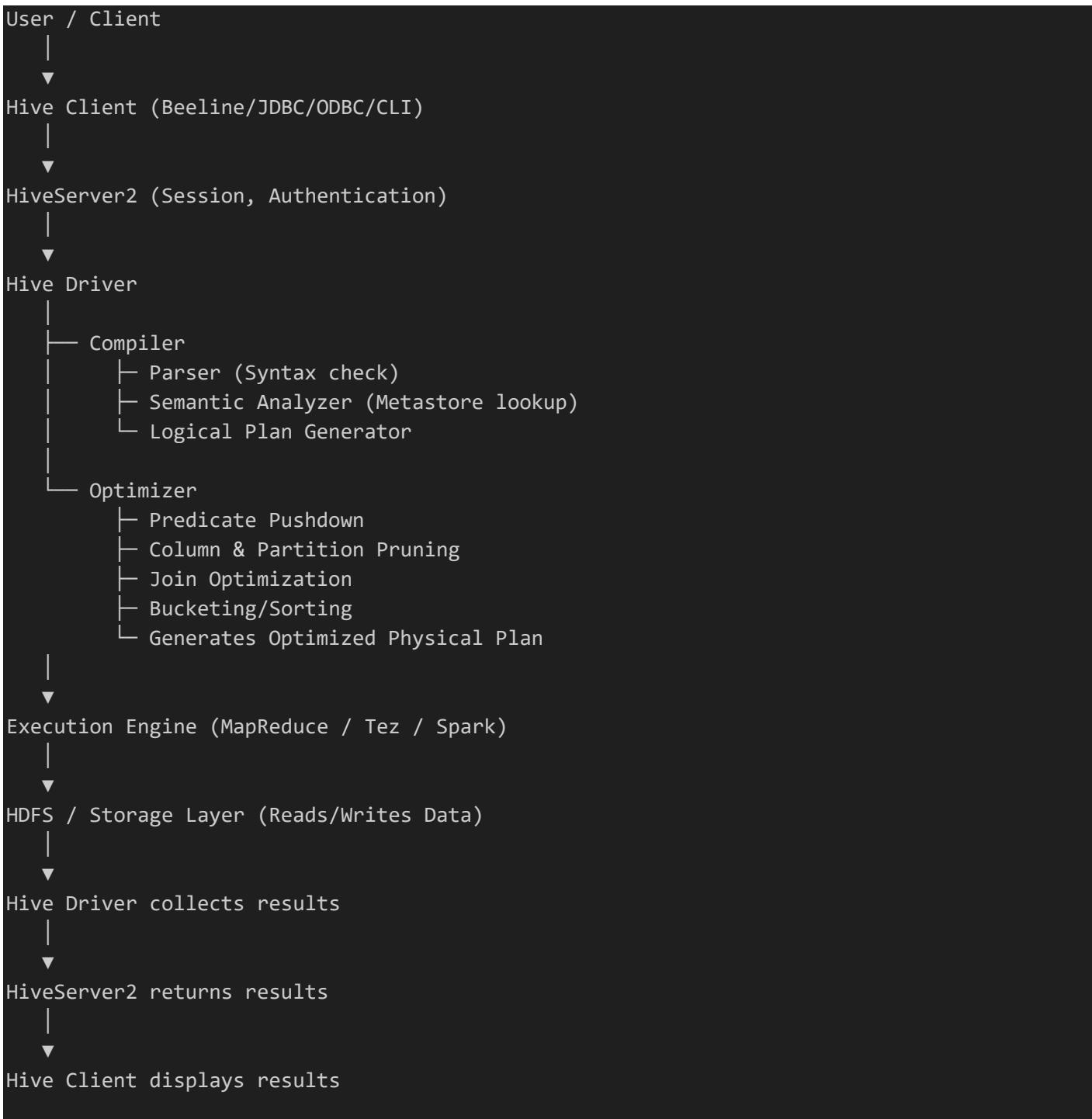
### Purpose:

- Helps optimizer do **partition pruning, bucketing, and schema validation**.

## Results Return to Client

- Execution Engine finishes tasks → **Driver collects results**.
- HiveServer2 sends results back to **Hive Client**.
- Client displays results (table view in Beeline, GUI in Hue, or programmatically in Python/Java).

## Complete Hive Query Flow Diagram



## Key Takeaways

1. **Client only submits query & receives results** → no data processing.
2. **HiveServer2** → multi-user gateway, manages sessions & security.
3. **Driver** → core query lifecycle manager.
4. **Compiler** → parses query, validates schema, generates logical DAG.
5. **Optimizer** → improves query plan (predicate pushdown, pruning, join optimization).
6. **Execution Engine** → actually runs tasks on HDFS/Spark/Tez.
7. **Metastore** → stores table metadata, partitions, schema info.

# Advanced Hive Topics

## Materialized Views in Hive

### ◆ What it is

- A **materialized view (MV)** is like a **cached query result** stored as a physical table.
- Instead of re-running expensive queries, Hive can use MVs to return results faster.

### ◆ Why useful?

- Saves time for repetitive queries (e.g., BI dashboards).
- Reduces recomputation cost on huge datasets.

### ◆ Example

```
-- Create a materialized view
CREATE MATERIALIZED VIEW mv_orders_summary
AS
SELECT customer_id, COUNT(*) AS total_orders
FROM orders
GROUP BY customer_id;

-- Refresh MV if source data changes
ALTER MATERIALIZED VIEW mv_orders_summary REBUILD;
```

- ◆ Key point      Hive can **automatically rewrite queries** to use MV when possible.

## Hive LLAP (Live Long and Process)

### ◆ What it is

- A **low-latency execution engine** for Hive.
- Instead of launching new containers for every query (slow in MapReduce/Tez), LLAP uses **long-running daemons**.
- Supports **in-memory caching + vectorized execution** → interactive speeds.

### ◆ Benefits

- Much faster for ad-hoc SQL queries.
- Supports BI tools (Tableau, PowerBI) better.
- Caches ORC/Parquet data in memory.

### ◆ How it works

1. LLAP daemons run continuously on cluster nodes.
2. Queries reuse these daemons → no startup overhead.
3. Data frequently accessed is cached in memory.

## Query Federation with Hive

### ◆ What it is:

- Hive can query **external systems** (like RDBMS, NoSQL, HDFS, S3) without moving data.
- Works via **storage handlers** and **connectors** (JDBC, HBase handler, Druid, etc.).

### ◆ Example

```
CREATE EXTERNAL TABLE mysql_orders (
    order_id INT,
    order_date STRING
)
STORED BY 'org.apache.hive.storage.jdbc.JdbcStorageHandler'
TBLPROPERTIES (
    "hive.sql.database.type" = "MYSQL",
    "hive.sql.jdbc.driver" = "com.mysql.jdbc.Driver",
    "hive.sql.jdbc.url" = "jdbc:mysql://localhost:3306/sales",
    "hive.sql.jdbc.username" = "root",
    "hive.sql.jdbc.password" = "root"
);
```

👉 Now you can run HiveQL directly on MySQL data.

### ◆ Use cases:

- Centralized query layer for **hybrid storage systems**.
- Avoids expensive ETL if only read access is needed.

## Hive Performance Tuning: Tez vs Spark

### ◆ Engines Hive supports:

- **MapReduce** (old, very slow).
- **Tez** (faster DAG execution engine).
- **Spark** (modern, in-memory parallelism).

### ◆ Comparison:

| Engine    | Pros                                                         | Cons                           |
|-----------|--------------------------------------------------------------|--------------------------------|
| MapReduce | Simple, fault-tolerant                                       | Very slow, high latency        |
| Tez       | DAG execution, low overhead, better suited for Hive          | Not as flexible as Spark       |
| Spark     | In-memory, supports ML & SQL, integrates well with pipelines | Needs more resources (RAM/CPU) |

👉 For interactive queries: LLAP + Tez.

👉 For ML/ETL pipelines: Spark engine.

### ◆ Tuning parameters

- `hive.execution.engine=tez or spark`
- Adjust `tez.am.resource.memory.mb` or Spark executor memory for performance.

## Optimizing ORC and Parquet Storage

### ◆ Columnar formats (ORC, Parquet) are optimized for analytics:

- Store data **by column** instead of by row.
- Enable **compression, predicate pushdown, vectorization**.

### ◆ Why ORC/Parquet?

- Smaller storage footprint.
- Faster queries (only read needed columns).
- Support for indexing, bloom filters.

### ◆ Optimizations:

1. **Column Pruning** → Hive reads only selected columns.
  2. **Predicate Pushdown** → Filters applied at storage level.
  3. `SELECT name FROM employees WHERE dept_id = 10;`
- 👉 Only `dept_id = 10` rows scanned.
4. **Compression:** Use ZLIB or SNAPPY.
  5. **Split ORC files properly** → Avoid too small files (use `hive.merge.*` properties).

### ◆ Example

```
CREATE TABLE employees_orc (
  id INT,
  name STRING,
  dept_id INT
)
STORED AS ORC
TBLPROPERTIES ("orc.compress"="ZLIB");
```

## Working with Skewed Data in Hive

### ◆ Problem

- Skew happens when certain values occur **much more frequently**.
- Example: In sales data, country='US' might have 80% of rows → causes reducer overload.

### ◆ Symptoms

- Some reducers take much longer than others.
- Job hangs near 99%.

### ◆ Solutions in Hive

#### 1. Skewed table handling

```
CREATE TABLE sales_skewed (
    item STRING,
    country STRING
)
SKEwed BY (country) ON ('US', 'IN')
STORED AS DIRECTORIES;
```

👉 Hive stores skewed values (US, IN) in separate directories.

#### 2. Enable skew optimization in joins

```
SET hive.optimize.skewjoin=true;
```

👉 Large skewed keys handled by map join, others by normal join.

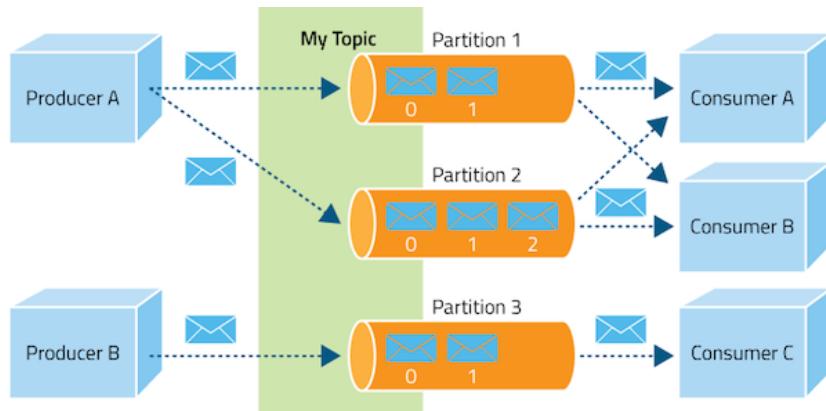
#### 3. Salting technique (manual)

- Add random numbers to keys (country+rand()) to distribute load, then re-aggregate.

## Summary Table

| Topic              | Purpose                      | Key Benefit                             |
|--------------------|------------------------------|-----------------------------------------|
| Materialized Views | Cached query results         | Faster repeated queries                 |
| LLAP               | Long-running daemons + cache | Interactive queries                     |
| Query Federation   | Query external DBs via Hive  | Single query layer                      |
| Tez vs Spark       | Execution engines            | Tez → fast Hive queries, Spark → ML/ETL |
| ORC/Parquet        | Columnar storage             | Faster queries + compression            |
| Skewed Data        | Balance load in reducers     | Avoid stragglers                        |

# Apache Kafka



## Introduction to Apache Kafka

### What is Kafka?

Apache Kafka is a **distributed event streaming platform** used to build **real-time data pipelines** and **event-driven applications**.

- It's like a **high-performance messaging system** but designed for **big data and real-time analytics**.
- Handles **millions of events per second**.
- Stores, processes, and replays data streams in a fault-tolerant and scalable way.

👉 Think of it as a **central nervous system** for data in modern organizations.

### History and Evolution (LinkedIn → Apache)

- **2010:** Developed at **LinkedIn** to handle their activity stream (feeds, clicks, messages).
- **2011:** Open-sourced as **Apache Kafka** (under Apache Software Foundation).
- **Today:** Used by **Netflix, Uber, Airbnb, Twitter, PayPal, LinkedIn, and banks** for real-time processing.

### Kafka vs Traditional Messaging Queues

| Feature             | Traditional Queues (RabbitMQ, ActiveMQ) | Apache Kafka                                     |
|---------------------|-----------------------------------------|--------------------------------------------------|
| <b>Primary use</b>  | Messaging (short-term)                  | Event streaming (long-term storage + processing) |
| <b>Storage</b>      | Messages deleted once consumed          | Data stored durably (days/weeks/months)          |
| <b>Throughput</b>   | Thousands of messages/sec               | Millions of messages/sec                         |
| <b>Scalability</b>  | Limited scaling                         | Horizontally scalable (add brokers)              |
| <b>Replay</b>       | Not possible (once consumed, gone)      | Consumers can re-read (event replay)             |
| <b>Use case fit</b> | Small apps, chat systems                | Big data pipelines, real-time analytics          |

## Kafka Ecosystem Overview

Kafka isn't just a messaging tool — it's a **complete ecosystem**.

### 1. Kafka Broker

- A **Kafka server** that stores and serves messages.
- Kafka cluster = multiple brokers (each broker can handle TBs of data).

### 2. Producers

- Applications that **send (publish) data** to Kafka.
- Example: An e-commerce app producing **order events** into a Kafka topic.

### 3. Consumers

- Applications that **read (subscribe)** from Kafka.
- Example: A recommendation engine consuming **user activity logs**.

### 4. Topics & Partitions

- **Topic** = logical channel where data is published (e.g., orders, payments).
- Each topic is split into **partitions** → enables **parallelism** and scalability.
- Example: Topic = orders, with 5 partitions = 5 consumers can process in parallel.

### 5. Zookeeper / KRaft

- Kafka originally used **Zookeeper** for managing cluster metadata (brokers, partitions, leadership).
- New versions (Kafka 2.8+) support **KRaft (Kafka Raft Metadata mode)** → removes dependency on Zookeeper for simplicity.

## Why Kafka is Popular

- ✓ Handles **real-time + batch** data.
- ✓ **Scalable:** add brokers and partitions easily.
- ✓ **Fault-tolerant:** data replicated across brokers.
- ✓ **Durable:** events stored for days/weeks.
- ✓ **High throughput:** millions of events/sec.

# Key Features of Apache Kafka

## 1. High Throughput

- Kafka can process **millions of messages per second**.
- Designed for **low latency** (few milliseconds).
- Perfect for high-volume use cases like IoT, logs, financial transactions.

## 2. Scalability

- Kafka is **horizontally scalable** → just add more brokers and partitions.
- Consumers can scale too (consumer groups).
- Example: A topic with 10 partitions can be processed by 10 consumers in parallel.

## 3. Durability & Persistence

- Messages are written to **disk** and replicated across brokers.
- Data is not lost even if a server fails.
- Retention policies: keep data for **hours, days, or indefinitely**.

## 4. Fault Tolerance

- Replication ensures no single point of failure.
- If a broker fails, another broker becomes the **leader** for partitions.
- Consumers/producers automatically reconnect to healthy brokers.

## 5. Publish-Subscribe Model

- Kafka uses a **distributed pub-sub system**:
  - **Producers** publish messages to topics.
  - **Consumers** subscribe and read messages.
- Multiple consumers can independently read the same data.

## 6. Event Replay & Time Travel

- Unlike traditional queues, Kafka stores data for a configurable period.
- Consumers can **re-read messages** (event replay).
- Useful for debugging, auditing, or reprocessing old data with new logic.

## 7. Stream Processing

- Kafka integrates with **Kafka Streams API, Apache Flink, Spark Streaming**.
- Enables real-time transformations: filtering, aggregations, joins.
- Example: Counting website clicks per second.

## 8. Exactly-Once Processing

- Modern Kafka ensures **exactly-once delivery semantics** (no duplicates, no loss).
- Critical for financial transactions and sensitive apps.

## 9. Flexible Storage (Retention Policies)

- Kafka can keep data for:
  - A few minutes (temporary logs).
  - Days/weeks (analytics).
  - Forever (data lake-like behavior).

## 10. Decoupling of Systems

- Producers and consumers don't know about each other.
- Kafka acts as a **buffer/middleman**.
- Example: E-commerce order → Payment, Inventory, and Shipping services consume independently.

## 11. Support for Batch + Real-Time

- Kafka supports both:
  - **Streaming** (real-time analytics).
  - **Batch** (ETL pipelines into Hadoop/Spark).

## 12. Security Features

- Authentication (SASL, Kerberos).
- Authorization (ACLs).
- Encryption (SSL for data in transit).

## 13. Ecosystem & Integration

- Works with **Kafka Connect** (connectors for DBs, cloud services, file systems).
- **Kafka Streams API** for processing inside apps.
- Integrates with **Spark, Flink, Hadoop, Storm, Elastic, and cloud platforms**.

 **In Short:** Kafka is **fast, scalable, durable, fault-tolerant, and real-time**, making it the backbone of modern data architectures.

# Different Kafka Applications – Examples, Problems, and Alternatives

## Real-Time Log Aggregation & Monitoring

### ◆ Problem

- Companies generate **logs from thousands of servers, apps, and services.**
- Need **centralized collection** for monitoring, debugging, and alerting.

### ◆ How Kafka is Used

- Servers send logs to **Kafka topics**.
- Tools like **Logstash, Fluentd, or Filebeat** push logs into Kafka.
- Consumers (e.g., **Elasticsearch, Splunk, Grafana**) process logs in real time.

### ◆ Why Kafka?

- Handles **millions of logs per second** with low latency.
- Logs are **durable and replayable** → can reprocess past logs.
- **Decouples producers & consumers** → multiple monitoring tools can read the same data.

### ◆ Alternatives

- **Flume, RabbitMQ, Syslog**.
- **✗ Problems:** Flume not as scalable, RabbitMQ struggles with very high throughput.
- **✓ Kafka wins for high volume + replayability.**

## E-Commerce Order Processing System

### ◆ Problem

- When a user places an order: **Payment, Inventory, Shipping, and Notifications** must be updated.
- Needs **event-driven communication** across services.

### ◆ How Kafka is Used

- An "Order Placed" event is published to Kafka.
- **Payment service, Inventory service, Shipping service, Email service** all consume the event independently.

### ◆ Why Kafka?

- **Decouples services** (no tight coupling like REST calls).
- **Scalable** → services can be added without changing producers.

- Ensures **reliability** → events aren't lost.

#### ◆ Alternatives

- **RabbitMQ, ActiveMQ** → good for simple messaging.
- **✗** But they don't scale as well under heavy e-commerce traffic.
- **✓** Kafka handles **millions of orders/events per second**.

## Banking & Financial Transactions

#### ◆ Problem

- Must process **billions of transactions** reliably.
- Need **exactly-once semantics** (no duplicates, no missing data).
- Real-time **fraud detection** on streaming data.

#### ◆ How Kafka is Used

- ATM/UPI/Card transactions are published to Kafka.
- Real-time fraud detection models (Spark/Flink/Kafka Streams) consume and flag suspicious activity.
- Downstream systems (databases, audit logs) also consume transactions.

#### ◆ Why Kafka?

- **Exactly-once delivery** guarantees.
- **Durable, replayable** event log → compliance with auditing.
- **Low latency + high throughput** → essential for finance.

#### ◆ Alternatives

- **RabbitMQ** (not durable enough at large scale).
- **Traditional DB replication** (slow).
- **✓** Kafka is best for **reliability + speed + replayability**.

## IoT Data Streaming (Smart Devices / Sensors)

### ◆ Problem

- IoT devices (cars, smart meters, wearables) generate **massive sensor data streams**.
- Data needs to be processed in **real-time** for insights and control.

### ◆ How Kafka is Used

- Each IoT device sends sensor readings to Kafka.
- Kafka streams → Spark/Flink process data → dashboards & alerts.
- Data also stored in HDFS/S3 for long-term analytics.

### ◆ Why Kafka?

- Handles **millions of device streams**.
- Works in **distributed environments**.
- Supports **real-time + batch storage**.

### ◆ Alternatives

- **MQTT, RabbitMQ** work for small-scale IoT.
- **✗** But they fail when devices scale to millions.
- **✓** Kafka is ideal for **large-scale IoT ingestion**.

## Ride-Sharing / Food Delivery Apps (Uber, DoorDash, Swiggy, Zomato)

### ◆ Problem

- Need to process **real-time location updates, ride events, order status changes**.
- Must match **drivers with passengers or restaurants with delivery partners** instantly.

### ◆ How Kafka is Used

- Driver location updates → Kafka.
- Passenger requests → Kafka.
- Matching algorithms consume Kafka streams to assign rides.
- Status updates → Kafka → push notifications to users.

### ◆ Why Kafka?

- **Low latency streaming** → updates in milliseconds.
- **Scalable** to millions of drivers and users.
- **Reliable event log** for auditing rides/orders.

#### ◆ Alternatives

- **RabbitMQ** → can't handle scale.
- **Direct DB polling** → too slow.
- Kafka ensures **real-time + high throughput + replayability**.

## Video Streaming Platforms (Netflix, YouTube)

#### ◆ Problem

- Need to **recommend videos in real-time**.
- Track **user activity logs** (play, pause, search).
- Optimize content delivery networks (CDN).

#### ◆ How Kafka is Used

- Every user action (watch, pause, like) → Kafka.
- Recommendation engine consumes logs → updates suggestions.
- Monitoring systems analyze **streaming quality in real-time**.

#### ◆ Why Kafka?

- **Scalable log ingestion** from millions of users.
- Supports **real-time + batch analytics**.
- **Event replay** helps in retraining recommendation models.

#### ◆ Alternatives

- **Flume, Pub/Sub (Google)**.
- Kafka wins for **open-source, flexibility, and scaling**.

## Data Lake Ingestion (ETL Pipelines)

### ◆ Problem

- Enterprises need to collect data from **databases, APIs, applications**.
- Store into **HDFS, S3, or data warehouse** for analytics.

### ◆ How Kafka is Used

- Kafka Connect + Debezium capture DB changes (CDC).
- Kafka stores events → consumers load them into **HDFS, Hive, Snowflake, or Redshift**.

### ◆ Why Kafka?

- **Change Data Capture (CDC)** makes real-time ETL possible.
- Ensures **fault tolerance & durability** during ingestion.
- One pipeline → many consumers (analytics, ML, dashboards).

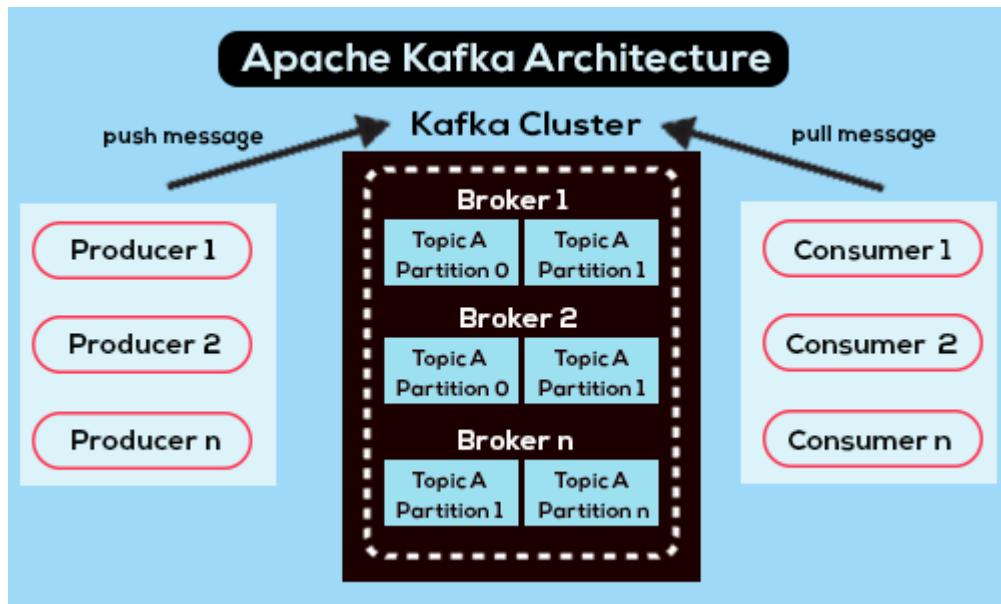
### ◆ Alternatives

- **Sqoop, Flume, traditional ETL tools.**
- **✗** They are batch-only and slow.
- **✓** Kafka provides **real-time + batch ETL**.

## Summary (Why Kafka Wins?)

- **High throughput** → millions of messages/sec.
- **Scalable** → easily add brokers/partitions.
- **Durable & replayable** → can reprocess data anytime.
- **Event-driven architecture** → services are decoupled.
- **Batch + real-time** → fits multiple workloads.

# Kafka Architecture



Apache Kafka is a **distributed streaming platform** designed for **high throughput, low latency, and fault-tolerant messaging**.

To understand how it works, let's break down the **core components**.

## Kafka Brokers

- A **Broker** is a **Kafka server** that stores data and serves clients (producers/consumers).
- Each broker is identified by a **unique ID** (e.g., broker-1).
- Brokers handle:
  - Receiving messages from **producers**
  - Storing them in **topics/partitions**
  - Serving them to **consumers**

👉 A single broker can handle **thousands of reads and writes per second**.

⚡ Example:

- If you run Kafka with 3 brokers → that's a **Kafka cluster**.
- If one broker goes down, others take over (fault tolerance).

## Kafka Clusters

- A **Kafka Cluster** is a group of brokers working together.
- Provides **scalability** (add more brokers = more capacity).
- Provides **fault tolerance** (data is replicated across brokers).

👉 Typically, production Kafka deployments run with **3–7+ brokers**.

## Topics

- A **Topic** is like a **category/feed name** to which records are sent.
- Producers write data to topics, consumers read data from topics.
- Topics are **multi-subscriber** (many consumers can read the same topic).

👉 Example:

- topic = "orders" → producer writes new e-commerce orders.
- Multiple consumers (fraud detection, shipping, analytics) can all read the same messages independently.

## Partitions

- A **topic is split into partitions** to allow **parallelism** and **scalability**.
- Each partition is an **ordered, immutable log** of messages.
- Messages inside a partition have a unique **offset** (sequence number).
- Partitions are distributed across brokers.

👉 Benefits:

- **Load balancing:** partitions spread across brokers.
- **Parallelism:** multiple consumers can process data in parallel.

## Example

- orders topic with **3 partitions** →
  - partition-0 on broker-1
  - partition-1 on broker-2
  - partition-2 on broker-3

## Offsets

- Each message in a partition is assigned a **unique ID called offset**.
- Offset = position of the message in the partition log.
- Consumers use offsets to **track which messages they have read**.

👉 Important:

- Offsets are **per-consumer group**, not global.
- If one consumer reads up to offset 100, another consumer group can still read from offset 0.

📌 Example:

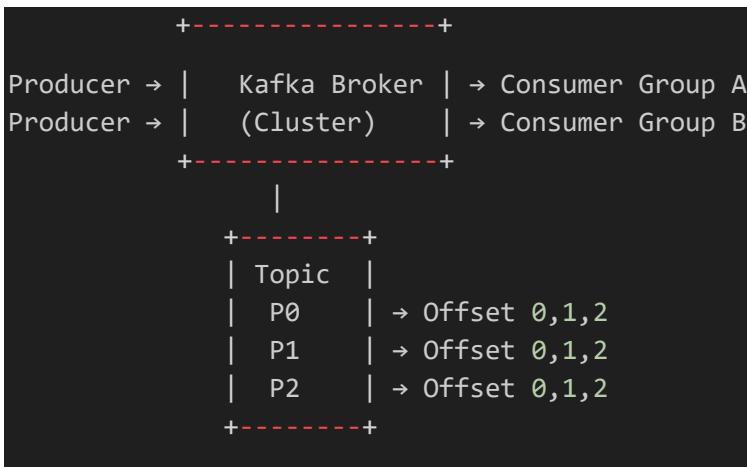
Partition-0: [msg0, msg1, msg2, msg3...]

Offsets: 0 1 2 3 ...

Consumer A may have processed up to offset 2,

Consumer B (different group) can start from 0.

## Visualizing Kafka Architecture



## Summary

- **Broker** → Kafka server (stores and serves data).
- **Cluster** → group of brokers (scalable & fault tolerant).
- **Topic** → logical stream (like “orders” or “payments”).
- **Partition** → splits a topic for parallelism (ordered logs).
- **Offset** → sequence number for each message in a partition.

# Kafka Partitioning

Partitioning is a **core concept in Kafka** that allows topics to scale horizontally and enables parallel processing of data.

## What is a Partition?

- A **partition** is an **ordered, immutable sequence of messages** in a Kafka topic.
- Each partition is like a **log file**.
- Messages inside a partition are **indexed by offset** (unique sequence number).

### Key Points:

- A topic can have **one or more partitions**.
- Partitions are **distributed across brokers** for scalability.
- Messages in a partition are **totally ordered**, but across partitions, **order is not guaranteed**.

## Why Partitioning is Important

### 1. Scalability:

- A topic with 1 partition → single broker handles all messages.
- Multiple partitions → messages spread across brokers → handles high throughput.

### 2. Parallelism:

- Multiple consumers in a **consumer group** can read different partitions in parallel.
- Each partition is consumed by **only one consumer per group**.

### 3. Fault Tolerance:

- Partitions can be **replicated** across brokers.
- If a broker fails, replicas take over.

## How Kafka Assigns Messages to Partitions

Kafka supports **two main partitioning strategies**

### 1 Round-Robin / Random

- If no **key** is provided, messages are distributed across partitions **evenly**.
- Example: 3 partitions → messages go P0, P1, P2, P0, P1, ...

## 2 Key-Based Partitioning

- If a **key** is provided, Kafka computes a **hash(key) % numPartitions** to determine partition.
- Ensures that all messages with the **same key go to the same partition** → preserves **order per key**.

📌 Example:

```
Producer.send("orders", key="customer_123", value="order details")
```

- All events for customer\_123 always go to the same partition.
- Ensures **per-customer ordering**.

## Partitioning Rules / Considerations

1. **Number of partitions** is decided when topic is created.
2. kafka-topics.sh --create --topic orders --partitions 5 --replication-factor 3 --bootstrap-server localhost:9092
3. **Increasing partitions** later is possible, but decreasing is not.
4. **Key choice** affects ordering and load distribution:
  - Poor key choice → **data skew** (some partitions overloaded).
5. **Consumer parallelism** limited to **number of partitions**.
  - 3 partitions → max 3 consumers can process in parallel per group.

## Benefits of Partitioning

| Feature                   | Benefit                                                  |
|---------------------------|----------------------------------------------------------|
| <b>Scalability</b>        | Handles high throughput by spreading load across brokers |
| <b>Parallelism</b>        | Multiple consumers can process in parallel               |
| <b>Fault tolerance</b>    | Replicas of partitions ensure no data loss               |
| <b>Ordered processing</b> | Messages with same key maintain order                    |
| <b>Replayability</b>      | Consumers can read from any offset independently         |

## Visual Example

```
Topic: orders (3 partitions: P0, P1, P2)
```

Producer sends messages:

```
Message 1 -> key: customer_101 -> Partition P0  
Message 2 -> key: customer_102 -> Partition P1  
Message 3 -> key: customer_101 -> Partition P0  
Message 4 -> key: customer_103 -> Partition P2
```

Consumer Group A

```
C1 reads P0  
C2 reads P1  
C3 reads P2
```

Result:

- Messages for same customer processed in order
- Parallel processing across partitions

## Partitioning Best Practices

1. Choose **number of partitions** carefully → affects scalability & parallelism.
2. Use **meaningful keys** → preserves ordering and reduces skew.
3. Monitor partition **load and offsets** → prevent uneven distribution.
4. Replicate partitions → ensure **high availability**.

## Summary:

- Partitioning allows Kafka to be **scalable, parallel, fault-tolerant, and ordered per key**.
- Key-based partitioning ensures **same key messages go to same partition**.
- Round-robin is useful when **order is not critical**.

## Kafka Producers

A **Producer** is a client application that **writes (publishes) messages to Kafka topics**. Producers are responsible for sending data efficiently and reliably.

### Asynchronous Sends

- **Synchronous send:** Producer waits for Kafka to confirm each message before sending the next.
- `producer.send(topic, value).get() # waits for acknowledgment`
  - **Safe but slow** → throughput is limited.
- **Asynchronous send** (recommended):
  - Producer **does not wait** for acknowledgment immediately.
  - Kafka batches messages and sends them efficiently.

```
future = producer.send(topic, value)
# can continue sending other messages
```

✓ Advantages: **High throughput, low latency**

⚠ Trade-off: Slight delay in knowing if send failed

### Acknowledgment Settings (acks)

The **acks** parameter controls **how many Kafka brokers must confirm a write** before the producer considers it successful.

| Setting                         | Description                                                | Pros                         | Cons                                                |
|---------------------------------|------------------------------------------------------------|------------------------------|-----------------------------------------------------|
| <code>acks=0</code>             | Producer does <b>not wait</b> for any broker response      | Fastest throughput           | Messages may be lost if broker fails                |
| <code>acks=1</code>             | Waits for <b>leader broker</b> to acknowledge              | Balanced reliability & speed | Data may be lost if leader fails before replication |
| <code>acks=all / acks=-1</code> | Waits for <b>all in-sync replicas (ISR)</b> to acknowledge | Strongest durability         | Slightly slower, but safe                           |

⚡ Example

```
producer = KafkaProducer(bootstrap_servers='localhost:9092', acks='all')
producer.send("orders", b"order_123")
```

## Batch Processing

- Producers **can batch multiple messages** together before sending to Kafka.
- Controlled by parameters like:
  - `batch.size` → max bytes per batch (default ~16 KB)
  - `linger.ms` → max wait time before sending a batch

## Why batching?

- Reduces network overhead.
- Increases throughput.

### Example:

- Producer accumulates 100 messages or waits 5 ms → sends them in a **single batch** to Kafka.

## Compression

- Kafka supports **message compression** to save network bandwidth and disk space.
- Supported codecs: gzip, snappy, lz4, zstd
- Works **per batch** → more efficient with larger batches.

```
producer = KafkaProducer(  
    bootstrap_servers='localhost:9092',  
    compression_type='snappy'  
)
```

### Benefits

- **Reduces network traffic**
- **Speeds up large message streams**
- **Better storage efficiency**

⚠ Trade-off: Slight CPU usage for compression/decompression.

## Producer Best Practices

1. **Use asynchronous sends** for high throughput.
2. **Set `acks=all`** for critical data to ensure durability.
3. **Enable batching and linger** → increase efficiency.
4. **Enable compression** for large volume messages.
5. **Handle errors** via callbacks or retries (retries parameter).

## Visual Overview

```
Producer Application
```

```
|
```

```
(batch, compress)
```

```
|
```

```
Asynchronous Send
```

```
|
```

```
Kafka Broker(s)
```

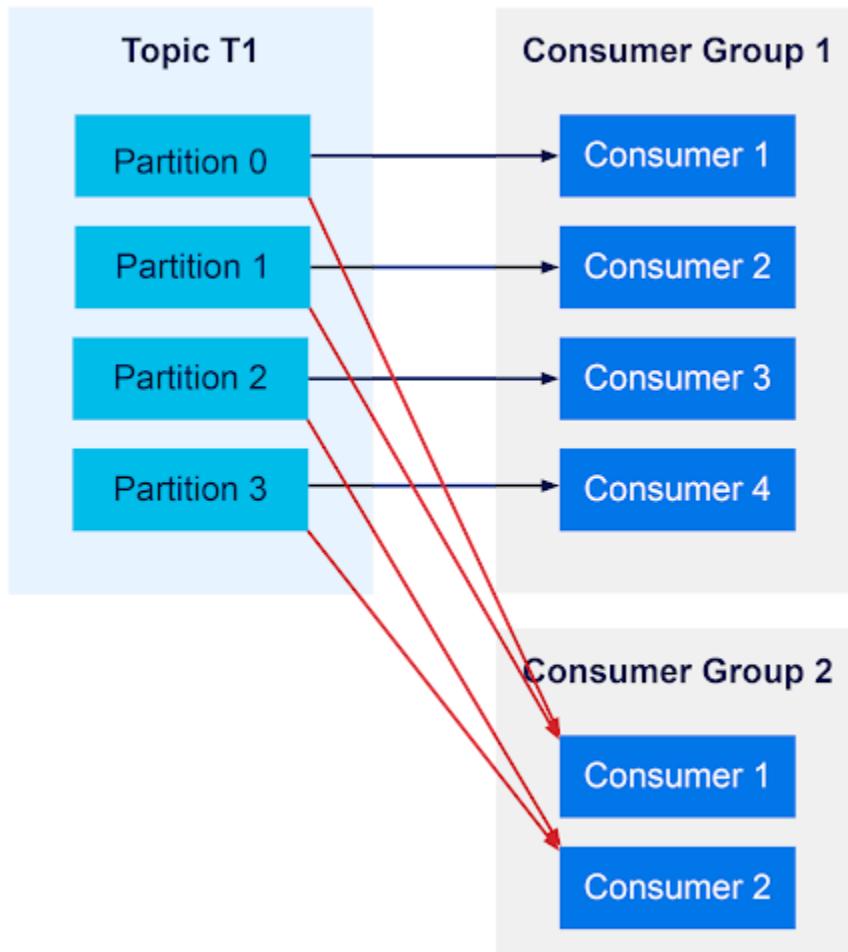
```
|
```

```
Acknowledgment: acks=0,1,all
```

## Summary

- Kafka producers can **send asynchronously** for high throughput.
- **Acks** control durability and reliability.
- **Batching + compression** increases efficiency and reduces network load.

## Kafka Consumers & Optimization



### Batch Processing & Compression (Producer Side Recap)

Even though this is about producers, it affects consumers too:

- **Batch Processing:**
  - Producers can **group multiple messages into a batch** before sending.
  - Configurations:
    - `batch.size` → maximum bytes per batch
    - `linger.ms` → maximum wait time before sending a batch
  - **Benefit:** fewer network calls → higher throughput
- **Compression:**
  - Supported codecs: gzip, snappy, lz4, zstd
  - Works **per batch**, reduces **network and disk usage**
  - Slight CPU overhead for compressing/decompressing messages

Consumers automatically **decompress messages**, no extra config needed.

## Kafka Consumers

- **Consumer** = client application that **reads messages from Kafka topics**.
- Can read **from beginning, end, or specific offsets**.

Key concepts:

- **Poll-based consumption:** Consumers poll messages from brokers.
- **Parallelism:** Multiple consumers can read from different partitions simultaneously.
- **Fault tolerance:** If a consumer fails, another can pick up its partitions.

## Consumer Groups

- A **consumer group** is a **set of consumers sharing the same group id**.
- Kafka guarantees that **each partition is consumed by only one consumer per group**.
- **Multiple groups** can read the same topic independently.

❖ Example:

- Topic orders has 3 partitions: P0, P1, P2
- Consumer Group A → 2 consumers:
  - Consumer1 → P0, P1
  - Consumer2 → P2
- Consumer Group B → another 2 consumers can independently read all partitions

✓ Benefit: **Horizontal scaling + parallelism**

## Rebalancing

- **Rebalancing** occurs when:
  1. A new consumer joins the group
  2. A consumer leaves the group
  3. Partitions change (topic updated)
- Kafka **redistributes partitions** among consumers in the group.
- **Impact:** Temporary pause in message consumption during rebalance.

❖ Tips to reduce rebalance impact:

- Use `max.poll.interval.ms` to allow consumers more processing time
- Keep consumer group size ≤ number of partitions

## Offset Management

- **Offset** = position of a message in a partition.
- Kafka supports **automatic and manual offset management**.

### Offset Types:

| Offset Type      | Description                                                                               |
|------------------|-------------------------------------------------------------------------------------------|
| <b>earliest</b>  | Consumer starts from the <b>oldest available message</b> in the partition                 |
| <b>latest</b>    | Consumer starts from the <b>latest messages</b> (messages produced after consumer starts) |
| <b>committed</b> | Consumer starts from the <b>last saved offset</b> (managed in Kafka or Zookeeper)         |

### Offset Commit Strategies

#### 1. Automatic commits

- `enable.auto.commit=true` → Kafka automatically commits offsets at intervals (`auto.commit.interval.ms`)

#### 2. Manual commits

- `enable.auto.commit=false` → app explicitly commits offsets after processing messages

#### Example (Python Kafka Consumer)

```
from kafka import KafkaConsumer

consumer = KafkaConsumer(
    'orders',
    bootstrap_servers='localhost:9092',
    group_id='order_processor',
    auto_offset_reset='earliest',    # start from beginning
    enable_auto_commit=True          # auto commit offsets
)

for message in consumer:
    print(message.value)
```

#### ✓ Benefit of proper offset management:

- Avoid **message loss**
- Avoid **duplicate processing**
- Enable **replay of messages** if needed

## Summary

| Concept                        | Purpose                     | Key Benefit                       |
|--------------------------------|-----------------------------|-----------------------------------|
| <b>Batch &amp; Compression</b> | Reduce network overhead     | Higher throughput                 |
| <b>Consumer</b>                | Reads messages              | Parallel, scalable consumption    |
| <b>Consumer Group</b>          | Multiple consumers in group | Partition-level parallelism       |
| <b>Rebalancing</b>             | Redistribute partitions     | Fault tolerance & load balancing  |
| <b>Offset Management</b>       | Track message position      | Prevent data loss / enable replay |

# Kafka Replication

## What is Kafka Replication?

- Kafka **replication** ensures **fault tolerance**.
- Each **topic** in Kafka is split into **partitions**.
- Each partition can have **multiple replicas** stored on different brokers.
- Among replicas:
  - **Leader:** Handles all read and write requests for that partition.
  - **Followers:** Replicas that **synchronize data** from the leader.
- If the leader fails, one of the followers automatically becomes the new leader.

## Key terms:

- **Replication factor:** Number of copies of a partition across the cluster.
- **ISR (In-Sync Replicas):** Followers that are fully caught up with the leader.

## Example Scenario

### Kafka Cluster Setup

- 3 brokers: Broker 1, Broker 2, Broker 3
- Topic: Orders
- Partitions: 2
- Replication factor: 2

### Partition assignment:

| Partition  | Leader  | Follower(s) |
|------------|---------|-------------|
| Partition0 | Broker1 | Broker2     |
| Partition1 | Broker2 | Broker3     |

### Step 1: Produce messages

When a producer sends a message to Orders partition 0:

1. Message goes to **Partition0 leader** (Broker1).
2. Leader appends message to its log.
3. Followers (Broker2) **replicate the message** asynchronously or synchronously (depending on **acks config**).
- If producer sets **acks=all**, Kafka ensures **all ISR replicas confirm** before returning success.

## Step 2: Broker failure

- Suppose **Broker1 fails** (leader of Partition0).
- Kafka automatically elects **Broker2** (follower) as the new **leader**.
- Producers and consumers now interact with Broker2.
- No data loss happens as long as **at least one replica in ISR** is available.

## Step 3: Recovery

- When Broker1 comes back online:
  - It becomes a **follower** of Partition0.
  - It **catches up** by replicating missed messages from the leader.
  - Once fully synced, it rejoins ISR.

## 3. Key Kafka Configurations

- **replication.factor** → Number of replicas per partition
- **min.insync.replicas** → Minimum replicas that must acknowledge a write for acks=all
- **acks** → Producer acknowledgment setting
  - 0 → No ack
  - 1 → Leader ack
  - all → All ISR ack

## 4. Quick Code

```
from kafka import KafkaProducer, KafkaConsumer

producer = KafkaProducer(bootstrap_servers='localhost:9092')

# send messages to a topic with replication factor 2
for i in range(10):
    producer.send('Orders', value=f"Order {i}".encode('utf-8'))

producer.flush()

consumer = KafkaConsumer('Orders', bootstrap_servers='localhost:9092',
                        auto_offset_reset='earliest',
                        group_id='order_group')

for msg in consumer:
    print(msg.value.decode())
```

- In the background, Kafka ensures messages are replicated to **followers**.
- If a broker fails, you will still be able to **consume messages** without loss.

## Kafka Architecture Components Recap

- **Producer** → Sends messages to Kafka topics.
- **Consumer** → Reads messages from Kafka topics.
- **Broker** → Kafka server storing messages.
- **Topic** → Category/feed name to which messages are published.
- **Partition** → Each topic is divided into partitions (parallelism).
- **Leader & Follower Replicas** → Leader handles requests, followers replicate for fault tolerance.
- **ZooKeeper / KRaft mode** → Manages cluster metadata and leader election.

### Example Scenario

**Use case:** Processing e-commerce orders.

- **Topic:** Orders
- **Partitions:** 2
- **Replication factor:** 2
- **Brokers:** Broker1, Broker2, Broker3

### Partition assignment example:

| Partition  | Leader  | Follower(s) |
|------------|---------|-------------|
| Partition0 | Broker1 | Broker2     |
| Partition1 | Broker2 | Broker3     |

## Kafka Workflow

### Step 1: Producer sends a message

1. A customer places an order on the website.
2. The backend app (Producer) serializes the order into a message.
3. Producer sends it to Kafka topic Orders.
4. Kafka decides **partition** using a **key** (like order\_id) → ensures same user orders go to same partition.

### Step 2: Broker handles the message

1. The **partition leader** receives the message.
2. Leader appends it to its **log** (disk storage).
3. Followers asynchronously replicate the message to maintain **replication factor**.
- If producer acks=all, Kafka waits for **all ISR replicas** to confirm before sending success back.

### Step 3: Consumer reads the message

1. Order processing service (Consumer) subscribes to topic Orders.
2. Consumer asks Kafka broker for new messages from its assigned partition(s).
3. Kafka sends messages in **offset order**.
4. Consumer processes the order (e.g., charge payment, update inventory).

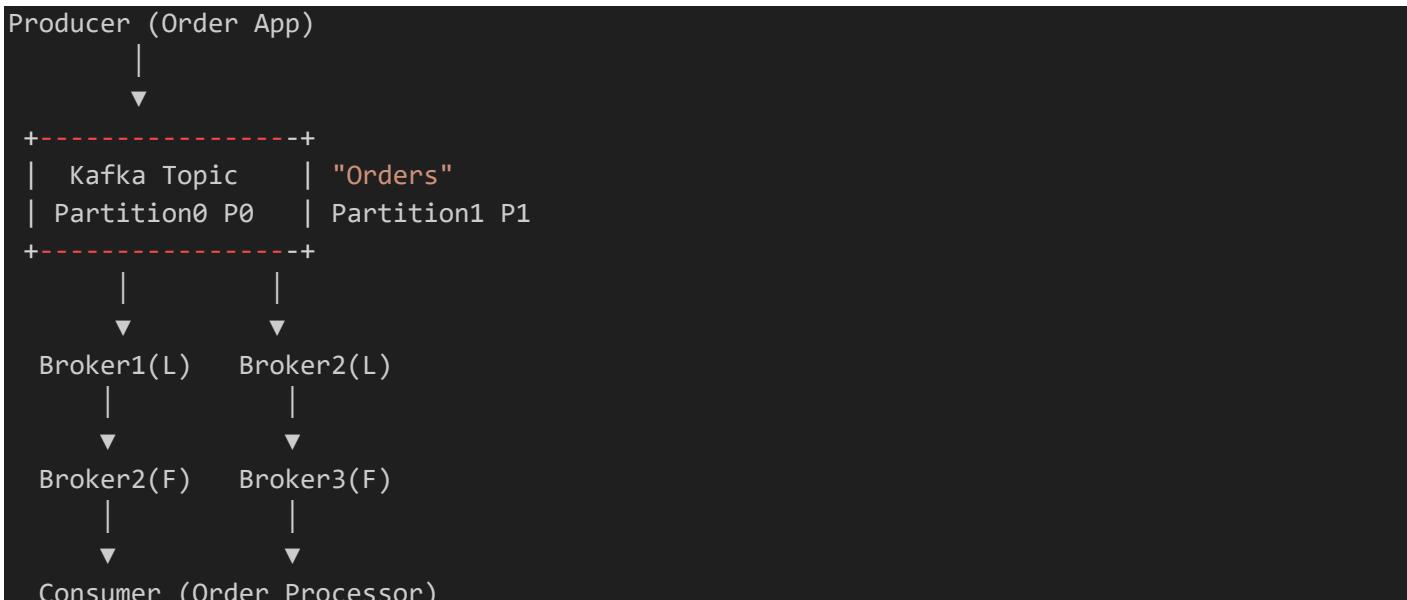
### Step 4: Offset management

- Kafka tracks **offsets** (position of the consumer in the partition).
- By committing offsets, consumers can restart without losing progress.

### Step 5: Replication and Failover

1. Suppose **Broker1 (leader of Partition0)** fails.
2. Kafka elects **Broker2 (follower)** as the new leader.
3. Producers and consumers automatically switch to the new leader.
4. Broker1 comes back → syncs missing messages → rejoins ISR.

## 4. Visual Diagram (Workflow)



- L → Leader, F → Follower
- Messages are first written to leaders → replicated to followers → consumed by consumers.

## Key Configurations in this Workflow

- `replication.factor = 2` → each partition has 2 copies

- acks = all → producer waits for all ISR replicas
- min.insync.replicas = 1 → minimum replicas required to acknowledge
- auto.offset.reset = earliest → start reading from beginning if no offset

## Summary

1. Producer → sends messages to Kafka topic partition.
2. Leader broker → stores message → replicates to followers.
3. Consumer → reads messages → commits offsets.
4. If broker fails → leader election ensures no downtime.

# Zookeeper vs KRaft in Kafka metadata management

## Overview

Kafka clusters need **metadata management** for:

- Tracking **brokers** in the cluster
- Managing **topics and partitions**
- Electing **partition leaders**
- Handling **controller election**

Historically, Kafka used **ZooKeeper**. Recently, Kafka introduced **KRaft mode** (Kafka Raft Metadata mode) to remove the dependency on ZooKeeper.

## ZooKeeper-Based Kafka

### Architecture

- ZooKeeper is a separate service.
- Kafka brokers connect to ZooKeeper.
- ZooKeeper stores:
  - Broker information
  - Topic metadata (partitions, replication factor)
  - Leader election state
- Kafka **controller** interacts with ZooKeeper for cluster coordination.

### Workflow

1. Broker starts → registers itself in ZooKeeper.
2. ZooKeeper maintains **cluster membership**.
3. When a leader fails:
  - Controller uses ZooKeeper to elect a new partition leader.
4. Consumers may also use ZooKeeper to store offsets (older versions).

### Pros

- Mature and stable.
- Strong coordination guarantees.

### Cons

- Extra operational complexity (ZooKeeper cluster to manage).
- Latency due to cross-service communication.

- Scaling issues with large clusters.
- Additional single point of failure if ZooKeeper misconfigured.

## KRaft-Based Kafka (Kafka Raft Mode)

### Architecture

- **KRaft** = Kafka Raft **metadata quorum**.
- Kafka brokers themselves store **metadata**; no external ZooKeeper.
- One broker acts as **controller**, others replicate metadata using **RAFT consensus protocol**.
- Metadata includes:
  - Broker membership
  - Topics, partitions, replication factor
  - Leader election state

### Workflow

1. Brokers form a **metadata quorum** (similar to Raft cluster).
2. Controller manages cluster state **internally**.
3. Partition leader election is handled **within Kafka**, no ZooKeeper calls.

### Pros

- No dependency on ZooKeeper → simpler architecture.
- Lower latency for metadata operations.
- Easier to scale.
- Single system for both metadata and messaging.

### Cons

- Newer, slightly less battle-tested.
- Migration from ZooKeeper mode requires careful planning.

## Key Differences: ZooKeeper vs KRaft

| Feature                       | ZooKeeper Mode                    | KRaft Mode (Kafka Raft)         |
|-------------------------------|-----------------------------------|---------------------------------|
| <b>Metadata Storage</b>       | External ZooKeeper cluster        | Kafka brokers themselves        |
| <b>Leader Election</b>        | Via ZooKeeper                     | Via Raft consensus inside Kafka |
| <b>Operational Complexity</b> | Higher (need separate ZK cluster) | Lower (single Kafka cluster)    |
| <b>Latency</b>                | Higher (cross-service calls)      | Lower (in-process)              |
| <b>Scaling</b>                | Limited by ZooKeeper              | Better scaling                  |
| <b>Maturity</b>               | Very mature                       | Newer, still evolving           |
| <b>Failover</b>               | Managed by ZK                     | Managed by Raft quorum          |

## Why KRaft?

- Reduce **operational complexity**.
- Simplify **metadata management**.
- Improve **scalability** and **performance**.
- Remove **ZooKeeper dependency**, making Kafka a **self-contained system**.

## Summary

- **ZooKeeper:** External service, coordinates brokers and metadata.
- **KRaft:** Internal Raft-based metadata quorum → simpler, faster, and scales better.

# Run kafka

## Local Installation

### a. Direct on OS (Linux/Windows/macOS)

- Download Kafka binaries from the [Apache Kafka website](#).
- Run **Zookeeper** (for older versions) and **Kafka broker** manually.
- Good for: learning, development, or small single-node setups.

#### Example (Linux):

```
# Start Zookeeper  
bin/zookeeper-server-start.sh config/zookeeper.properties
```

```
# Start Kafka broker
```

```
bin/kafka-server-start.sh config/server.properties
```

### b. With Package Managers

- On Linux:

```
# Using apt (Debian/Ubuntu)
```

```
sudo apt-get install kafka
```

- Not always latest version.
- Simplifies installation but less flexible.

## Docker

### a. Single Container

- Pull Kafka + Zookeeper Docker images:

```
docker run -d --name zookeeper -p 2181:2181 bitnami/zookeeper:latest
```

```
docker run -d --name kafka -p 9092:9092 \
```

```
-e KAFKA_CFG_ZOOKEEPER_CONNECT=zookeeper:2181 \  
-e ALLOW_PLAINTEXT_LISTENER=yes bitnami/kafka:latest
```

### b. Docker Compose (Recommended for clusters)

- Run multiple brokers + Zookeeper using docker-compose.yml.
- Easier to manage multi-node clusters for dev/testing.

## Kubernetes / Cloud Native

### a. Kafka on Kubernetes

- Use **Helm charts or Operators** (e.g., [Strimzi](#)).
- Advantages:
  - Auto-scaling brokers
  - Persistent storage via PVCs
  - Easy multi-node cluster deployment

#### Example (Strimzi Helm Chart):

```
helm repo add strimzi https://strimzi.io/charts/
```

```
helm install kafka strimzi/strimzi-kafka-operator
```

### b. Managed Kafka Services

- For production or cloud deployments:
  - **AWS MSK (Managed Streaming for Kafka)**
  - **Confluent Cloud Kafka**
  - **Azure Event Hubs (Kafka compatible)**
  - **GCP Pub/Sub (Kafka compatible)**
- Advantages: no infrastructure management, high availability, automatic scaling, security handled.

## Kafka Modes

### a. ZooKeeper Mode (Legacy)

- Brokers connect to **external Zookeeper** for metadata and leader election.

### b. KRaft Mode (New)

- Kafka runs **without Zookeeper**, using internal Raft quorum for metadata.
- Simplifies cluster management.

**Summary Table**

| Way                                         | Use Case                            | Pros                               | Cons                      |
|---------------------------------------------|-------------------------------------|------------------------------------|---------------------------|
| <b>Local installation</b>                   | Learning, development               | Easy, full control                 | Single-node, manual setup |
| <b>Docker</b>                               | Dev/test single/multi-node clusters | Portable, reproducible             | Needs Docker knowledge    |
| <b>Kubernetes + Strimzi</b>                 | Cloud-native deployments            | Auto-scaling, HA, operator-managed | Complex setup             |
| <b>Managed Kafka (MSK, Confluent Cloud)</b> | Production, no infra management     | Fully managed, HA, auto-scaling    | Cost, less control        |
| <b>KRaft mode</b>                           | Modern deployments without ZK       | Simplified, lower latency          | Newer, migration needed   |

# Confluent Kafka

## Cluster Creation

Confluent offers **two ways** to create a Kafka cluster

### ◆ Option 1: Confluent Cloud (Managed Service)

The screenshot shows the Confluent Cloud Cluster Creation interface. It consists of three main sections: Step 1 (Cluster name), Step 2 (Cluster type), and Step 3 (Provider and region).

- Step 1: Cluster name**  
The input field contains "cluster\_0".
- Step 2: Cluster type**  
A row of five cards:
  - Basic**: \$0.14 /eCKU-hr. Description: Start small with no risk. Test your use case and easily upgrade to Standard with one click.
  - Standard**: \$0.75 /eCKU-hr. Description: Great for most production-ready use cases with an extended feature set.
  - Enterprise**: \$1.75 /eCKU-hr. Description: Ideal for mission-critical use cases and sensitive data, offering private networking and increased scale.
  - Freight**: \$2.25 /eCKU-hr, minimum 2 eCKUs. Description: For high-scale use cases with relaxed latency requirements.
  - Dedicated**: \$2.66 /eCKU-hr. Description: Fully customizable for use cases with distinct networking or throughput requirements.
- Step 3: Provider and region**  
A row of three cards:
  - AWS
  - Microsoft Azure
  - Google Cloud (selected)

Region dropdown: S. Carolina (us-east1)

**Right-hand sidebar:**

- Cluster cost**: First eCKU free, then \$0.14 /eCKU-hr. Estimated monthly starting cost: \$0 /month. Assumes 1 eCKU. Actual cost is based on usage. When not utilized, eCKUs scale to zero to save you money.
- Cost details** table:

|                 |                  |
|-----------------|------------------|
| 1st eCKU        | Free             |
| Additional eCKU | \$0.14 /eCKU-hr  |
| Data in/out     | \$0.05 /GB       |
| Storage         | \$0.08 /GB-month |
- Summary** table:

|               |                        |
|---------------|------------------------|
| Cluster type  | Basic                  |
| Provider      | Google Cloud Platform  |
| Region        | S. Carolina (us-east1) |
| Uptime SLA    | 99.5%                  |
| Networking    | Internet               |
| Security      | Automatic              |
| Minimum eCKUs | 1                      |
- Buttons:** Launch cluster (blue button), Cancel, and a message icon.

- No need to manage servers
- You just sign up and create a cluster in a few clicks

## Steps

1. Go to [Confluent Cloud](#).
2. Click **Create Cluster**.
3. Choose **Cloud provider** (AWS, Azure, or GCP).
4. Choose **Region** (close to your users/data).
5. Choose **Cluster type**:
  - o **Basic** → Small, low-cost (for dev/test).
  - o **Standard** → Production workloads.
  - o **Dedicated** → Enterprise workloads.
6. Cluster gets provisioned (like getting a “Kafka server in the cloud”).

## Example

- Create a **Basic Cluster** on AWS, region ap-south-1 (Mumbai).
- Now you have a running Kafka broker without installing anything.

#### ◆ Option 2: Confluent Platform (On-Prem/Local Install)

- Download Confluent Platform
- Run Kafka + ZooKeeper (or KRaft in newer versions) + Schema Registry + Connect + Control Center

👉 Useful for learning locally or in enterprise environments.

📌 Example:

confluent local services start

This starts all services (Kafka, Schema Registry, Connect, Control Center).

### Topic Creation

A **topic** = A “channel” or “stream” where Kafka stores messages.

Confluent allows creating topics via

- **Confluent Control Center (UI)**
- **CLI**
- **API**

### Topic Settings

## New topic

**General**

|               |         |               |   |
|---------------|---------|---------------|---|
| Topic name* ⓘ | topic_0 | Partitions* ⓘ | 6 |
|---------------|---------|---------------|---|

**Storage**

|                  |        |                  |          |
|------------------|--------|------------------|----------|
| Cleanup policy ⓘ | Delete |                  |          |
| Retention time ⓘ | 1 week | Retention size ⓘ | Infinite |

Enable infinite retention for this topic

Retain your data in Kafka infinitely to access historical and current data all in one place for data analysis, regulatory purposes, or other use cases.  
[Learn more ↗](#)

**Message size**

|                                 |               |
|---------------------------------|---------------|
| Maximum message size in bytes ⓘ | 2097164 bytes |
|---------------------------------|---------------|

**Other Settings**

|                     |           |
|---------------------|-----------|
| cluster             | cluster_0 |
| replication.factor  | 3         |
| min.insync.replicas | 2         |

When creating a topic, you set

### 1. Topic Name

- Unique identifier (e.g., orders, payments, user-activity).

### 2. Number of Partitions

- Each topic is split into partitions → enables **parallelism & scalability**.
- More partitions = more throughput, but also more overhead.

Example

- Topic: orders with **3 partitions** → messages distributed across 3 partitions.

### 3. Replication Factor

- How many copies of each partition are stored across brokers.
- Ensures **fault tolerance**.
- Must be  $\leq$  number of brokers.

Example

- Replication factor = 3 → If one broker fails, data still safe on 2 others.

### 4. Retention Policy

- How long messages stay in topic.

- Two options:
  - **Retention time** → e.g., keep messages for 7 days.
  - **Log compaction** → Keep latest message per key (useful for state storage).

Example

- Retain orders messages for **7 days**.
- Or for user-profile topic, enable **log compaction** (always keep the latest user profile).

## 5. Cleanup Policy

- delete → Delete messages after retention period.
- compact → Keep last value per key.
- delete,compact → Hybrid.

## 6. Min In-Sync Replicas (ISR)

- Number of replicas that must acknowledge a message before it's considered committed.
- Works with acks=all in producers.

Example

- min.insync.replicas = 2, replication = 3
- Producer acks=all → at least 2 brokers must confirm before success.

## Example – Create a Topic

### Using CLI

```
kafka-topics --create \
--bootstrap-server my-cluster:9092 \
--replication-factor 3 \
--partitions 5 \
--topic orders
```

👉 Creates topic orders with 5 partitions, 3 replicas.

### Using Confluent Control Center (UI)

- Go to Topics → Create → Fill details (name, partitions, replicas, retention).

## Data Contracts

A **data contract** = an agreement between producer & consumer about **what the message will look like** (key, value, schema).

Without it → Consumers may break if data format changes.

With it → **Schema Registry** enforces rules.

### ◆ Message Keys

- Decide **partitioning**.
- Messages with the same key → always go to the same partition.
- Example keys
  - user\_id for user-events
  - order\_id for orders

Example

```
Key = {"user_id": 123}
Value = {"event": "login", "timestamp": "2025-08-31T10:00:00Z"}
```

### ◆ Message Values

- Actual event payload.
- Can be JSON, Avro, or Protobuf.
- Stored in **Schema Registry**.

❖ Example (Avro schema for orders value):

```
{
  "type": "record",
  "name": "Order",
  "fields": [
    {"name": "order_id", "type": "string"},
    {"name": "amount", "type": "double"},
    {"name": "currency", "type": "string"}
  ]
}
```

## ◆ Setting Up Schema Registry

### 1. Start Schema Registry (if local):

```
confluent local services schema-registry start
```

### 2. Register Schema for a Topic

```
curl -X POST http://localhost:8081/subjects/orders-value/versions \
-H "Content-Type: application/vnd.schemaregistry.v1+json" \
-d '{
    "schema": 
    "{\"type\":\"record\", \"name\":\"Order\", \"fields\":[{\"name\":\"order_id\", \"type\":\"string\"}, {\"name\":\"amount\", \"type\":\"double\"}]}"
}'
```

👉 Registers schema for orders topic **values**.

### 3. Producer sends Avro message

- Producer encodes messages with schema ID from Schema Registry.
- Consumer decodes using the same schema.

## Data Contract Types

### 1. For Keys

- Typically simple (string, int, UUID).
- Example: "user\_id"

### 2. For Values

- Complex objects (JSON, Avro, Protobuf).
- Example: Order event

### 3. Compatibility Rules (Schema Registry)

- **Backward compatible** → New schema can read old data.
- **Forward compatible** → Old consumers can read new data.
- **Full compatibility** → Both forward & backward.

## Why Data Contracts Matter?

- Prevents “breaking consumers” when schema changes.
- Enforces consistency across producers & consumers.
- Makes Kafka pipelines reliable in large systems.

## Build a client

# Build a client for cluster\_0

Produce and consume data with your cluster in the programming language of your choice.

## 1 Choose your language



## 2 Create an API key

It looks like you haven't set up an API key for this cluster. Create a new one to authenticate your client applications with your Kafka cluster.

[+ Create new API key](#)

## 3 Select a topic

Select an existing topic or create a new one.

[Use existing topic](#)

[+ Create a new topic](#)

## 4 Produce and Consume with Python

Download the quick start template and follow the setup guide to get up and running with your Python client.

[Download](#)

[Preview README](#)

## 5 Verify new client messages

Send a message to kafka (Producer side)

## Send a message to Confluent Kafka

```
[1] !pip install confluent-kafka
→ Collecting confluent-kafka
  Downloading confluent_kafka-2.11.1-cp312-cp312-manylinux_2_28_x86_64.whl.metadata (23 kB)
  Downloading confluent_kafka-2.11.1-cp312-cp312-manylinux_2_28_x86_64.whl (3.9 MB)
    ━━━━━━━━━━━━━━━━ 3.9/3.9 MB 41.3 MB/s eta 0:00:00
Installing collected packages: confluent-kafka
Successfully installed confluent-kafka-2.11.1
```

```
[2] # Required connection configs for kafka producer consumer and admin
from confluent_kafka import Producer
import json
import time
import pandas as pd
import numpy as np
```

```
[5] # create a json
json_records = df.to_dict(orient='records')
json_file = 'first_100_customers.json'
with open(json_file, 'w') as f:
    json.dump(json_records, f, indent=4)
print("File converted to json")
```

```
→ File converted to json
```

```
[4] df = pd.read_csv("/content/first_100_customers.csv")
df.head(5)
```

|   | customer_id | name       | city      | state       | country | registration_date | is_active | grid icon |
|---|-------------|------------|-----------|-------------|---------|-------------------|-----------|-----------|
| 0 | 0           | Customer_0 | Pune      | Maharashtra | India   | 2023-06-29        | False     | copy icon |
| 1 | 1           | Customer_1 | Bangalore | Tamil Nadu  | India   | 2023-12-07        | True      |           |
| 2 | 2           | Customer_2 | Hyderabad | Gujarat     | India   | 2023-10-27        | True      |           |

```

[12] # Create a client on confluent
conf = {
    'bootstrap.servers': 'pkc-619z3.us-east1.gcp.confluent.cloud:9092', # replace with your cluster
    'security.protocol': 'SASL_SSL',
    'sasl.mechanisms': 'PLAIN',
    'sasl.username': 'SZQMKVH3ZUOA7WJX',
    'sasl.password': 'cfI7z4aP03ld9Pby9LF1Ng607NgoJm5jHri05QUgD92hjFvmVyShyKDqIxI/LA'
}

[13] producer = Producer(conf)
topic = "ecommerce"

[14] # Load json
with open("/content/first_100_customers.json",'r') as file:
    customers_data = json.load(file)

# Send a single value

value = customers_data[0]
key = value["customer_id"]
print(value)
print(key)

producer.produce(topic, key=key, value=json.dumps(value))
producer.flush()

{'customer_id': 0, 'name': 'Customer_0', 'city': 'Pune', 'state': 'Maharashtra', 'country': 'India', 'registration_date': '2023-06-29'}
```

Traceback (most recent call last)  
 /tmp/ipython-input-1881721616.py in <cell line: 0>()  
 6 print(key)  
 7  
--> 8 producer.produce(topic, key=key, value=json.dumps(value))  
 9 producer.flush()

**TypeError**: a bytes-like object is required, not 'int'

[17] # value and key should be byte like objects. So need to be converted  
 producer.produce(topic, key= str(key).encode('utf-8'), value=str(value).encode('utf-8'))  
 producer.flush()

0

**ecommerce**

Messages   Monitor   Data contract   Configuration

Production in last hour -- messages   Consumption in last hour -- messages   Total messages 1   Retention time 1 week

Partition All   Consume Latest   Max results 1,000   CSV   JSON

Search...

| Timestamp                | Partition | Offset | Key | Value                                                                                                                                            |
|--------------------------|-----------|--------|-----|--------------------------------------------------------------------------------------------------------------------------------------------------|
| 2025-08-31T07:53:57.038Z | 2         | 0      | 0   | {'customer_id': 0, 'name': 'Customer_0', 'city': 'Pune', 'state': 'Maharashtra', 'country': 'India', 'registration_date': '2023-06-29', 'is_a... |

Page 1 < > Items per page 50 1 messages shown

## Send All data

```
# Callback function that Kafka calls after each message is delivered (or fails).
# This helps us confirm that our messages actually reached the Kafka cluster.
def delivery_report(err, msg):
    if err is not None:
        # If delivery fails, print the error
        print(f'Message delivery failed: {err}')
    else:
        # If delivery succeeds, print topic name and partition where message went
        print(f'Message delivered to {msg.topic()} [Partition: {msg.partition()}]')

# Loop through all customer records we want to send to Kafka
for record in customers_data:
    try:
        # Extract customer_id as the key
        # The key ensures that all messages from the same customer
        # go to the SAME partition (preserves ordering for that customer).
        key = record["customer_id"]

        # Convert the record (dictionary) into a string for sending.
        # Kafka messages must be bytes, so we encode strings into UTF-8.
        value = record
        producer.produce(
            topic,
            key=str(key).encode('utf-8'),           # Convert customer_id into bytes
            value=str(value).encode('utf-8'),         # Convert the whole record into bytes
            callback=delivery_report               # Call delivery_report when send is done
        )

        # Poll Kafka to trigger the delivery_report callback
        # This processes background network events and ensures delivery confirmation.
        producer.poll(1)

    except Exception as e:
        # If any error happens while producing, print it
        print(f"Error: {e}")

# Flush ensures that ALL buffered messages are sent before program exits.
# Without this, some messages may remain in memory and never reach Kafka.
producer.flush()
```

# Kafka Partitioning – How Messages are Assigned to Partitions

Kafka topics are divided into **partitions** for **scalability and parallelism**.

When a producer sends a message to a topic, Kafka decides **which partition** the message goes to based on some rules.

## 1 Partitioning Rules

Kafka decides the partition using this logic:

If a key is provided:

Partition = hash(key) % number\_of\_partitions

Else:

Round-robin (or sticky) assignment

### ✓ Case 1: Message has a Key

- If you provide a **key** when producing, Kafka **always sends all messages with the same key to the same partition**.
- Useful for ordering messages by customer, user, transaction, etc.
- ◆ Example:

```
producer.produce("orders-topic", key="user_123", value="order #1")
```

```
producer.produce("orders-topic", key="user_123", value="order #2")
```

→ Both messages go to the **same partition**, so their order is preserved.

### ✓ Case 2: Message has No Key

- Kafka will assign the partition **randomly (round-robin)** across available partitions.
- Distributes load evenly.
- But **ordering is not guaranteed** across different messages.
- ◆ Example:

```
producer.produce("orders-topic", value="order #3")
```

```
producer.produce("orders-topic", value="order #4")
```

→ Might end up in different partitions.

## 2 Why Partitioning Matters

- **Scalability** → More partitions = more consumers can process in parallel.
- **Ordering guarantee** → Messages are ordered **only within a partition**, not across partitions.
- **Data locality** → Same key → same partition → useful for joins, aggregations.

## 3 Example with 3 Partitions

Suppose topic orders-topic has 3 partitions (0, 1, 2):

| Key      | Hash(key) % 3 | Partition  |
|----------|---------------|------------|
| user_1   | 1             | P1         |
| user_2   | 2             | P2         |
| user_3   | 0             | P0         |
| (no key) | Round-robin   | P0, P1, P2 |

## Kafka Producer Functions Explained

| Function                          | What it does                                                                                                               | Delivery Guarantee                                                            | Why we need it                                                                                                                                | What happens if NOT used                                                                                                                       |
|-----------------------------------|----------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Callback (delivery_report)</b> | A user-defined function that runs <b>after a message is delivered (success/fail)</b> . It confirms delivery.               | Helps confirm message was <b>actually acknowledged</b> by broker (or failed). | - Debugging: Know which messages succeeded/failed.- Reliability: Log or retry failed messages.- Monitoring: Track per-message delivery stats. | - You send data "blindly" without knowing if Kafka accepted it.- Hard to debug if data is missing.- Failures (e.g., broker down) go unnoticed. |
| <b>poll(timeout)</b>              | Processes <b>background network events</b> . This is how the producer receives broker responses and triggers the callback. | Ensures that callback is <b>executed promptly</b> (delivery confirmation).    | - Without poll, callbacks won't run.- Required for producer's internal I/O loop.- Keeps producer responsive.                                  | - Callbacks may never trigger.- Messages might appear "stuck" in buffer.- You won't know if                                                    |

|                |                                                                                |                                                                                |                                                                                                                        |                                                                                                                                        |
|----------------|--------------------------------------------------------------------------------|--------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------|
|                |                                                                                |                                                                                |                                                                                                                        | delivery succeeded.                                                                                                                    |
| <b>flush()</b> | Blocks until <b>all buffered messages are sent</b> to Kafka (or error occurs). | Guarantees <b>at-least-once delivery</b> of all messages before program exits. | - Ensures no pending messages are left in memory.- Needed when app is about to close.- Good safety net at program end. | - Messages still in buffer may never reach Kafka.- Risk of <b>data loss</b> if program ends early.- Very common mistake for beginners. |

## Delivery Guarantee Levels

Kafka producers have 3 delivery guarantees (controlled by acks):

- **acks=0** → *fire-and-forget*, no guarantee, fastest but unsafe.
- **acks=1** → Leader broker acknowledges, **possible data loss** if leader crashes before replication.
- **acks=all** → All replicas acknowledge, **strongest guarantee** (safe but slower).

👉 Even with acks=all, if you skip flush(), unsent messages in **local buffer** can still be lost when program exits.

# Confluent Kafka Connectors

## What is Kafka Connect?

Kafka Connect is a **framework to move data in and out of Apache Kafka** without writing custom producer/consumer code.

- Instead of you writing Python/Java apps to read/write Kafka,
- Kafka Connect uses **connectors** (pre-built or custom plugins),
- Connectors handle **integration with databases, cloud services, and filesystems**.

👉 Think of it as a **data pipeline tool built into Kafka**.

## What is a Kafka Connector?

A **Kafka Connector** is a **plugin** (like a driver) that knows how to connect Kafka with another system.

There are **two types of connectors**:

| Connector Type          | Role                                                  | Example                                                 |
|-------------------------|-------------------------------------------------------|---------------------------------------------------------|
| <b>Source Connector</b> | Brings data <b>into Kafka</b> from an external system | Reading rows from PostgreSQL, streaming them into Kafka |
| <b>Sink Connector</b>   | Pushes data <b>out of Kafka</b> to an external system | Writing Kafka topic events to Elasticsearch or S3       |

## How Kafka Connect Works

Kafka Connect runs as a **separate service** (cluster of worker nodes).

It has 3 main parts:

1. **Connector** – high-level definition (e.g., "read from MySQL → write to Kafka topic").
2. **Tasks** – parallel workers that actually do the data transfer (scalable).
3. **Workers** – JVM processes running Connect. Can be standalone or distributed.

## Why use Kafka Connect?

Instead of reinventing the wheel with custom code:

- Prebuilt connectors save development time.
- Scalable (can increase tasks/workers).
- Managed offsets (no need to handle checkpoints manually).
- Standardized configs (JSON/YAML based).
- Integrates easily with **Confluent Cloud**.

## Confluent Cloud Connectors

Confluent (Kafka as a managed service) provides **fully managed connectors** — you don't even need to host Kafka Connect workers.

- You just configure the connector via Confluent UI or API.
- No servers to manage.
- Works with popular services (AWS S3, GCS, BigQuery, Snowflake, MongoDB, PostgreSQL, Datadog, Elastic, etc).

👉 Example: You can stream Kafka topic data into Snowflake for analytics in **minutes**.

## Example Workflow

### Scenario: Stream orders from PostgreSQL → Kafka → S3

1. **Source Connector: PostgreSQL CDC**
  - Captures inserts/updates/deletes from PostgreSQL database.
  - Streams them into Kafka topics (e.g., orders).
2. **Kafka Topic: orders**
  - Holds real-time order data.
3. **Sink Connector: S3**
  - Writes orders topic data into Amazon S3 as JSON/Avro/Parquet files.
  - Data can then be used for analytics, ML, or backups.

## Standalone vs Distributed Mode

Kafka Connect can run in 2 modes:

| Mode               | Use Case                       | How It Works                                          |
|--------------------|--------------------------------|-------------------------------------------------------|
| <b>Standalone</b>  | Local testing, small jobs      | Single JVM process runs 1 connector at a time         |
| <b>Distributed</b> | Production, scalable pipelines | Multiple worker nodes, tasks balanced, fault-tolerant |

👉 In **Confluent Cloud**, you don't worry about this — it's managed for you.

## Delivery Guarantees

Kafka Connect supports **exactly-once semantics (EOS)** in some sinks (like Kafka → Kafka, or Kafka → transactional DBs).

Otherwise, it provides **at-least-once** delivery by default.

⚠ This means:

- At-least-once = you may see duplicates if retries happen.
- Exactly-once = strict deduplication (but supported only by some connectors).

## Configuring a Connector

A connector is configured using a **JSON config file**. Example:

### Source Connector (Postgres → Kafka)

```
{  
  "name": "postgres-source-connector",  
  "connector.class": "io.confluent.connect.jdbc.JdbcSourceConnector",  
  "connection.url": "jdbc:postgresql://db:5432/ecommerce",  
  "connection.user": "dbuser",  
  "connection.password": "dbpass",  
  "mode": "incrementing",  
  "incrementing.column.name": "id",  
  "topic.prefix": "postgres-",  
  "poll.interval.ms": 1000  
}
```

## Sink Connector (Kafka → S3)

```
{  
  "name": "s3-sink-connector",  
  "connector.class": "io.confluent.connect.s3.S3SinkConnector",  
  "topics": "postgres-orders",  
  "s3.bucket.name": "kafka-orders-data",  
  "s3.region": "us-east-1",  
  "flush.size": 1000,  
  "format.class": "io.confluent.connect.s3.format.json.JsonFormat"  
}
```

👉 Deploy by sending JSON config to the Connect REST API or via Confluent Cloud UI.

## Popular Confluent Kafka Connectors

- **Databases (JDBC, MongoDB, Cassandra, Postgres, MySQL)**
- **Analytics (Snowflake, BigQuery, Redshift, Elasticsearch)**
- **Cloud Storage (AWS S3, GCS, Azure Blob Storage)**
- **Monitoring (Datadog, Splunk, Prometheus)**
- **Messaging (Pub/Sub, Kinesis, RabbitMQ)**

## Advantages & Limitations

### ✓ Pros

- Prebuilt, reliable connectors (saves dev effort).
- No need to write producer/consumer apps.
- Scales horizontally (distributed workers).
- Works seamlessly in Confluent Cloud.

### ⚠️ Cons

- Limited by available connectors (may need custom dev).
- May add latency (e.g., batch sinks like S3 flush periodically).
- EOS not supported everywhere.

## Custom Connectors

If no connector exists for your system:

- You can **write a custom connector** in Java (implementing SourceConnector/SinkConnector APIs).
- Or use **Kafka Connect transforms (SMTs)** to tweak data in-flight (lightweight ETL).

