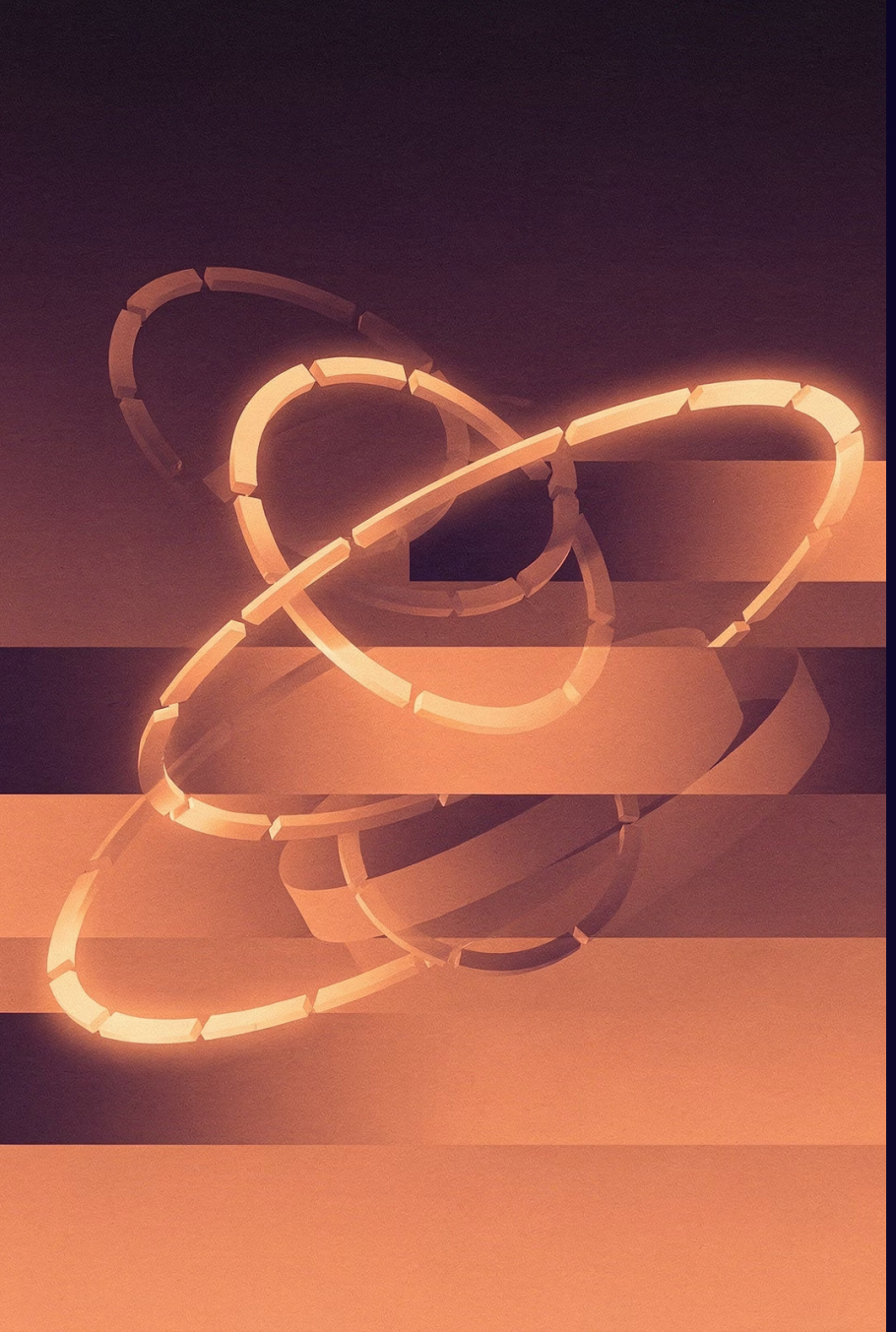
Spark Core — RDDs & Distributed Execution

Introduction to RDDs

Resilient Distributed Datasets (RDDs)

- Fundamental distributed data abstraction in Spark
- Immutable, partitioned collection of records
- Operated on in parallel across a cluster
- Fault-tolerant via lineage reconstruction



Why Partitions Matter

Partitioning Overview

Split into Partitions

RDDs are split into logical partitions

One Task Per Partition

Each partition equates to one task during execution

Parallelism Formula

Parallelism ≈ number of partitions * available cores

Auto-Determined

Spark determines partitions based on input & cluster config

Logical vs. Physical Partitioning

Logical Partitioning

- Defined when creating an RDD (e.g., `parallelize(data, numPartitions)`)
- Controls task granularity

Physical Partitioning

- How data actually resides on disk/storage (e.g., HDFS blocks)
- Affects locality & scheduling performance

File-Based RDD Partitioning

HDFS & File Input

Block-Based Reading

File data is read in blocks (default 128MB in HDFS)

Block-Aligned Partitions

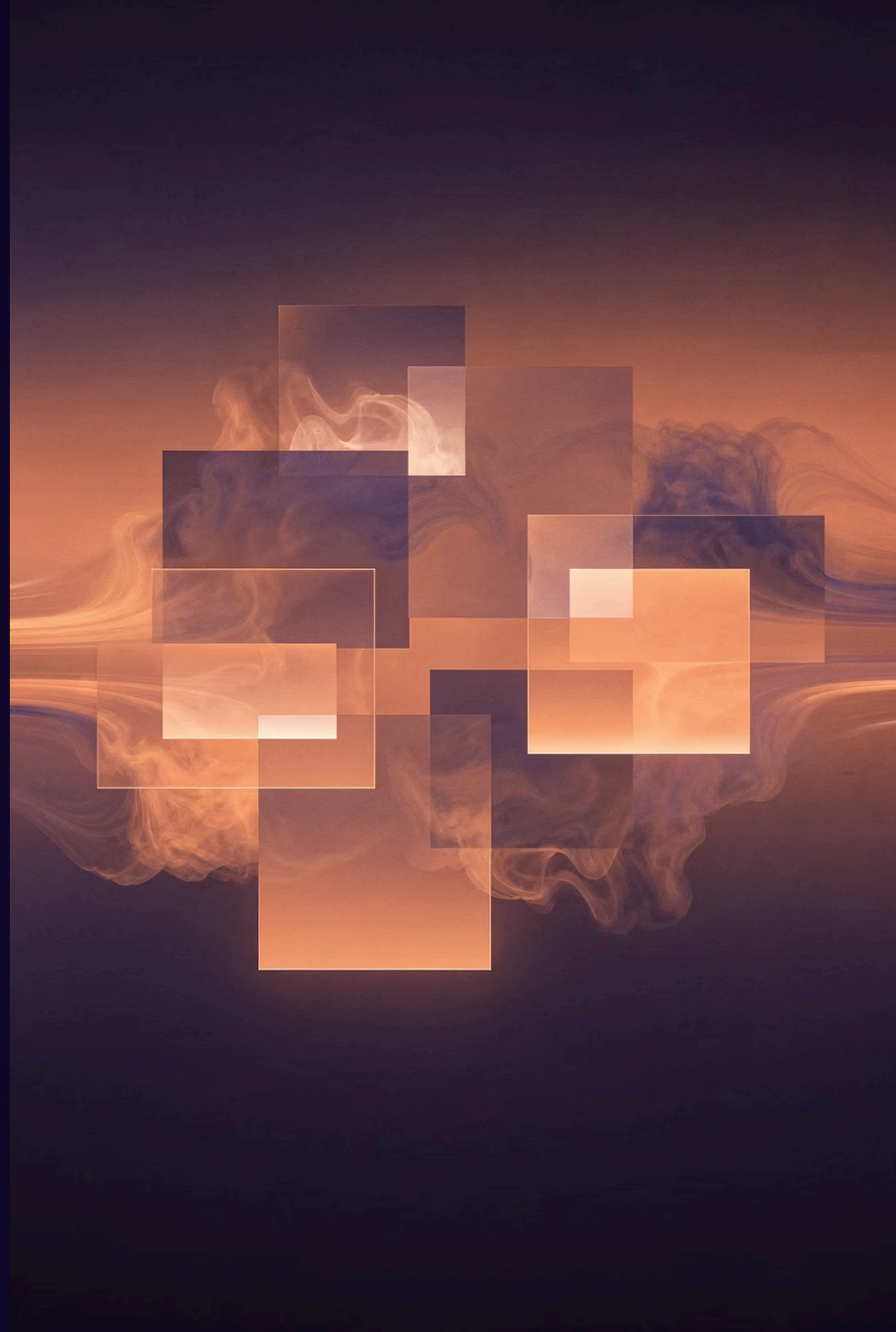
Spark divides files into partitions aligned with blocks

User Override

Users can override default number of partitions

Default Ratio

One partition per file block unless overridden



HDFS and Data Locality

Data Locality in Spark Scheduling

Spark scheduler tries to run tasks where data resides. Better locality → less network shuffle → higher performance.

1

PROCESS_LOCAL

Data in same JVM process — fastest possible

2

NODE_LOCAL

Data on same node — fast, no network

3

RACK_LOCAL

Data on same rack — moderate network cost

4

ANY

Data anywhere — highest network overhead

Transformations vs. Actions

Transformations (Lazy)

- Generate new RDDs
- Examples: `map`, `filter`, `flatMap`, `reduceByKey`
- Not executed until an action is called

Actions (Trigger Execution)

- Execution begins
- Examples: `collect`, `count`, `take`, `saveAsTextFile`

DAG, Stages & Tasks

DAG (Directed Acyclic Graph)

- Spark builds a logical plan from RDD transformations
- Used for execution optimization

Stages & Tasks

- DAG split into stages at shuffle boundaries
- Each partition in a stage → one task
- Tasks run in parallel across executors

Lazy Evaluation & Fault Tolerance

Lazy by Design

- RDD transformations don't compute immediately
- Spark keeps transformation DAG until an action
- Enables optimization & reduced work

RDD Lineage

- Spark maintains record of transformations that built an RDD
- If a partition is lost, Spark recomputes only that partition from lineage
- No full dataset checkpoint needed unless explicitly configured

RDD Persistence

Why Persist?

Recomputed by Default

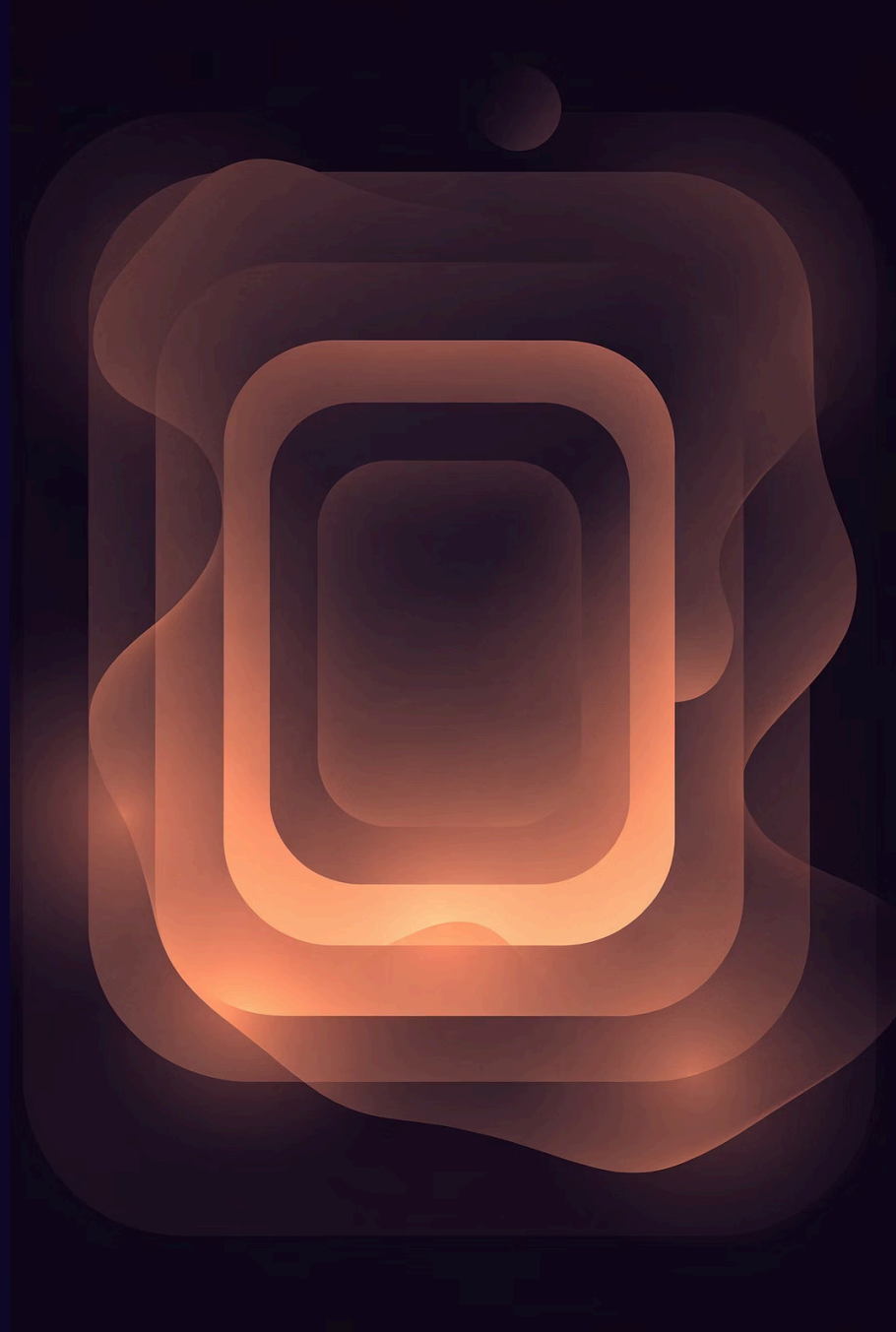
By default, RDDs are recomputed for every action

Persist to Reuse

Persist (cache) to reuse data across actions

Iterative Workloads

Improves performance especially in iterative/algo workloads



Persistence Levels

StorageLevel Examples

MEMORY_ONLY

Store in RAM — fastest access, no disk fallback

MEMORY_AND_DISK

RAM first, spill to disk if needed — balanced approach

DISK_ONLY

Store only on disk — reliable but slower

Trade-offs: Memory → faster | Disk → reliable but slower

Broadcast Variables & Accumulators

Broadcast Variables

- Share read-only data efficiently with all nodes
- Avoid shipping large datasets with every task
- Useful for joins with small reference datasets

Accumulators

- Shared variables for aggregating metrics (counters, sums)
- Only driver can read the value
- Executed in tasks for custom metric capture

Summary & Best Practices

Key Takeaways

- Partitions drive parallelism
- Lazy evaluation optimizes execution
- Persistence accelerates repetitive reads
- Broadcasts and accumulators enable efficient sharing and metrics

Performance Tips

- Tune partition count for balanced tasks
- Persist RDDs reused in multiple actions
- Leverage data locality for scheduling