# 0) The “problem” Spark solves (why it exists)

## Why not just MapReduce?

Your lecture says MapReduce can be **expensive** for some apps like **iterative** ones and it lacks efficient **data sharing**.

**Story:**
Imagine you’re cooking a big biryani for 1000 people. MapReduce is like:

- after every small step (cut onions, fry, add masala), you **pack everything**, **store it in a cupboard**, then **take it out again** for the next step.
  That constant “store to disk and read back” is slow.

Your slide literally shows this as “Expensive save to disk for fault tolerance.”

## Spark’s big idea: memory instead of disk

Spark says:

“Keep intermediate results in RAM (memory) so repeated steps are fast.”

The lecture contrasts: **Spark uses memory for data sharing**, Hadoop uses **disk**.

That’s the biggest reason Spark is fast.

---

# 1) What Spark is (definition + what it can do)

Your lecture describes Spark as:

- “lightning-fast cluster computing” for fast computation,
- supports interactive queries + stream processing,
- main feature: **in-memory cluster computing** that increases speed.

## Spark is not only “batch”

Spark is designed to cover many workloads:

- batch applications
- iterative algorithms (ML)
- interactive queries

- streaming

And Spark can work with many data sources (HDFS, HBase, Hive, S3, JSON, MySQL…) and many languages (Scala/Java/Python/R/SQL).

### Spark “modules” (big picture)

Spark has components: Spark Core, Spark SQL, Spark Streaming, MLlib, GraphX.

**Easy memory trick:**
Core = engine
SQL = tables/queries
Streaming = live data
MLlib = ML algorithms
GraphX = graphs

---

# 2) Spark Cluster Architecture (WHO does WHAT)

Your diagram shows: **Driver Program → Cluster Manager → Worker Nodes (Executors + Cache + Tasks)**.

Let’s make each role crystal.

## A) Driver (the “boss brain”)

The lecture says driver:

- runs SparkSession/SparkContext
- defines transformations/actions
- converts code → DAG → execution plan
- requests resources from cluster manager
- schedules tasks on executors
- “Driver = brain of Spark jobs”

**Story:** Driver is the manager who:

- reads your instructions
- makes a plan
- hires workers

- assigns work pieces

## B) Cluster Manager (the “resource allocator”)

It:

- launches executors on nodes
- allocates CPU/memory
- monitors node health

Types in lecture: Standalone, YARN, Mesos, Kubernetes.

**Story:** Cluster manager is like HR + operations: “how many workers, where, and how much power do they get?”

## C) Worker Nodes + Executors (the “workers”)

Executors:

- run tasks on partitions
- read data
- perform transformations
- store intermediate results in memory
- send results back

**Story:** Executors are the factory workers + machines.

## D) Tasks (smallest unit)

Lecture says: **Job** → **Stage** → **Task** execution hierarchy.

- **Job**: triggered by an action
- **Stage**: chunk of job separated by shuffle boundaries
- **Task**: smallest unit run on one partition

---

# 3) The Spark Workflow (end-to-end movie)

This slide is the full pipeline:

1. Input data (HDFS/S3/JDBC…)

2. Spark Driver creates DAG
3. Logical plan → optimized → physical plan → tasks
4. Cluster manager allocates executors
5. Executors run tasks on partitions
6. Shuffles/recompute if needed
7. Output (memory/disk/return to driver)

**Key exam line:** Spark builds plan first, then executes when action happens.

---

# 4) The CORE IDEA your lecture screams: PARTITIONS

Your slide literally says: “A Core Idea – Understand it First” then “Core Idea: Partition.”

## What is a partition?

A partition is:

- the smallest logical chunk of data inside an RDD/DataFrame
- processed independently and in parallel
- **one partition is processed by one task on an executor**

Also lecture connects HDFS blocks and partitions (like 128MB blocks mapping to partitions).

## Why partitions are EVERYTHING

Because Spark is built for parallelism:

- 1 TB dataset is not loaded on one machine
- it’s split (e.g., 100 partitions)
- partitions run in parallel

### The most testable line:

**#tasks per stage = #partitions**

So:

- 10 partitions → 10 tasks → parallel
- More partitions → more parallelism, but too many → scheduling overhead

**Simple example:**
If you have 4 partitions and 4 CPU cores available, Spark can process all 4 at the same time. If you have 100 partitions but only 4 cores, it will do them in waves.

---

# 5) Transformations vs Actions (LAZY vs EXECUTE)

This is one of the biggest exam favorites, and your lecture has a table.

## Transformations (LAZY)

They “set up” work and create new RDDs.
Examples:

- map, filter, flatMap, reduceByKey, groupByKey, join

**Story:** You’re writing a recipe, not cooking yet.

## Actions (EXECUTE NOW)

They trigger computation and either:

- return results to driver, OR
- write to storage (HDFS/file)

Examples:

- count, collect, take, saveAsTextFile, reduce

**Critical lecture sentence:**
“When collect() is called, Spark executes all transformations in a pipeline.”

---

# 6) DAG (Directed Acyclic Graph) — Spark’s “plan”

Lecture says Spark builds a **logical execution plan (DAG)** and submits it when an action is called.

And it defines DAG clearly:

- DAG tracks dependencies (lineage)
- nodes are RDDs
- arrows are transformations

### DAG in easy words:

Your code produces a flowchart:

RDD0 (input)
→ filter
→ map
→ reduceByKey
→ output

Spark keeps this as a graph because:

- it can optimize execution
- it can recover from failures using lineage

---

# 7) Stages + Shuffle Boundaries (where Spark “breaks” your job)

Lecture:

- DAG is divided into **stages** based on **shuffle boundaries**
- each stage becomes tasks (one per partition)

So what is a shuffle boundary?

---

# 8) Shuffle (the expensive monster)

Lecture defines shuffle as:

- redistribute data across partitions so same keys end up together
- happens between stages when Spark must group/join/aggregate across cluster

### Story example (super clear)

Imagine you have marks data split across 3 partitions:

- Partition 0: (Ali, 3), (Sara, 5)
- Partition 1: (Ali, 4)
- Partition 2: (Sara, 2)

Now you want total marks per student.
Ali's data is split across partitions, so Spark must MOVE records so all Ali records meet together.
That movement is shuffle.

Lecture gives the list of transformations that cause shuffle:
reduceByKey, groupByKey, join, distinct, repartition, sortByKey

---

# 9) Narrow vs Wide transformations (connected to shuffle)

Lecture has a table:

## Narrow (no shuffle)

- map, filter, flatMap
- can be done within partition

**Story:** each worker can finish its piece alone.

## Wide (shuffle)

- groupByKey, reduceByKey, join
- requires shuffle between executors

**Story:** workers must exchange data across network.

---

# 10) Fault tolerance: Lineage (how Spark recovers)

Lecture says:

- If executor fails, cluster manager reassigns tasks
- RDD/DataFrame lineage allows recomputation of lost partitions

And defines lineage:

- history of transformations used to create an RDD
- Spark doesn't store intermediate data by default
- if RDD lost, Spark recomputes using lineage
- lineage forms a DAG

Also: "No cost if no failure."

**Story:**
Instead of saving every step to disk (like MapReduce), Spark keeps the recipe.
If one partition is lost, Spark cooks that partition again from the recipe.

---

# 11) RDD (Resilient Distributed Dataset) — Spark's core data structure

Lecture lists RDD properties:

- Immutable (can't be changed, only transformed)
- Distributed (partitioned across cluster)
- Lazy evaluated (until action)
- Resilient (recovers using lineage)
- can be cached for reuse

### What "immutable" means (easy)

If you do `rdd2 = rdd1.map(...)`, rdd1 is unchanged. You get a NEW dataset.

---

# 12) Caching / Persist (the “in-memory superpower”)

Lecture says Spark allows caching to speed iterative computations.

If you keep using the same RDD repeatedly (ML training, PageRank), caching saves you from recomputing and re-reading from disk each time.

Storage levels:

- MEMORY_ONLY
- MEMORY_AND_DISK
- DISK_ONLY
- MEMORY_ONLY_SER

Meaning (in easy words):

- MEMORY_ONLY: keep in RAM, if lost → recompute
- MEMORY_AND_DISK: RAM first, overflow spills to disk
- DISK_ONLY: store on disk
- MEMORY_ONLY_SER: serialized (compressed-ish) in RAM

---

# 13) Optimization: groupByKey vs reduceByKey (EXAM TRAP)

Lecture says:

- groupByKey shuffles **all values** for each key → heavy network + memory
- reduceByKey does **local aggregation first**, then shuffles partial results → less data movement

**Story:**
groupByKey = bring every receipt of every store to head office, then add.
reduceByKey = each store adds locally, sends only totals.

---

# 14) More optimization techniques (also in lecture)

Lecture lists:

- repartition intelligently (repartition/coalesce)
- broadcast variables for small lookup data
- avoid large collect() (crashes driver)

**Collect() is dangerous** because it brings EVERYTHING to driver memory.

---

# 15) Spark + Hadoop integration (big picture)

Lecture: Hadoop is used for:

- storage (HDFS)
- resource management (YARN)
- metadata/SQL interoperability (Hive/HCatalog)
- formats (Parquet/ORC/Avro)
- security (Kerberos/ACLs)

And Spark can be built with Hadoop in 4 ways:

1. Spark on HDFS (storage only)
2. Spark on YARN (cluster manager)
3. Spark with Hive/HCatalog (metastore + SQL)
4. Spark with full Hadoop stack (HDFS + YARN + Hive + HBase + Security)

---

# 16) DataFrames vs RDDs vs Datasets (breadth topic)

Lecture says Spark has three APIs:

- RDD (low-level, Spark 1.0)
- DataFrames (Spark 1.3)
- Datasets (Spark 1.6)

## DataFrame

- data in named columns like a table
- higher-level abstraction
- optimized by Spark SQL engine (Catalyst)

## Dataset

- extension of DataFrame
- type-safe in Scala/Java (compile-time checks)
- Python/R don't support Dataset type safety fully

**Exam-friendly line:** DataFrames/Datasets are optimized; RDDs give low-level control.

---

# 17) Spark vs Hadoop MapReduce (comparison points)

Lecture summary:

- Spark is generally faster due to in-memory
- but needs lots of memory; performance degrades if not enough memory
- MapReduce is mature, good for 1-pass jobs, works alongside other services

---

# 18) Spark use cases + real-world examples

Lecture lists use cases:
streaming, ML, interactive analysis, warehousing, batch, EDA, graph, GIS, etc.

Real-world:

- Uber: Kafka + Spark Streaming + HDFS continuous ETL
- Pinterest: Spark ETL + Spark Streaming for real-time recommendations

- Netflix/Capital One/Conviva: streaming analytics, fraud patterns, recommendations

---

# 19) When NOT to use Spark (important breadth)

Lecture says:

- for simple use cases, MapReduce/Hive might be better
- Spark not designed as multi-user environment
- users must ensure memory is sufficient
- multiple users require coordination of memory usage

---

# 20) Final “one-page” mental model (if you remember ONLY this, you pass)

1. Spark splits data into **partitions**
2. Each partition → **one task** (parallel)
3. Your transformations are **lazy** (build plan)
4. Action triggers execution → Spark builds **DAG**
5. DAG splits into **stages** at **shuffle** boundaries
6. Shuffle = data movement across executors (expensive)
7. Fault tolerance via **lineage** (recompute lost partitions)
8. Cache/persist makes iterative algorithms fast