PREPNPLACED

✦ learn · prep · get placed ✦

handwritten notes ✍️

APACHE SPARK

End-to-End · how it works · components · architecture · PySpark

INSIDE THESE 20 PAGES →

- ✓ How Spark works
- ✓ Cluster architecture
- ✓ DAG · stages · tasks
- ✓ Joins, caching, streaming
- ✓ Components & ecosystem
- ✓ RDDs & DataFrames
- ✓ PySpark internals + code
- ✓ Optimization cheat sheet

PLACEMENT PREP SERIES

20 pages · interview-ready ⚡

What's inside ✍️

— tick topics as you revise!

★ = asked a LOT in interviews — revise twice!

What is Apache Spark? — the big picture

DEFN: Spark = an open-source, distributed, in-memory computing engine for processing **huge datasets in parallel** across a cluster of machines.

Why did we need it? (the MapReduce problem)

100x faster — imp!! ★

- → Data got too big for one machine ⇒ split it across many machines & process in parallel.
- → Hadoop MapReduce did this, BUT it **writes to disk after every step** ⇒ painfully slow for multi-step jobs.
- → Spark keeps intermediate data in RAM (memory) ⇒ up to **100x faster** in memory, ~10x on disk.

Key features ⚡

- ✓ **In-memory compute** — RAM first, disk only if needed
- ✓ **Lazy evaluation** — plans first, runs only on action
- ✓ **Fault tolerant** — rebuilds lost data via lineage
- ✓ **Polyglot** — Python, Scala, Java, R, SQL
- ✓ **Unified engine** — batch + streaming + ML + graph
- ✓ **Runs anywhere** — YARN, Kubernetes, standalone, cloud

Spark vs MapReduce — 4 lines to remember

	SPARK ⚡	MAPREDUCE
Speed	in-memory ⇒ very fast	disk after each step ⇒ slow
Ease of use	rich APIs — df.groupBy( )...	verbose map + reduce code
Workloads	batch, streaming, ML, SQL	batch only
Interactive?	yes — shell & notebooks	no

☞ *One-liner for interviews: "Spark is a unified, in-memory, distributed compute engine — MapReduce's faster, friendlier successor."*

The Spark ecosystem — one engine, many libraries

Spark SQL
tables, SQL queries, DataFrames

Structured Streaming
real-time data pipelines

MLlib
distributed machine learning

GraphX
graph processing

↓ ↓ ↓ ↓ (all sit on top of...)

SPARK CORE ⚙
RDD API · task scheduling · memory mgmt · fault recovery · storage interaction

↓ (runs on a cluster manager) ↓

Standalone | **YARN** | **Kubernetes** | **Mesos (deprecated)**

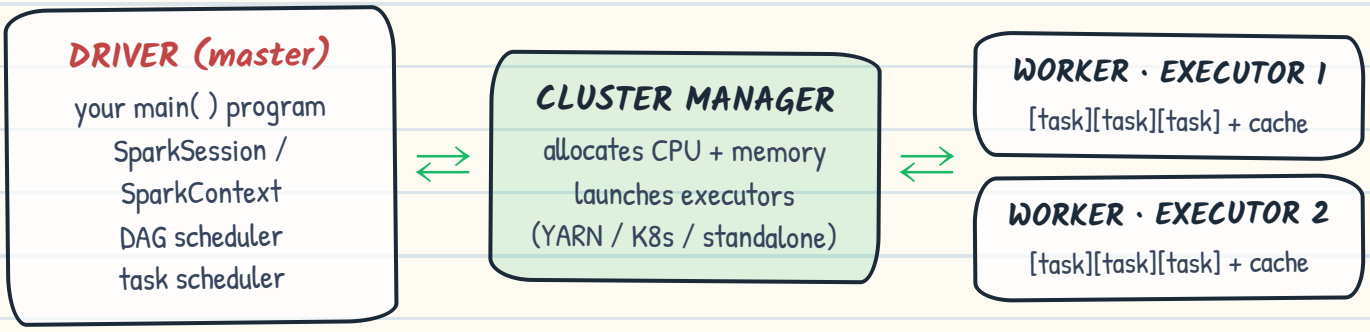
Each component in one line ✍

- → **Spark Core** — the engine: RDDs, DAG scheduler, memory & fault tolerance. Everything else compiles down to this.
- → **Spark SQL** — work with structured data via SQL or DataFrame API; brings the Catalyst optimizer (pg 12).
- → **Structured Streaming** — treat a live stream as an ever-growing table; same DataFrame code (pg 19).
- → **MLlib** — scalable ML: pipelines, classification, regression, clustering, recommenders.
- → **GraphX** — graph algorithms (PageRank, connected components) on RDDs (Scala-side).
- → **Languages** — Scala (native), Python = PySpark, Java, R, SQL.

NOTE ✎ — "Unified engine" is the selling point: ETL + SQL + streaming + ML in **one framework**, one deployment, one API family. No tool zoo!

Cluster architecture ★

— master / worker model



driver ⇢ sends **tasks** · executors ⇠ send back **results + heartbeats**

Who does what?

draw this in interview! ★

① **Driver** — runs your main( ), creates the SparkSession, converts code → DAG → stages → tasks, schedules them on executors, tracks progress, collects results. Dies ⇒ application dies!

② **Cluster manager** — the resource broker: driver asks it for CPU/memory; it launches executor processes on worker nodes.

③ **Executors** — JVM processes on worker nodes that **actually run tasks** (1 task per core-slot) and hold cached data in memory. Report status back to driver.

Remember ✍ — 1 executor = 1 JVM with N cores ⇒ runs N tasks in parallel. Total parallelism = executors × cores each.

Q. "Is the driver on the cluster or your laptop?" → depends on **deploy mode** — see pg 6!

How a job runs — end to end ★

1. You submit the app: **spark-submit my_job.py** (or run a notebook cell).
2. **Driver** process starts → creates SparkSession (entry point to the cluster).
3. Driver asks the **cluster manager** for resources → it launches **executors** on workers; they register back with the driver.
4. Transformations are recorded lazily — nothing executes yet, Spark just builds the plan (DAG).
5. An **action** (collect, count, write…) triggers a **job** → DAG scheduler splits it into **stages** (at shuffles) → stages into **tasks** (1 per partition).
6. Task scheduler ships tasks to executors — preferring nodes where the data already lives (**data locality**).
7. Executors run tasks (shuffling data between stages if needed), then return results / write output.
8. **spark.stop( )** → executors shut down, cluster manager reclaims resources. Done ✓

```
# a typical submit —
spark-submit --master yarn --deploy-mode cluster \
  --num-executors 10 --executor-cores 5 --executor-memory 8g \
  my_job.py
```

client mode — driver runs on your machine / edge node. Good for notebooks & debugging.

cluster mode — driver runs inside the cluster. Production default — survives your laptop closing!

RDDs — the foundation

— everything compiles to these

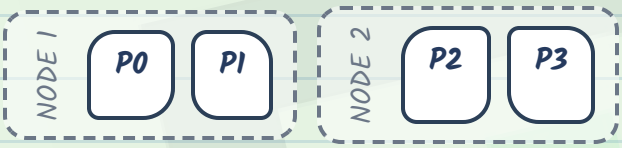
RDD = Resilient Distributed Dataset → an immutable, partitioned collection of records that can be processed in parallel.

Resilient
fault tolerant — lost partitions are rebuilt from lineage, no replication needed

Distributed
data lives in partitions spread across executor nodes

Dataset
just a collection of records — lines, tuples, objects...

one RDD, four partitions, two nodes ↓



← 4 partitions ⇒ 4 parallel tasks!

Properties to rattle off ✍

- ✓ **Immutable** — transformations make new RDDs
- ✓ **Lazy** — computed only when an action fires
- ✓ **Partitioned** — unit of parallelism
- ✓ **Lineage-tracked** — remembers how it was built
- ✓ **In-memory** — cacheable across actions
- ✓ **No schema** — raw objects (DataFrames add schema)

3 ways to create one

```
rdd1 = spark.sparkContext.parallelize([1, 2, 3, 4])  # from a python listrdd2 = spark.sparkContext.textF
```

☞ Today you'll mostly write DataFrames (pg 11) — but under the hood, it's RDDs all the way down!

Transformations & actions

TRANSFORMATIONS

RDD/DF → new RDD/DF · LAZY — just noted in the plan

map( ) · filter( ) · flatMap( )

select( ) · withColumn( ) · distinct( )

groupBy( ) · reduceByKey( ) · join( )

union( ) · orderBy( ) · repartition( )

ACTIONS

trigger real execution → result to driver / storage

collect( ) · count( ) · show( )

first( ) · take(n) · reduce( )

saveAsTextFile( ) · write.parquet( )

foreach( )

Lazy evaluation — the superpower

- → Spark **doesn't run** transformations when you write them — it builds a DAG (recipe) instead.
- → Only an **action** makes it execute — so Spark sees the whole pipeline before running.
- → Why it's great: Spark can **optimize globally** — combine steps, push filters early, skip unused columns, avoid pointless work.

lazy eval = fav question!

read file → filter( ) → map( ) → count( ) ⚡

↑ nothing has executed yet... just a plan

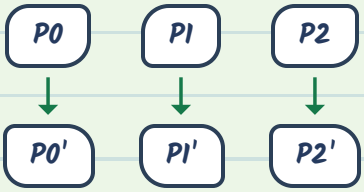
↑ action! NOW the whole job runs

CAREFUL ⚠ — **collect( )** pulls ALL data to the driver. On big data ⇒ OutOfMemory crash. Use show( ) / take(n) to peek instead!

Narrow vs wide ★

— aka "what is a shuffle?"

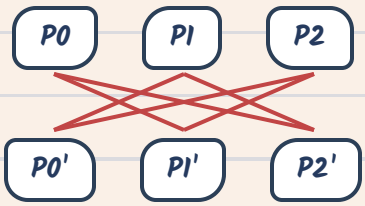
NARROW ✓ fast



each parent partition feeds **exactly one** child — no data movement

map · filter · flatMap · union

WIDE ⚠ shuffle!



children need data from **many** parents — rows regrouped by key across the network

groupBy · join · distinct · orderBy · repartition

Why shuffles hurt

✗ Data is **written to disk** on the map side, **sent over the network**, read back on the reduce side ⇒ slowest thing Spark does.

✗ Every shuffle creates a stage boundary in the DAG (pg 10) — the next stage must wait.

✓ So: filter early, avoid needless **groupBy / distinct / orderBy**, broadcast small tables (pg 17).

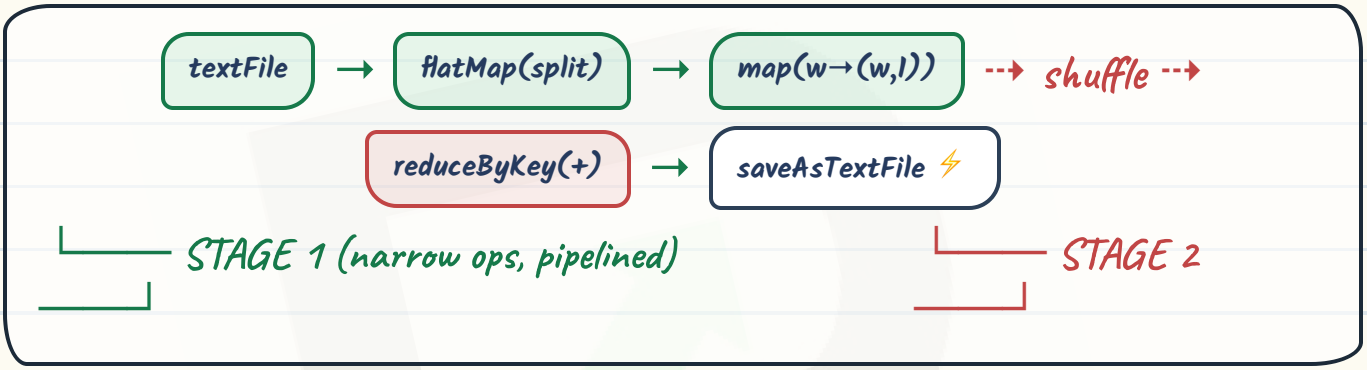
CLASSIC Q. reduceByKey vs groupByKey? → **reduceByKey** combines values on each node first (map-side combine) ⇒ far less data shuffled. Prefer it!

DAG → jobs → stages → tasks

DAG = Directed Acyclic Graph — Spark's execution plan: operators as nodes, data flow as arrows, no cycles. Built lazily from your code.

APPLICATION one spark-submit → **JOB** one per action → **STAGE** split at each shuffle → **TASK** one per partition

Worked example — word count



- **DAG scheduler** — cuts the DAG at wide dependencies into stages; inside a stage, narrow ops are pipelined together (map+filter run as one pass!).
- **Task scheduler** — packages each stage into tasks (= #partitions) and launches them on executors; retries failed tasks.
- Stage 1 has 8 partitions ⇒ **8 tasks**. Cluster has 8 free cores ⇒ all run at once. ✓

DEBUG TIP ✎ — open the **Spark UI (port 4040)** → see jobs, stages, tasks, shuffle sizes & the DAG visualization. First place to look when slow!

DataFrames & Datasets — structure = speed

DataFrame = a distributed table — rows + named, typed columns (a schema). Think SQL table / pandas df, but partitioned across a cluster. Built on RDDs.

- → Because Spark **knows the schema**, it can optimize your query (Catalyst, pg 12) and pack data efficiently (Tungsten, pg 13) ⇒ much faster than raw RDD code.
- → **Dataset** = DataFrame + compile-time type safety. Scala/Java only — **Python is dynamic**, so PySpark has DataFrames only (a DF is really Dataset[Row]).
- → Same engine underneath: DF ops compile down to RDDs of binary rows.

RDD vs DataFrame vs Dataset

	RDD	DataFrame ⚡	Dataset
Schema?	✗ raw objects	✓ named columns	✓ + JVM types
Catalyst optimizer	✗ you optimize	✓ automatic	✓ automatic
Type safety	compile-time	runtime only	compile-time
Use when...	low-level control, unstructured data	default choice — esp. PySpark	Scala + strict types

```
# same intent, two styles —
rdd.map(lambda x: (x.dept, x.sal)).groupByKey().mapValues(avg_fn)    # RDD: how
df.groupBy("dept").agg(avg("sal"))                                   # DF: what ✓
```

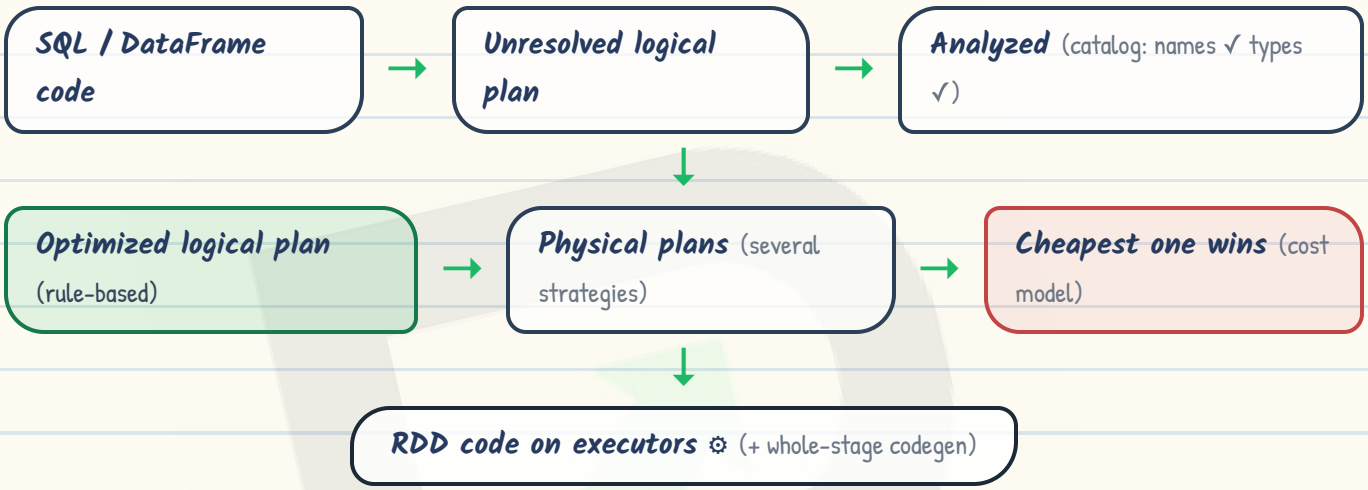
☞ *Interview line: "RDD when I need control, DataFrame when I need speed — which is almost always."*

Spark SQL & Catalyst

— you say WHAT, it decides HOW

```
df.createOrReplaceTempView("employees")
top = spark.sql("SELECT dept, avg(salary) AS avg_sal FROM employees GROUP BY dept")
# SQL string or DataFrame API — BOTH land in the same optimizer ↓
```

The Catalyst pipeline (say it in order!)



learn the pipeline order!

Favourite optimizations (name-drop these ✍)

① **Predicate pushdown** — filters run as early as possible, even inside the file reader (parquet skips whole row-groups!).

② **Column pruning** — reads only columns you actually use. select 2 of 200 cols ⇒ tiny I/O.

③ **Constant folding** — pre-computes literals (1+2 → 3) once, not per row.

④ **Join reordering / broadcast choice** — picks the cheapest join strategy (pg 17).

TRY IT ✎ — **df.explain(True)** prints all 4 plans. Spark 3.x adds **AQE**: re-optimizes at runtime using real stage statistics (pg 20).

Tungsten & memory — why DataFrames fly

Project Tungsten = the execution engine

- **Binary row format** — rows stored as compact bytes (UnsafeRow), not JVM objects ⇒ way less memory + almost no garbage-collection pressure.
- **Off-heap memory** — Spark manages its own memory outside the JVM heap when enabled.
- **Whole-stage codegen** — fuses a stage's operators into one generated Java function — no virtual calls, CPU-cache friendly, vectorized reads.

Executor memory — the split ✍️

Reserved	User memory	Unified memory ~60% (spark.memory.fraction)
~300 MB	~40% — your objects, UDF data	execution (shuffles, joins, sorts) ⇄ storage (cache) — borrow from each other!

spark.executor.memory = the whole box (+ memoryOverhead for off-heap/Python)

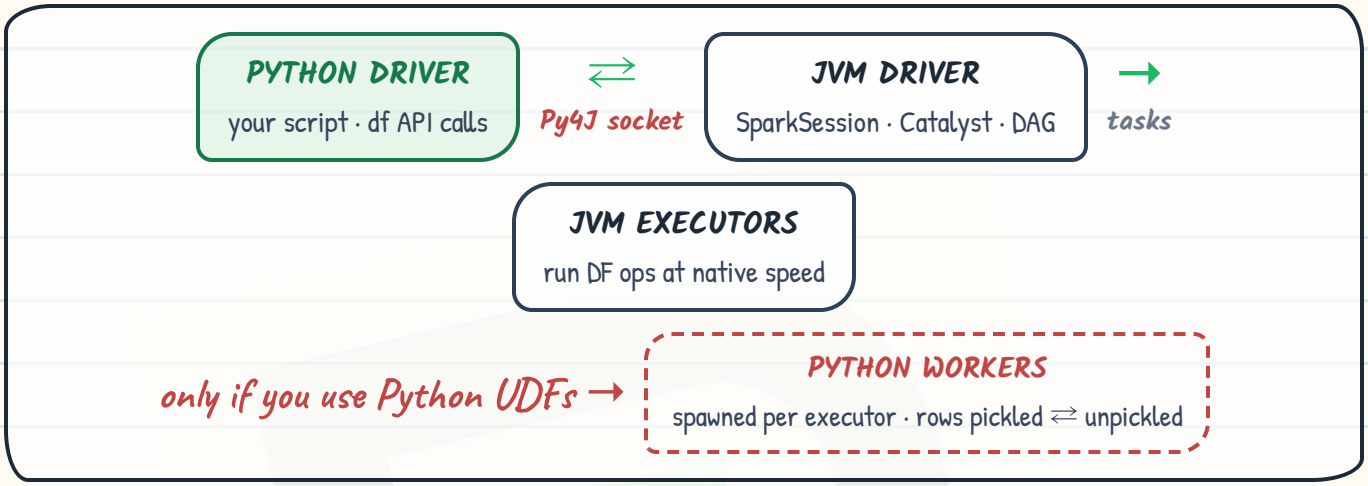
execution evicts cache if starving ✓

- **Execution & storage share** the unified pool — boundary moves at runtime (storage can be evicted down to a floor).
- Doesn't fit? Spark **spills to disk** — job survives, just slower. Frequent spills in Spark UI = raise memory or partitions.
- OOM usually = **skewed partition** or giant collect( ), not "Spark is broken".

ONE-LINER ✎ — "Catalyst optimizes the plan, Tungsten optimizes the execution — plan smart, run lean."

PySpark — how it works ★

PySpark = the Python API for Spark. Your Python code remote-controls a JVM — Spark itself still runs in Java/Scala land!



The key insight (interviewers love this)

✓ **DataFrame / SQL ops** — Python just sends the plan; execution is 100% JVM ⇒ same speed as Scala.

✗ **Python UDFs / RDD lambdas** — every row is serialized to a Python process and back ⇒ slow. Avoid when a built-in function exists!

✓ Need custom Python? Use **pandas UDFs** (@pandas_udf) — Apache **Arrow** moves data in columnar batches, vectorized with pandas/numpy ⇒ 10-100x faster than row UDFs.

```
slow ✗ udf(lambda s: s.upper())          # row-by-row picklingfast ✓ upper(col("name"))
```

PySpark in practice — the code you'll write daily

1 · START — SparkSession = entry point (Spark 2.0+, wraps SparkContext)

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, avg, count, desc

spark = (SparkSession.builder
        .appName("prepnplaced_demo")
        .master("local[*]")          # all local cores; on cluster → yarn/k8s
        .getOrCreate())
```

2 · READ — lazy! schema comes from file or inference

```
emp = spark.read.csv("emp.csv", header=True, sc-camel-infer-schema=True)
logs = spark.read.parquet("s3://bucket/logs/")  # parquet = best friend ✓
emp.printSchema();  emp.show(5)
```

3 · TRANSFORM — chain, chain, chain (all lazy)

```
result = (emp
  .filter(col("salary") > 50000)            # rows
  .withColumn("bonus", col("salary") * 0.10)    # new column
  .groupBy("dept")
  .agg(avg("salary").alias("avg_sal"),
       count("*").alias("headcount"))
  .orderBy(desc("avg_sal")))
```

4 · ACTION & WRITE — now it actually runs ⚡

```
result.show()
(result.write.mode("overwrite")
  .partitionBy("dept")                  # folder per dept (pg 16)
  .parquet("out/dept_stats/"))
spark.stop()
```

SQL twin ✍️ — same thing: emp.createOrReplaceTempView("emp") then spark.sql("SELECT dept, AVG(salary)… GROUP BY dept")

GOTCHA ⚠ — inferSchema scans the file (extra pass). Production ⇒ define schema explicitly with StructType.

Partitioning — the #1 tuning lever

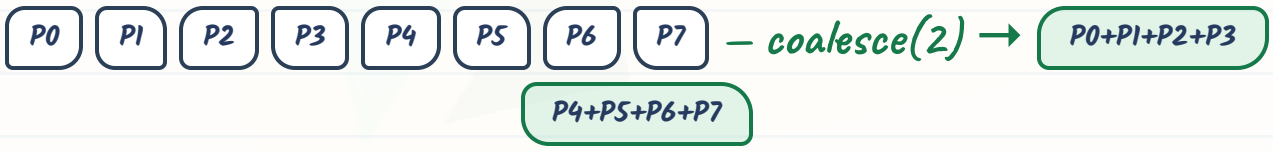
A **partition** = one chunk of your data = one task = one core working. Partitions ARE your parallelism.

- **Too few** ⇒ idle cores, giant slow tasks, OOM risk. **Too many** ⇒ scheduling overhead, tiny files.
- Defaults: reading files ⇒ ~1 partition per **128 MB** block; after any shuffle ⇒ **spark.sql.shuffle.partitions = 200** (tune this — 200 is rarely right!). AQE in Spark 3 coalesces them automatically.
- Rule of thumb: 2-4 × total cores, each partition ~100-200 MB.

repartition( ) vs coalesce( ) — classic! ★

repartition(n)
- **full shuffle** — expensive
- count can go **up or down**
- result: **evenly balanced** ✓
- can hash by column: repartition("dept")

coalesce(n)
- **no shuffle** — just merges neighbours (narrow)
- count can only go **down**
- cheap ✓ but can be **uneven** ⚠
- classic use: shrink before writing files



local merges only — no rows cross the network ✓

Partitioned writes = free speed later

```
df.write.partitionBy("year", "month").parquet("sales/")  # → sales/year=2026/month=07/...
spark.read.parquet("sales/").filter(col("year") == 2026)  # reads ONLY that folder — partition pruning ✓
```

Joins & strategies ★

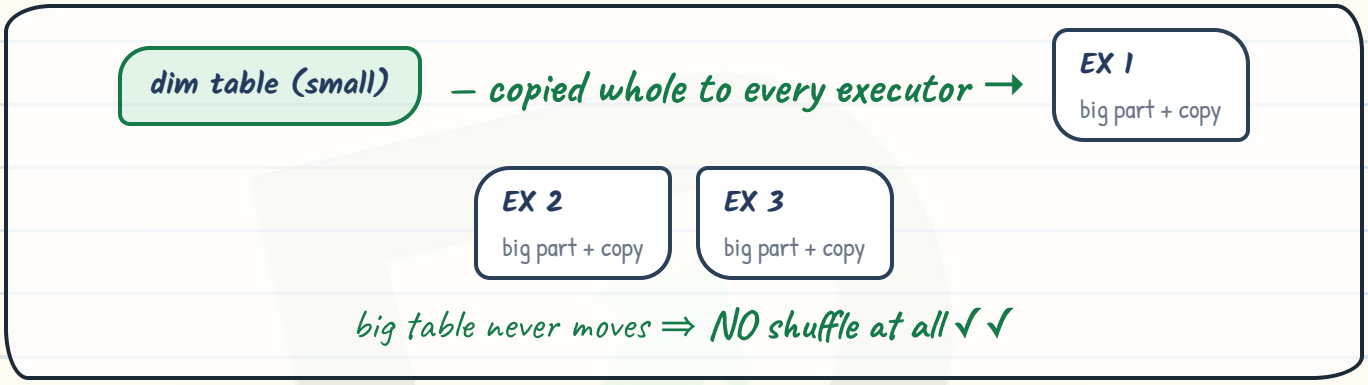
— where jobs go to die 💀 (kidding... mostly)

① Shuffle sort-merge join — the default (big ⋈ big)

→ Both tables are **shuffled by the join key** (matching keys land on the same partition), **sorted**, then merged like a zipper.

✗ Cost: TWO full shuffles ⇒ network heavy. Reliable, scales to huge tables.

② Broadcast hash join — small ⋈ big = the fast one ⚡




```
from pyspark.sql.functions import broadcast
sales.join(broadcast(dim_stores), "store_id")      # force it with a hint# auto when table < spark.sql.autoBr
```

③ The villain: skewed joins

✗ One hot key (e.g. customer_id = null, or "Mumbai") ⇒ one monster partition ⇒ 199 tasks done, 1 task runs forever.

✓ Fixes: **AQE skew-join** splits fat partitions automatically (Spark 3+) · **salting** — add a random suffix to the hot key, join on (key, salt), aggregate after · filter/handle nulls first.

CHEAT ✎ — small⋈big → broadcast · big⋈big → sort-merge · skewed → AQE/salt. Check which ran: **df.explain( )**.

Caching & persistence

— stop recomputing everything!

→ **Problem:** every action re-runs the whole lineage from scratch. Use a df in 3 actions ⇒ it's computed 3 times.

→ **Fix:** df.cache( ) — keep the computed result on the executors, reuse it across actions.

→ Caching is **lazy** too — materialized on the first action after cache( ).

```
clean = raw.filter(...).join(...).withColumn(...)   # expensive pipeline
clean.cache()                                       # mark it
clean.count()                                       # materialize once
clean.groupBy("city").count().show()                # fast — reads cache ✓
clean.write.parquet("out/")                         # fast — reads cache ✓
clean.unpersist()                                   # free the memory when done!
```

Storage levels — persist(level)

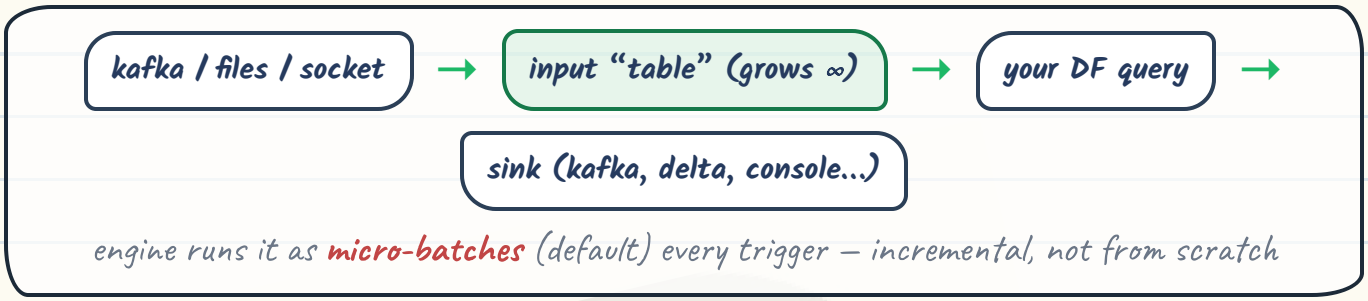
MEMORY_ONLY	fastest; partitions that don't fit are recomputed (RDD cache( ) default)
MEMORY_AND_DISK ✓	spill extras to disk (**DataFrame cache( ) default**)
MEMORY_ONLY_SER	serialized — smaller, bit more CPU (RDD/Java side)
DISK_ONLY	huge data, still cheaper than recomputing
..._2 (replicated)	2 copies on different nodes — survives an executor dying

cache vs checkpoint — cache keeps lineage (recompute possible); **checkpoint( )** writes to reliable storage & cuts lineage — for iterative jobs with huge DAGs.

WHEN ⚠ — cache only what's reused ≥ 2 times and expensive. Caching everything = wasted memory = slower jobs!

Structured Streaming — batch code, live data

BIG IDEA: treat the stream as an unbounded table — every new event = a new row appended. Then just write normal DataFrame code on it!




```
events = (spark.readStream.format("kafka")
  .option("kafka.bootstrap.servers", "broker:9092")
  .option("subscribe", "orders").load())

counts = (events.groupBy(window(col("ts"), "10 minutes"), col("city"))
  .count()
  .withWatermark("ts", "15 minutes"))          # tolerate late data 15 min

(counts.writeStream.outputMode("update")
  .option("checkpointLocation", "chk/orders/")  # crash recovery!
  .format("console").start().awaitTermination())
```

append	update	complete
only new rows → sink (default)	only rows whose aggregate changed	whole result table every time

→ **Watermark** = "how late can data be?" — lets Spark drop old window state instead of remembering forever.

→ **Exactly-once** = checkpointing + replayable source (Kafka) + idempotent/transactional sink.

Optimization + cheat sheet ★

- ✓ Prefer **DataFrame API + built-ins** over RDDs & Python UDFs
- ✓ **Broadcast** small dimension tables in joins
- ✓ **cache( )** data reused ≥2×, unpersist after
- ✓ Never **collect( )** big data to the driver
- ✓ **Filter & select early** — let pushdown/pruning shrink I/O
- ✓ Tune **shuffle partitions**; enable **AQE** (auto-coalesce + skew fix)
- ✓ Store as **parquet**, partitionBy high-use filter columns
- ✓ Watch the **Spark UI** — spills, skew, shuffle sizes tell the story

Rapid-fire revision — cover the right side & test yourself!

driver vs executor?	driver plans & coordinates; executors run tasks & hold cache
transformation vs action?	lazy plan-building vs actually executing the job
narrow vs wide?	1-to-1 partition flow vs shuffle across the network (stage boundary)
repartition vs coalesce?	full shuffle & even vs no shuffle, merge-down only
why is DF faster than RDD?	schema ⇒ Catalyst optimizes the plan, Tungsten packs binary rows + codegen
cache vs checkpoint?	memory + lineage kept vs reliable storage + lineage cut
how does PySpark run?	Python drives the JVM via Py4J; DF ops run in JVM, UDFs need Python workers

All the best — go crack it! ⚡

revise ✍️ · practice ⌨ · get placed ✓ — Team Prepnplaced