Hadoop & Apache Spark — Q/A Study Sheet

# Hadoop Q/A

## 1) What are Hadoop distributions?

Packaged, pre-integrated stacks including Hadoop plus ecosystem tools, installers, management, and support.

* Apache Hadoop (vanilla OSS)
* Cloudera: legacy CDH; current CDP (after Cloudera+Hortonworks merger)
* Hortonworks HDP (rolled into CDP)
* MapR (now HPE Ezmeral Data Fabric)
* Cloud-managed: AWS EMR, GCP Dataproc, Azure HDInsight

## 2) Difference between CDH and CDP?

* CDH: older, primarily on‑prem Hadoop distro (HDFS/YARN/Hive/Impala/etc.), cluster‑centric ops.
* CDP (Cloudera Data Platform): successor to CDH+HDP; hybrid on‑prem & cloud, SDX security/governance, autoscaling, data services for DW/ML, better batch + streaming.

## 3) Hadoop architecture (big picture)

Hadoop = Storage (HDFS) + Cluster resources (YARN) + Compute (MapReduce/Spark).

* HDFS: NameNode (metadata), DataNodes (store blocks), Standby NN (HA), Secondary NN (checkpointing).
* YARN: ResourceManager (Scheduler + ApplicationsManager), NodeManagers, ApplicationMaster (per app).
* Compute: MapReduce (older) / Spark (newer).

## 4) Configuration files used during Hadoop installation

* core-site.xml — FS configs (fs.defaultFS), IO, RPC timeouts
* hdfs-site.xml — HDFS (dfs.replication, NN/DN dirs, HA/JN settings)
* yarn-site.xml — RM/NM addresses, container resources/limits
* mapred-site.xml — MapReduce on YARN (history server, speculative flags)
* hadoop-env.sh — JAVA\_HOME, heaps, JVM opts
* workers (slaves) — DN/NM host list
* capacity-scheduler.xml or fair-scheduler.xml — if using those schedulers

## 5) Difference between `hadoop fs` and `hdfs dfs`

* `hadoop fs` — generic shell for any Hadoop‑compatible filesystem (HDFS, S3, etc.).
* `hdfs dfs` — HDFS‑specific shell; targets HDFS by default.

## 6) Difference between Hadoop 2 and Hadoop 3

* Erasure Coding in HDFS (H3) for cold data; big storage savings vs 3× replication.
* YARN improvements: opportunistic containers, Timeline Service v2, better container integration.
* HDFS: router‑based federation (global namespace), intra‑DN balancer.
* Java/deps modernization; rolling upgrades, stability.

## 7) Replication factor — what & why?

Number of replicas per HDFS block (default 3). Ensures fault tolerance and availability; NN maintains target replication via re‑replication.

## 8) What if a DataNode fails?

NameNode marks it dead after missed heartbeats; under‑replicated blocks are re‑replicated to healthy nodes (rack‑aware); pipelines reroute; returning DN re‑registers.

## 9) What if the NameNode fails?

* Without HA: cluster becomes unavailable (SPOF).
* With HA: Active/Standby NNs + JournalNodes + ZKFC handle automatic failover.
* Secondary NameNode is not a hot standby; it only checkpoints.

## 10) Why block size ~128 MB? What if changed?

* Large blocks reduce seeks & metadata load → high sequential throughput.
* Increase (256–512 MB): fewer blocks, less metadata, fewer mappers; less parallelism.
* Decrease (64 MB): more parallelism but more metadata & seeks.

## 11) Small files problem

Millions of tiny files overload NameNode memory and hurt throughput (seek‑heavy).

* Ingest/compact to large Parquet/ORC; Hive compaction
* CombineFileInputFormat, Spark coalesce/repartition
* Use HBase for many small records with random access
* HDFS concat, DistCp+merge; consider object storage or Ozone

## 12) Rack awareness

NN places replicas across racks to survive rack/network failures; with RF=3: local rack, plus at least one replica on another rack.

## 13) What is SPOF? Resolution?

* Single Point of Failure — component whose failure halts the system.
* Historically: NameNode, ResourceManager.
* Resolved with HA: Active/Standby + ZooKeeper‑based failover.

## 14) ZooKeeper (in Hadoop context)

Highly available coordination service for leader election, service discovery, failover (e.g., NN‑HA via ZKFC, RM‑HA); also used by HBase/Kafka.

## 15) `-put` vs `-copyFromLocal`

Synonyms in HDFS CLI (local → HDFS upload). `-get` ≈ `-copyToLocal`.

## 16) Erasure coding (EC)

* Parity‑based data protection (e.g., RS(6,3)).
* Pros: ~1.5× storage overhead vs 3× replication (saves space).
* Cons: CPU/network cost on rebuild; not ideal for hot/small files.

## 17) Speculative execution

Duplicate slow (straggler) tasks; whichever finishes first wins. Good on noisy clusters; avoid with non‑idempotent side‑effects.

## 18) YARN architecture

* ResourceManager (Scheduler + ApplicationsManager)
* NodeManagers on each node (launch containers, report health)
* ApplicationMaster per app (requests containers, orchestrates tasks)

## 19) Applications Manager vs Application Master

* ApplicationsManager (in RM): accepts submissions, allocates first container for AM, tracks lifecycle.
* ApplicationMaster (per app): negotiates containers with RM, schedules tasks on NMs.

## 20) MapReduce working (high level)

* Input split → RecordReader → map(k1,v1) → (k2,v2)
* Partition by key → Shuffle & Sort
* Optional Combiner on mapper output
* Reduce(k2, list[v2]) → OutputFormat writes

## 21) How many mappers for a 1 GB file?

Approximately file\_size / split\_size. With 128 MB splits → ~8 mappers. Non‑splittable compression can reduce this.

## 22) How many reducers for a 1 GB file?

Not set by input size; configured via job (e.g., mapreduce.job.reduces) or engine heuristics. Default MR often 1 if not set.

## 23) Combiner — what/why/where

Local mini‑reduce on mapper output to cut shuffle volume; requires associative & commutative functions (e.g., sums, min/max).

## 24) Partitioner — what/why/where

Decides which reducer a key goes to (default HashPartitioner). Use custom/range partitioners for balance, order, or domain routing.

## 25) `hadoop fs -put` vs `hdfs dfs -put` in practice

Same operation for HDFS upload; use `hdfs dfs` for HDFS‑only scripts; `hadoop fs` if you might target other filesystems via scheme.

# Apache Spark Q/A

## 1) Advantages of Spark over MapReduce

* In‑memory processing (orders faster for iterative workloads)
* Rich unified APIs (SQL/DF/ML/Graph/Streaming)
* Catalyst optimizer + DAG scheduling
* Interactive shells; pipelines; caching
* Batch & streaming on one engine

## 2) Spark architecture (high level)

* Driver: builds logical → optimized physical plan; schedules jobs/stages/tasks
* Cluster Manager: allocates resources (Standalone/YARN/Kubernetes)
* Executors: run tasks, cache data, shuffle; report to driver
* External Shuffle Service moves data across executors (when enabled)

## 3) YARN architecture (Spark on YARN)

* ResourceManager (Scheduler + ApplicationsManager)
* NodeManagers run containers
* ApplicationMaster (per Spark app) negotiates executors; driver runs in client or cluster mode

## 4) What is a cluster manager? Which ones are used?

Allocator of CPU/memory for apps. Options: Standalone, YARN, Kubernetes (Mesos legacy). Choose based on environment (on‑prem vs cloud‑native).

## 5) SparkContext vs SparkSession

* SparkSession (Spark 2+): unified entry point (SQL, catalog, streaming).
* SparkContext: lower‑level RDD API; accessible via spark.sparkContext. Prefer SparkSession.

## 6) Spark execution modes

* Local (dev)
* Client (driver where you submit; executors on cluster)
* Cluster (driver inside cluster for resilience)

## 7) DataFrame vs RDD — when use RDD?

* DF: schema, optimized by Catalyst, vectorized → faster & concise.
* RDD: low‑level control; use for custom partitioners, unstructured/complex transforms, or when you need per‑record control.

## 8) Transformations vs Actions

* Transformations (lazy): select, filter, map, groupBy, join
* Actions (trigger execution): count, show, collect, write.save

## 9) Narrow vs Wide transformations

* Narrow: each output partition from one input partition (map, filter)
* Wide: require shuffle across partitions (groupByKey, join, distinct)

## 10) Lazy evaluation

Spark builds a DAG from transformations; executes only upon an action, enabling global optimization.

## 11) map vs flatMap

* map: A→B (1→1)
* flatMap: A→Iterable[B] then flattens (1→0..n); use for tokenization/explodes

## 12) What is a DAG?

Directed Acyclic Graph of transformations/stages representing dependencies for scheduling and optimization.

## 13) What is lineage?

Transformation history for a dataset; used for fault recovery (recompute lost partitions).

## 14) DAG vs Lineage

DAG is the execution graph for an action; lineage is the per‑dataset history enabling recomputation.

## 15) What happens when you submit a Spark job?

* Driver creates SparkSession → logical plan → Catalyst optimization → physical plan
* Requests executors from cluster manager
* Action triggers job → split into stages at shuffles → tasks per partition
* Executors run tasks, spill/shuffle as needed; results to driver/sinks

## 16) Client vs Cluster mode — when to use

* Client: driver near data/user; good for dev/interactive.
* Cluster: driver in cluster; good for prod, resilient to client node failure.

## 17) DataFrame vs Dataset

* Scala/Java: DataFrame = Dataset[Row] (untyped), Dataset[T] is typed with Encoders.
* Python: no Dataset API; use DataFrame.

## 18) Pandas DF vs Spark DF

* Pandas: single‑node, in‑memory, Python ops.
* Spark: distributed, lazy, optimized; Arrow accelerates Pandas↔Spark exchange.

## 19) Coalesce vs Repartition — when to use

* coalesce(n): reduce partitions without shuffle; quick shrink; may be imbalanced.
* repartition(n or by cols): full shuffle for balanced partitions or key‑based partitioning.

## 20) If both reduce partitions, which to choose?

Use coalesce to quickly shrink when balance isn’t critical; use repartition when even balance is important.

## 21) When to reduce partitions?

* After heavy filters/aggregations leaving many near‑empty partitions
* Before writing to avoid many small files

## 22) When to increase partitions?

* Before wide ops on large data to improve parallelism
* When task queues are long relative to available cores

## 23) For a 1 GB file, how many partitions?

With 128 MB input splits, ≈ 8 partitions. Non‑splittable compression may yield fewer (even 1). Verify via df.rdd.getNumPartitions().

## 24) Jobs, stages, tasks — how many?

* Jobs: one per action
* Stages: separated by shuffle boundaries
* Tasks: one per partition per stage

## 25) What is a driver? Driver‑side examples

* Driver runs your main program, builds plans, coordinates jobs
* Examples: explain(), collect()/take(), creating broadcast variables/accumulators, logging

## 26) What is an executor? Executor‑side examples

* Workers running tasks, executing UDFs, shuffles/aggregations, reading/writing blocks
* Your map/flatMap/UDF code runs here

## 27) When to use a broadcast join?

When one side is small enough to fit in memory on each executor, to avoid shuffling the large table.

## 28) Broadcast variable — what & why faster?

Read‑only value shipped once per executor (cached), not with every task → less serialization/network overhead. Good for lookups.

## 29) cache() vs persist()

* cache() = persist(MEMORY\_ONLY)
* persist(level) chooses tiers: MEMORY\_AND\_DISK, DISK\_ONLY, serialized, etc.; unpersist() to free

## 30) What’s a shuffle?

Data redistribution across partitions (joins, groupBy). Expensive due to network I/O, sort, disk spill, serialization.

## 31) What is spill? How to improve performance

* Spill: when data doesn’t fit in executor memory during shuffle/sort/agg, it writes to disk.
* Improve: size executors appropriately, adjust shuffle partitions, map‑side combine, reduce data early, efficient codecs.

## 32) Spark performance tuning — methods & use cases

* Partitioning: tune spark.sql.shuffle.partitions; repartitionByRange; avoid tiny files (coalesce)
* Skew: key salting, AQE skew join, pre‑aggregation
* Joins: broadcast small side, prune columns, filter early, ensure key types match
* Caching: cache reused expensive subplans only; match storage level to memory
* Serialization & UDFs: Kryo, avoid heavy Python UDFs; use SQL/exprs
* File formats: Parquet/ORC + predicate pushdown; Snappy/ZSTD compression
* Enable AQE for adaptive coalesce/join switching

## 33) Typical challenges in Spark projects

* Data skew causing stragglers
* Executor/driver OOM from large collect()/broadcast
* Too many small files
* Schema drift and bad records
* Slow object‑storage listings; shuffle spill storms
* Python UDF serialization overhead

## 34) OOM error — reasons

* collect() on large DF to driver
* Broadcasting large tables
* Caching too much or with high storage level
* Skewed shuffles, huge partitions
* Insufficient executor memory/overhead

## 35) Data partition vs table partition

* Data partition: runtime chunks for parallelism (Spark).
* Table partition: layout of files by partition columns (Hive/Delta) enabling pruning.

## 36) If both datasets are large, optimizing a join

* Filter early; select only needed columns
* Repartition both sides by join key; ensure same partitioner
* Use bucketed tables when possible
* AQE to adapt join type and handle skew
* Pre‑aggregate to reduce size

## 37) Logical plan vs physical plan

* Logical: what to compute (analyzed/optimized by Catalyst)
* Physical: how to compute (operators, join types, shuffle counts). Use explain("extended").

## 38) What is an accumulator?

Write‑only counters from executors aggregated on driver (e.g., bad record counts). Good for monitoring, not control flow.

## 39) Spark Streaming vs Structured Streaming

* DStreams (legacy): micro‑batches of RDDs.
* Structured Streaming: DF/SQL engine with Catalyst, exactly‑once sinks, watermarking, stateful ops, event‑time.

## 40) Dynamic Partition Pruning (DPP)

At runtime, engine prunes partitions of large fact tables using join filter values from the dimension side, cutting scan I/O.

## 41) Adaptive Query Execution (AQE)

* Coalesces shuffle partitions at runtime
* Switches join types (e.g., to broadcast) based on observed sizes
* Handles skew by splitting skewed partitions

## 42) File formats: Parquet, Avro, CSV, JSON — pros/cons

* Parquet/ORC: columnar, compressed, predicate pushdown → best for analytics
* Avro: row‑based with schema evolution; good for logs/interop
* CSV: human‑readable; no schema; large and slow
* JSON: flexible semi‑structured; larger and slower; good as landing format—convert to Parquet/ORC for query

## 43) Compression formats & specialties

* Snappy: fast, common default with Parquet/ORC
* Gzip: high ratio, not splittable
* Bzip2: splittable, slower
* LZ4: very fast
* ZSTD: excellent ratio & speed (newer)

## 44) How Spark memory management works (high level)

Unified Memory Manager divides executor memory into execution (shuffle, join, sort) and storage (cache/broadcast). Storage can evict. Tune executor memory/cores/overhead; avoid huge per‑task footprints.

## 45) How many stages and tasks are created

Stages are segments between shuffles; tasks = partitions per stage. A wide op typically creates a new stage; tasks equal current partition count.

## 46) How are executors created? How to size them?

* Cluster manager launches them per driver request (num executors or dynamic allocation)
* Size via --executor-cores, --executor-memory, and overhead
* Use Spark UI/logs to assess GC/spill/time and tune empirically

## 47) spark-submit — common parameters

* --master, --deploy-mode, --name
* --conf k=v, --packages/--jars/--py-files, --files/--archives
* --class (JVM apps)
* Resources: --num-executors (YARN), --executor-cores, --executor-memory, --driver-memory, --queue
* Kubernetes: namespace, service account, image, etc.

## 48) What is data skew? How to fix it?

* Few hot keys dominate; causes stragglers and OOM
* Fix: key salting, AQE skew join, pre‑aggregations, custom partitioner, increase parallelism

## 49) Key salting — idea & use case (pseudo‑code)

Add a small random/range suffix to hot keys to spread load; aggregate twice (by salted key then original key).

## 50) Role of checkpointing in Spark & streaming

* RDD/DF checkpoint cuts lineage depth to avoid recomputation storms; needs checkpoint dir
* Structured Streaming checkpoint stores offsets/state for recovery & exactly‑once semantics

## 51) For 1 GB file: partitions and practicality

Rule of thumb: ~8 partitions with 128 MB splits; validate with getNumPartitions(); adjust for compression and throughput goals.

## 52) For any program: how many jobs, stages, tasks?

Jobs: per action; stages: per shuffle boundary; tasks: per partition per stage. Example pipeline with join+agg typically ≥3 stages.