1) Why was Spark introduced if Hadoop MapReduce already exists?

Answer:

- MapReduce becomes **expensive for iterative workloads** because it writes intermediate results to disk each iteration.
- It **lacks efficient data sharing** across steps/jobs (disk I/O becomes the bottleneck).
- Spark supports **in-memory reuse**, so repeated computations become much faster.
  Example: Machine learning training (10 iterations) is slow in MapReduce (disk each round) but fast in Spark by caching.

---

2) Explain “Spark uses memory instead of disk” and why it matters.

Answer:

- Spark keeps intermediate datasets in **RAM** (in-memory data sharing).
- Disk writes are avoided unless needed → **lower latency** and faster iteration.
- Fault tolerance still works via **recompute using lineage**, so it’s not “unsafe.”
  Example: PageRank repeatedly joins ranks with links—Spark caches ranks to avoid re-reading/writing every loop.

---

3) Spark vs Hadoop MapReduce: give 3 differences (performance + model).

Answer:

- Spark is usually faster because it can process **in-memory**, while MapReduce persists after map/reduce to disk.
- Spark uses **DAG + lazy evaluation**, MapReduce is more linear map→shuffle→reduce.
- Spark may **degrade** when memory is insufficient; MapReduce is better for simple one-pass jobs and coexisting services.
  Example: One-time log scan (single pass) can be OK in MapReduce; iterative ML is best in Spark.

---

4) What is a Spark cluster? Name its components and roles.

Answer:

- A Spark cluster is a group of nodes executing Spark apps **in parallel**.
- **Driver** coordinates the job and schedules tasks.
- **Cluster Manager** allocates resources (executors/cores/memory), and **Executors** run tasks.
  Example: On YARN, cluster manager launches containers for executors across machines.

---

5) Define Driver and give 3 responsibilities.

Answer:

- Runs SparkContext/SparkSession and acts as the **brain** of the job.
- Converts code into a **DAG / execution plan**.
- Requests resources and **schedules tasks** on executors.
  Example: When you call `collect()`, the driver triggers execution and gathers results.

---

6) What is a Cluster Manager? List 4 types.

Answer:

- It launches executors and **allocates CPU/memory** to the Spark app.
- Monitors health/resource usage and manages worker availability.
- Types: **Standalone, YARN, Mesos, Kubernetes**.
  Example: Kubernetes runs executors as containers; YARN runs them as Hadoop containers.

---

7) What are Executors? Give 3 key points.

Answer:

- Executor is a process (JVM) on worker nodes that **runs tasks** on partitions.
- Performs transformations and can **store intermediate results in memory** for reuse.

- Sends results back to the driver or writes to storage.
  Example: An executor computes partial sums per partition before a shuffle merge.

---

8) Explain the Spark execution hierarchy: Job → Stage → Task.

Answer:

- **Job** is triggered by an action (e.g., `count()`, `collect()`).
- Spark divides the DAG into **stages based on shuffle boundaries**.
- Each stage is split into **tasks (one per partition)**.
  Example: A `reduceByKey()` typically creates a shuffle → new stage boundary.

---

9) What is a DAG in Spark and why is it important?

Answer:

- DAG tracks dependencies/lineage of transformations (no cycles).
- Helps Spark optimize execution into stages and tasks.
- Enables fault tolerance by recomputing lost partitions from lineage.
  Example: If a node dies, Spark rebuilds the missing partition by replaying DAG steps.

---

10) What is a Partition in Spark? Why is it "core idea"?

Answer:

- Partition is the **smallest logical chunk** of an RDD/DataFrame.
- Each partition is processed independently by **one task**.
- Partitioning enables parallelism; too many partitions add scheduling overhead.
  Example: 1 TB split into 100 partitions lets many tasks run in parallel.

---

11) If a stage has 10 partitions, how many tasks are created? Explain.

Answer:

- Number of tasks in a stage = **number of partitions**.
- So **10 partitions** → **10 tasks**.
- Tasks run in parallel depending on available cores/executors.
  Example: With 5 cores available, 10 tasks run in two waves.

---

12) Define Shuffle and explain when it happens.

Answer:

- Shuffle is **redistributing data across partitions** so same keys end up together.
- Happens **between stages**, especially for join/group/aggregate.
- It's expensive due to network + disk/memory pressure.
  Example: `join()` forces shuffling because matching keys may live on different executors.

---

13) Name 5 transformations that trigger shuffle and why.

Answer:

- `reduceByKey()` → needs all values for key in one place.
- `groupByKey()` → groups values per key globally.
- `join()` → matches keys from two datasets.
- `distinct()` → removes duplicates globally.
- `sortByKey()` / `repartition()` → requires global ordering / redistribution.
  Example: `distinct()` must compare across partitions to ensure duplicates removed everywhere.

---

14) Narrow vs Wide transformations: define and compare.

Answer:

- Narrow: each output partition depends on **one** input partition (no shuffle).
- Wide: output depends on **multiple** input partitions (requires shuffle).

- Wide is slower due to network I/O and stage boundary creation.
  Example: `map()` is narrow; `groupByKey()` is wide.

---

15) Why is `groupByKey()` discouraged? Give the "under the hood" reason.

Answer:

- It shuffles **all values** for each key across network (huge movement).
- High memory overhead because it must hold a list of values before combining.
- Causes slow stages and possible OOM for skewed keys.
  Example: Key "Karachi" has millions of records → `groupByKey()` gathers all, may crash.

---

16) Why is `reduceByKey()` better than `groupByKey()`?

Answer:

- Performs **local aggregation first** within partitions.
- Shuffles only **partial results**, reducing network traffic.
- More scalable for large key groups.
  Example: Counting words → use `reduceByKey((a,b)=>a+b)` not `groupByKey().mapValues(sum)`.

---

17) Define RDD and list 4 properties (exactly like Spark expects).

Answer:

- RDD is a fault-tolerant distributed collection operated in parallel.
- **Immutable** (transformations create new RDDs).
- **Distributed** (partitioned across cluster).
- **Lazy evaluated** (runs only when an action occurs) and **resilient** via lineage.
  Example: `rdd.filter(...).map(...)` doesn't run until `count()` is called.

---

18) What is “lazy evaluation” and why is it useful?

Answer:

- Transformations build a plan but **don’t execute immediately**.
- Execution happens only when an **action** is called.
- Allows Spark to optimize pipelines and reduce unnecessary work.
  Example: Two filters can be combined before running, saving one full pass.

---

19) Differentiate transformations vs actions with examples.

Answer:

- Transformations create new RDDs (lazy): `map`, `filter`, `reduceByKey`.
- Actions return results or write output (execute now): `count`, `collect`, `saveAsTextFile`.
- Actions trigger the full DAG execution.
  Example: `rdd.map(...).filter(...)` is lazy; `rdd.count()` triggers computation.

---

20) Why can `collect()` be dangerous?

Answer:

- It brings **all data to the driver**.
- Driver memory may overflow → crash or hang.
- Better alternatives: `take(n)`, `count()`, `saveAsTextFile()`.
  Example: Collecting 200GB dataset to driver laptop will crash.

---

21) What is caching/persistence in Spark and why needed for iterative algorithms?

Answer:

- Caching stores computed RDD partitions in memory across executors.
- Without caching, each iteration recomputes previous transformations + reloads from disk.
- Big speedup for ML/graph algorithms using same dataset repeatedly.
  Example: Logistic regression benefits greatly by caching training points.

---

22) Compare these storage levels: MEMORY_ONLY vs MEMORY_AND_DISK vs DISK_ONLY.

Answer:

- MEMORY_ONLY: store in RAM; recompute if lost.
- MEMORY_AND_DISK: keep in RAM, spill overflow to disk.
- DISK_ONLY: store only on disk; slower but fits huge datasets.
  Example: Large dataset > RAM → use MEMORY_AND_DISK to avoid recomputation + avoid OOM.

---

23) What is RDD lineage and how does it give fault tolerance?

Answer:

- Lineage = history of transformations that created an RDD.
- Spark can recompute lost partitions by replaying lineage steps.
- No need to replicate intermediate data like MapReduce.
  Example: If executor fails, Spark rebuilds missing partition from original HDFS data + transformations.

---

24) Scenario: Driver fails vs Executor fails—what happens?

Answer:

- If **executor/worker fails**, cluster manager reassigns tasks and Spark recomputes lost partitions via lineage.
- If **driver fails**, the whole job typically restarts (unless checkpointed).

- This is why stable driver deployment (cluster mode) matters.
  Example: In YARN cluster deploy-mode, driver runs in cluster (more reliable than local client driver).

---

25) Explain "logical plan → physical plan → tasks" in Spark.

Answer:

- Driver creates a **logical plan/DAG** from your transformations.
- Plan gets optimized, converted to **physical plan** and then **tasks**.
- Tasks are sent to executors; shuffles happen between stages if required.
  Example: A `join()` creates shuffle stage; Spark generates tasks per partition per stage.

---

26) What is the "closure" Spark sends to executors?

Answer:

- Closure = the function + captured variables it needs to run.
- Driver serializes and ships closures to executors for distributed execution.
- Bad closures (huge objects) increase network overhead and memory use.
  Example: Capturing a massive lookup map in a closure is worse than broadcasting it.

---

27) Give 3 RDD optimization techniques from lecture.

Answer:

- Use `cache()`/`persist()` for iterative workloads.
- Avoid `groupByKey()`; prefer `reduceByKey()` / `aggregateByKey()`.
- Avoid large `collect()`; repartition/coalesce intelligently and use broadcast for small lookup data.
  Example: WordCount → `reduceByKey`, not `groupByKey`, and write output to HDFS instead of collect.

---

28) What does repartition do? When can it help and when can it hurt?

Answer:

- Repartition redistributes data into new number of partitions (often triggers shuffle).
- Helps balance skew, improve parallelism, and avoid single hot partition.
- Hurts if used unnecessarily: extra shuffle overhead and time.
  Example: After filtering to small dataset, repartitioning to 200 partitions wastes overhead.

---

29) Explain Spark's "generality" (unifying programming models).

Answer:

- RDDs allow unification of workloads: streaming, graph processing, ML.
- Spark has components like Spark SQL, Streaming, MLlib, GraphX.
- Reduces need to maintain separate systems for each workload.
  Example: Same cluster runs ETL + ML training + graph analytics on user interaction data.

---

30) Explain Spark SQL / DataFrames: what are DataFrames and why are they useful?

Answer:

- DataFrame = distributed dataset organized into **named columns** like a table.
- Enables higher-level operations (groupBy/join) and Spark can optimize queries.
- DataFrames still track lineage and support distributed computation.
  Example: `students.groupBy("gender").count()` is simpler than manual RDD pair operations.

---

31) RDD vs DataFrame: give 3 differences.

Answer:

- RDD is lower-level; DataFrame has **schema (columns/types)**.
- DataFrames are optimized by Spark SQL engine (Catalyst optimization).
- RDD gives more control but less automatic optimization; DataFrame is more declarative.
  Example: Complex SQL-style joins are easier and faster in DataFrames than raw RDD joins.

---

32) What is a Dataset and how is it different from DataFrame?

Answer:

- Dataset is an extension of DataFrame API with **type-safety** (compile-time checks).
- DataFrame is more like untyped rows/columns; Dataset uses typed JVM objects.
- Datasets are mainly for Scala/Java; Python/R only DataFrames.
  Example: In Scala, a wrong field type can be caught at compile time in Dataset.

---

33) Why do Python and R not support Dataset the same way as Scala/Java?

Answer:

- Python/R lack compile-time type safety checks.
- Errors are detected at **runtime**, not compile time.
- Scala/Java run on JVM with typed objects → Dataset type mismatches caught early.
  Example: Mistyping an integer column as string may compile in Python but fail during execution.

---

34) Explain “Spark with Hadoop” — list 4 integration modes.

Answer:

- Spark on **HDFS only** (HDFS as storage, Spark as compute).
- Spark on **YARN** (YARN as cluster manager).

- Spark with **Hive/HCatalog** (SQL + metadata layer).
- Spark with the **entire Hadoop stack** (HDFS + YARN + Hive + security etc.).
  Example: Enterprise big data often runs Spark on YARN with Hive Metastore tables.

---

35) Scenario: You have HDFS but no YARN. Can Spark still work? How?

Answer:

- Yes: Spark can use **HDFS as storage** layer while managing compute via Standalone/Kubernetes.
- It can still read/write from HDFS using Hadoop client libs.
- You must manage cluster resources yourself (no YARN scheduling).
  Example: A small team runs Spark standalone cluster + HDFS for storage.

---

36) "Spark sets sort record" question (numerical): compute speedup by elapsed time.

Given: Hadoop MR 72 mins, Spark 23 mins.
Answer:

- Speedup by time ≈ 72 / 23 ≈ **3.13× faster**.
- Spark did it with far fewer nodes (206 vs 2100) → efficiency is much higher.
- In-memory + optimized execution contributes to improvement.
  Example: Sort benchmark (100TB) shows Spark achieving higher throughput even on virtualized cluster.

---

37) What are Spark's main use cases? Give at least 5.

Answer:

- Streaming data, machine learning, interactive analysis.
- Data warehousing, batch processing, exploratory data analysis.
- Graph analytics, spatial (GIS) analytics.
  Example: Uber uses Kafka + Spark Streaming + HDFS for continuous ETL pipeline (real-time events).

38) Give 2 real-world company examples using Spark and explain “why Spark fits”.

Answer:

- **Uber**: continuous ETL from mobile events using Kafka + Spark Streaming + HDFS; Spark fits for near-real-time pipelines + scalable processing.
- **Netflix**: analyzes viewing habits for recommendations; Spark fits for large-scale analytics and ML pipelines.
- Key reasons: scalability, in-memory speed, unified analytics/ML stack.
  Example: Recommendation engines need repeated model training → caching helps.

---

39) When should you NOT use Spark? Give 3 reasons.

Answer:

- For many simple use cases, MapReduce/Hive may be more appropriate.
- Spark wasn’t designed as a multi-user environment (coordination issues).
- Requires sufficient memory; if dataset can’t fit and services compete, performance degrades.
  Example: Many users sharing a small-memory cluster may cause frequent spills/recompute and instability.

---

40) Exam-style “trap” question: Why is Spark fault tolerant even though it caches in memory?

Answer:

- Spark tracks **lineage** (DAG of transformations).
- If cached partition is lost (executor failure), Spark **recomputes** it from lineage.
- Cluster manager can reassign tasks; the job continues unless driver dies.
  Example: Cached RDD partition disappears → Spark reruns map/filter on original HDFS block to rebuild it.