Spark (Lecture 12) — deep, exam-style

Q1) Why was Spark introduced when MapReduce already existed?

Answer

- MapReduce becomes **expensive for iterative workloads** because each iteration tends to read/write to disk repeatedly.
- MapReduce **lacks efficient data sharing** between steps (disk is the "sharing medium").
- Spark improves this by enabling **in-memory data sharing**, reducing repeated disk I/O.
  Example: ML training (many iterations over same dataset) is much faster when cached in memory.

Q2) Explain "Spark uses memory instead of disk" in fault tolerance and iterative processing.

Answer

- Spark keeps intermediate data **in RAM** (optionally spills to disk), unlike MR which writes after map/reduce.
- In iteration-heavy jobs, Spark can **reuse cached partitions** instead of re-reading HDFS each time.
- Fault tolerance is handled by **recomputing lost partitions using lineage** rather than always persisting everything to disk.
  Example: PageRank runs repeated joins; caching avoids reloading the same graph each iteration.

Q3) Define a Spark "cluster" and name its main components.

Answer

- A Spark cluster is a **group of nodes** that execute Spark applications **in parallel**.
- **Driver**: coordinates the job (brain).
- **Cluster Manager**: allocates resources; **Executors** run computation; **Tasks** are smallest units.
  Example: Running a log analysis job on 10 worker nodes instead of one laptop.

Q4) What exactly does the Driver do in Spark?

Answer

- Runs the main Spark context/session and defines transformations/actions.
- Converts code into a **DAG execution plan**, then schedules tasks.
- Requests resources from the cluster manager and coordinates execution.
  Example: Driver decides how many stages exist and sends tasks to executors.

Q5) What is a Spark "Executor" and why is it important?

Answer

- Executor is a **process** (typically JVM) running on worker nodes that executes tasks.
- It **reads data**, performs transformations, and can store intermediate results in memory.
- Executors communicate results back to the driver.
  Example: If you have 100 partitions, multiple executors process them in parallel.

Q6) Explain the flow: Input → Driver → DAG → Stages/Tasks → Output.

Answer

- Driver creates a **logical plan (DAG)** then optimizes into a physical plan and tasks.
- Cluster manager allocates executors; executors run tasks on **partitions**; shuffle/recompute happens if needed.
- Output can be returned to driver or written to storage.
  Example: Reading HDFS logs → filter errors → aggregate counts → write results.

Q7) Define DAG in Spark and relate it to lineage.

Answer

- DAG = **Directed Acyclic Graph** that tracks dependencies between operations.
- In Spark, DAG nodes represent **RDDs**, and arrows are **transformations**.
- DAG represents **lineage**, enabling recomputation if a partition is lost.
  Example: If executor dies, Spark rebuilds only the lost partition using DAG history.

Q8) What triggers Spark to actually execute computations?

Answer

- **Actions** trigger execution; transformations alone are lazy.
- When an action is called, Spark submits DAG to the scheduler.
- The DAG is split into stages (shuffle boundaries) then tasks.
  Example: `.collect()` executes everything before it.

Q9) Define “partition” in Spark and why it matters.

Answer

- A partition is the **smallest logical chunk** of an RDD/DataFrame.
- Each partition is processed by **one task** on an executor.
- More partitions increase parallelism but also scheduling overhead.
  Example: 10 partitions → 10 tasks running across executors.

Q10) What is “shuffle” and why is it expensive?

Answer

- Shuffle redistributes data so that the **same keys end up together** in partitions.
- It causes **data movement across executors/nodes**, involving network + disk spill risk.
- Shuffle boundaries split a DAG into stages; it’s a major performance cost.
  Example: `join()` needs to move records so matching keys meet on same partition.

Q11) Differentiate narrow vs wide transformations (with impact).

Answer

- **Narrow** (map/filter/flatMap): each output partition depends on **one input partition**, usually no shuffle.
- **Wide** (groupByKey/reduceByKey/join): output needs data from **many partitions**, requires shuffle.
- Wide transformations cost more due to network transfer + stage boundary creation.
  Example: `filter()` is narrow; `groupByKey()` is wide.

Q12) Why is `groupByKey()` usually worse than `reduceByKey()`?

Answer

- `groupByKey()` shuffles **all values** for each key across the network.
- It has higher memory overhead because it gathers the full list before combining. (This is the classic implication of wide shuffles.)
- `reduceByKey()` performs partial/local aggregation before shuffling smaller results (conceptually reduces shuffle load).
  Example: Summing purchases per user: use `reduceByKey(sum)` instead of grouping all purchases.

Q13) Define RDD and list its 4 core properties.

Answer

- RDD is Spark's core abstraction: **fault-tolerant distributed collection** for parallel ops.
- **Immutable**: cannot be changed, only transformed.
- **Lazy evaluated**: transformations run only when action is called.
- **Resilient**: recovers via lineage on failure.
  Example: Log RDD filtered repeatedly for error patterns.

Q14) What is lineage and how is it used for fault tolerance?

Answer

- Lineage is the **history of transformations** used to create an RDD.
- Spark doesn't always store intermediates; if data is lost, it **recomputes** lost partitions.
- This avoids always replicating intermediate data like MR would.
  Example: Executor crash loses partition 7 → Spark re-runs the transformations for that partition only.

Q15) Explain Job → Stage → Task hierarchy.

Answer

- Spark breaks an application into **stages and tasks** based on transformations.
- Stages are split at **shuffle boundaries**.
- Each stage has tasks, usually **one per partition**.
  Example: `textFile → map → filter → reduceByKey` becomes multiple stages if shuffle exists.

Q16) What are transformations vs actions? Give 3 examples each.

Answer

- Transformations create new RDDs and are **lazy**. Examples: `map`, `filter`, `reduceByKey`.
- Actions return results/write output and execute **immediately**. Examples: `count`, `collect`, `saveAsTextFile`.
- Actions trigger the whole pipeline execution.
  Example: `rdd.map(...).filter(...).count()` runs only when `count()` is called.

Q17) What does "cache/persist" do and when should you use it?

Answer

- Stores RDD partitions in memory across executors after first compute.
- Speeds up iterative algorithms by avoiding recomputation and repeated reads.
- Choose storage levels (memory only vs memory+disk vs disk).
  Example: Logistic regression training loop uses same dataset many times → cache it.

Q18) "Spark is not designed as a multi-user environment" — why is that a problem?

Answer

- Spark relies heavily on memory; users must ensure the dataset fits enough memory or performance degrades.
- Multiple users cause contention; coordinating memory becomes complicated (common cluster issue).
- For simple use cases, MR/Hive may be more appropriate than Spark's heavy in-memory approach.
  Example: Many analysts running Spark jobs simultaneously can crash executors via OOM.

Q19) Compare Spark vs MapReduce (3 strong differences).

Answer

- Spark: **in-memory** reuse; MR: writes intermediate to **disk**.
- Spark: DAG with lazy evaluation; MR: linear map→shuffle→reduce pattern.
- Spark: best for iterative/interactive/streaming; MR: fine for one-pass batch jobs.
  Example: Iterative ML → Spark; nightly ETL batch (simple) → MR/Hive can be okay.

Q20) Scenario: executor fails mid-job. What happens?

Answer

- Cluster manager can **reassign tasks** if an executor/worker fails.
- Lost partitions are **recomputed using lineage**.
- If driver fails, job may restart unless checkpointed (driver is critical).
  Example: One node dies during shuffle → Spark recomputes missing partitions and continues.

MongoDB (Distributed) — from exam guide topics

Q21) Explain Primary vs Secondary in a MongoDB replica set (read/write).

Answer

- **Primary** handles **all writes**; secondaries replicate from primary's log.
- **Secondaries** can serve reads (depending on read preference), but must stay consistent via replication.
- Failover: if primary dies, an election chooses a new primary (availability).
  Example: E-commerce orders write to primary; analytics dashboard reads from secondary.

Q22) What is the oplog and why is it essential?

Answer

- It is the operation log that records changes so secondaries can replicate updates.
- It enables **asynchronous replication**: secondaries apply operations in order.
- It supports recovery: a lagging secondary catches up by replaying missing operations.
  Example: Primary updates inventory; secondary replays oplog entries to match.

Q23) Define Read Concern and why it matters in distributed reads.

Answer

- Read concern controls **how "fresh/committed"** the data must be before read returns.
- It's about consistency vs latency: stricter concerns reduce stale reads but can increase wait.
- In replica sets, it affects whether you might see data not yet majority-committed.
  Example: Bank balance read should prefer stronger consistency than "latest but risky."

Q24) Define Write Concern and explain "majority" in simple words.

Answer

- Write concern specifies **how many nodes must acknowledge** a write before it's considered successful.
- "majority" means **more than half** of replica set members confirmed the write → safer.
- Stronger write concern reduces risk of rollback during failover but increases latency.
  Example: Placing an order should use majority; logging page views might not.

Q25) What is an Arbiter and when is it used?

Answer

- Arbiter is an extra node that **votes in elections** but does not store full data.
- Used to achieve an **odd number of votes** to avoid election ties.
- Tradeoff: helps election reliability but doesn't improve read capacity or redundancy.
  Example: 2 data nodes + 1 arbiter → still can elect a primary if one data node fails.

Q26) Explain sharding in MongoDB and why "shard key" selection is critical.

Answer

- Sharding splits data across multiple machines to scale horizontally.
- Shard key decides how documents distribute; poor keys cause **hotspots** (uneven load).

- Good shard key supports balanced load and efficient queries (targeted vs scatter-gather).
  Example: Sharding by `userId` balances; sharding by `country` might hotspot one shard.

Q27) Scenario: You shard by "date" and all new inserts go to one shard. What went wrong and how to fix?

Answer

- Monotonic shard key (increasing date) causes "hot shard" because new data lands in the newest chunk.
- Fix with better key design: add randomness or use compound key (e.g., `date + userId`).
- Use balancing/chunk splitting properly to distribute writes.
  Example: Logs shard by `(day, deviceId)` instead of just timestamp.

ZooKeeper — znodes, watches, leader election, atomic broadcast

Q28) What is ZooKeeper and why do distributed systems use it?

Answer

- It provides coordination services like **leader election, configuration, synchronization**.
- Uses a hierarchical namespace (tree) of znodes to store small coordination metadata.
- Helps systems stay consistent (who is leader, who is alive, shared state).
  Example: Kafka historically used ZooKeeper to coordinate brokers and metadata.

Q29) Define a znode and list 3 things its "content" can contain.

Answer

- znode is a node in ZooKeeper's tree; it stores small data + metadata.
- Content can include: configuration values, leader identity, service discovery addresses.
- Also stores metadata like versioning and timestamps (used for consistency control).
  Example: `/services/payment/instance-3 = "10.0.0.8:9000"`.

Q30) Explain watches in ZooKeeper and how they are “sent” (conceptually).

Answer

- A watch is a one-time notification mechanism when a znode changes.
- Client sets watch during read; when change occurs, server notifies client, then watch must be re-registered.
- Used to avoid constant polling and enable reactive coordination.
  Example: Workers watch `/leader` so they immediately know when leader changes.

Q31) How does ZooKeeper do leader election using “smallest sequence number” idea?

Answer

- Clients create **ephemeral sequential** znodes under an election path (e.g., `/election/n_0001`).
- The client with the **smallest sequence number** becomes leader; others watch the node just before them.
- If leader dies, its ephemeral node disappears → next smallest becomes leader automatically.
  Example: 5 brokers compete; broker with `n_0003` becomes leader after `n_0001` and `n_0002` disappear.

Q32) Explain leader vs follower in a ZooKeeper cluster.

Answer

- Leader orders writes and ensures cluster consistency; followers replicate and serve reads (common pattern).
- Followers acknowledge proposals; leader commits when enough confirmations happen (reliability).
- This supports fault tolerance: if leader fails, new leader is elected.
  Example: Config updates go through leader; clients can read config from followers.

Q33) What is “atomic broadcast” in ZooKeeper (high-level)?

Answer

- Means all servers deliver the **same sequence of updates** in the same order.
- Guarantees consistency: every node agrees on state transitions.
- Needed so watchers and reads don’t observe conflicting histories.
  Example: Everyone sees config change “v5” after “v4”, never swapped.

HBase — data model + read/write flow + architecture

Q34) Define HBase data model using “multidimensional map”.

Answer

- HBase stores data as: **RowKey → ColumnFamily:Qualifier → (Timestamp → Value)** (multi-dimensional).
- Column Families are physical storage groups; qualifiers are dynamic columns inside them.
- Timestamps allow multiple versions of the same cell.
  Example: RowKey=`user123`, CF=`profile`, qualifier=`email`, versions for old/new emails.

Q35) What is a RowKey and what should it contain (3 best practices)?

Answer

- RowKey uniquely identifies a row and controls physical ordering and access pattern.
- Should support your query pattern (prefix scans), avoid hotspots (don’t use purely increasing IDs).
- Can be composite: include userId + region + reversed timestamp depending on access needs.
  Example: For time-series reads newest-first, use reversed timestamp prefix.

Q36) Why are Column Families important (storage + performance)?

Answer

- Column families define **physical storage layout**; data in same CF is stored together on disk.
- Accessing only one CF avoids scanning irrelevant data (I/O efficiency).
- CFs affect compaction and read amplification; fewer well-chosen CFs is better.
  Example: CF `meta` for small frequently-read fields; CF `events` for large sparse event logs.

Q37) Explain HBase write path: MemStore, WAL, SSTable/HFile (conceptual).

Answer

- Write hits in-memory structure (MemStore) and is also logged for durability (WAL idea).
- When MemStore fills, it flushes to disk into sorted files (often called SSTable/HFile concept).
- Compactions merge files over time for read efficiency.
  Example: High-volume sensor writes buffered in memory and periodically flushed.

Q38) Explain HBase read path (why reads are not “just one place”).

Answer

- Read may check cache + MemStore (recent writes) and multiple disk files due to flushes/versions.
- Needs to merge results by timestamp/version to return correct value.
- Proper schema (RowKey + CF design) reduces the amount of data touched.
  Example: Reading user profile should touch only `profile` CF not full `events`.

Q39) Client-server architecture of HBase: what flows where (conceptual)?

Answer

- Client sends get/put/scan to the correct region server (based on RowKey range).
- Region servers manage reads/writes for their regions and interact with storage files.
- Metadata/coordination (conceptually) is needed so clients know which region holds the key.
  Example: A big table is split into regions; each region handled by different server.

Q40) When should you use HBase instead of a relational DB? (3 clear reasons)

Answer

- Need **very large scale** with sparse wide tables and fast key-based access.
- Need **time-versioned** data (timestamps) and high write throughput.
- Need integration in Hadoop ecosystem (storage + big-data pipelines).
  Example: Storing clickstream events keyed by `(userId, time)`.

Hive — partitions, bucketing, external table (basic but examinable)

Q41) Define Hive partitioning and why it helps.

Answer

- Partitioning splits table data into directories based on a partition column (e.g., date).
- It reduces scan by pruning partitions (reads only relevant folders).
- Improves query performance and manageability of large datasets.
  Example: Query last week's sales reads only `dt=...` partitions.

Q42) Compare partitioning vs bucketing (3 differences).

Answer

- Partitioning is directory-level separation; bucketing splits data into fixed number of files by hash.
- Partitioning helps with partition pruning; bucketing helps joins/sampling by predictable file structure.
- Too many partitions = small files problem; bucketing controls file counts more predictably.
  Example: Partition by `date`, bucket by `userId` for faster user-based joins.

Q43) What is an External Table in Hive and when should you use it?

Answer

- External table points to data stored in HDFS; dropping table doesn't delete underlying data (safety).
- Used when data is shared between tools or managed outside Hive.
- Helps keep raw data durable while changing schemas/queries.
  Example: Spark writes Parquet to HDFS; Hive external table reads it for SQL reporting.

Kafka — producer/consumer/offset (light but must be correct)

Q44) Explain "offset" in Kafka and why it matters.

Answer

- Offset is the position of a consumer in a partition's log (what has been read).
- Enables replay: consumers can re-read from earlier offsets for recovery/debugging.
- Supports consumer groups: each consumer tracks offsets for load sharing.
  Example: If dashboard fails, restart consumer from last committed offset.

Heavy design question (20 marks): Lambda Architecture

Q45) Design a Lambda Architecture for a real-time e-commerce streaming app (3 layers + tech + flow).

Answer

- **Batch layer (accuracy + full history):** Store raw events in **HDFS**; process periodic recomputations using **Spark** batch jobs for "ground truth" aggregates.
- **Speed layer (low latency):** Ingest events via **Kafka**; run near-real-time processing using **Spark (streaming concept)** to produce fast incremental views.
- **Serving layer (queryable results):** Store serving views in **HBase** (fast random reads by key) or **MongoDB** for flexible query patterns; expose dashboards.
  Example flow: User clicks "Add to cart" → Kafka topic → speed layer updates "top products now" → dashboard shows instantly; nightly batch recompute fixes any late events.