Q1) What is HBase and where does it sit in the Hadoop ecosystem?

Answer:

- HBase is a **distributed, column-oriented data store**.
- It is built **on top of HDFS**, and HBase files are stored internally in HDFS.
- It provides storage for **Hadoop distributed computing** (fast access pattern compared to raw HDFS).
  Example: Store user profile + activity logs in HBase while raw logs stay in HDFS.

Q2) "HBase looks like a table but is NOT a relational table." Explain.

Answer:

- It looks tabular but it's **not an RDBMS** (no relational constraints/joins).
- It's a **sparse, distributed, multidimensional, sorted map**.
- Its data model is key-based: **(RowKey, CF, Qualifier, Timestamp) → Value**.
  Example: Some rows can have `personal:email`, others don't — no NULL storage overhead.

Q3) Write and explain the exact HBase "multidimensional map" formula.

Answer:

- Exact formula: **(RowKey, ColumnFamily, ColumnQualifier, Timestamp) → Value**.
- Meaning: a "cell" is uniquely identified only when you include the timestamp.
- This allows **multiple versions** of the same column over time.
  Example: `info:name@ts1="Rasheed"` then later `info:name@ts2="Rasheed Ahmed"`.

Row Key (big exam trap)

Q4) What is a RowKey and why does its design affect performance?

Answer:

- Every row has a **single unique RowKey**.

- Rows are sorted **lexicographically** by RowKey, enabling very fast range scans.
- Best performance comes from **designing RowKeys wisely** (because storage + region splits follow that order).
  Example: If queries are by `userId`, make RowKey start with userId so lookups are O(1)-like.

Q5) Lexicographic vs numeric sorting: why is `"1","2","10"` dangerous?

Answer:

- Strings sort lexicographically: `"1","10","2","20"` (wrong numeric order).
- HBase compares RowKeys as **byte arrays**, so your encoding matters.
- Use fixed-width binary encoding (e.g., `Bytes.toBytes(int)`) to preserve numeric order.
  Example: Store integer IDs in binary, not as "strings", if you need correct numeric ranges.

Q6) Why does HBase use byte arrays (byte[]) for keys/qualifiers?

Answer:

- **Flexibility:** client can encode any datatype.
- **Performance:** raw byte comparison is fast.
- **Efficient sorting/splitting:** lexicographic ordering helps regions split cleanly.
  Example: UUID, timestamp, hash — all become byte[] and still sort consistently.

Q7) List at least 5 possible RowKey data types from the slide.

Answer:

- Any type can be RowKey: **String, Integer, Long, Timestamp, UUID**.
- Also: **Composite keys, Hashes, Encoded binary keys**.
- All ultimately become **byte[]** internally.
  Example: `(regionId#userId#reverseTime)` as a composite RowKey for time-series per region.

Column Families (most important)

Q8) What is a Column Family and why is it “the unit of storage”?

Answer:

- HBase groups columns into **Column Families (CFs)**.
- CFs are fixed at schema definition; you can’t add/remove them frequently.
- Data in same CF is stored together in the same physical files (**HFiles**) → huge performance impact.
  Example: Store `personal` CF separately from `activity` CF so scanning profile doesn’t touch activity history.

Q9) Explain how CFs are stored for a single row `user_001`.

Answer:

- For one row, CFs are stored separately: `personal` in one HFile, `orders` in another, `activity` in another.
- You can scan **one CF without touching others**.
- Different CFs can be totally different sizes (good for isolation).
  Example: Analytics can scan only `activity` CF and ignore `personal`.

Q10) Why are too many column families a bad idea?

Answer:

- Best practice: **max 2–3 CFs**, up to 5 is risky, 1–2 ideal.
- Each CF has separate MemStore/HFiles/WAL edits/flushes/compactions → overhead explodes.
- Many CFs waste RAM and can cause **compaction storms**.
  Example: 10 CFs means 10 separate write buffers per region → region server struggles.

Q11) Give 3 CF settings and what they do.

Answer:

- **TTL:** auto-delete old data after X seconds.

- **Max versions:** how many historical versions to keep.
- **Bloom filters:** reduce disk seeks (faster existence checks).
  Example: `activity` CF uses TTL=7 days so old app events vanish automatically.

Q12) Explain CF isolation with “personal / orders / activity” pattern.

Answer:

- `personal`: small, rarely updated → low compaction pressure.
- `orders`: large, frequently updated → separate compaction needed.
- `activity`: high-write time series → can use short TTL.
  Example: Keep `personal` forever, expire `activity` quickly, compact `orders` heavily.

Column Qualifiers (schema-less within a family)

Q13) What is a Column Qualifier and why does it give “schema-less” flexibility?

Answer:

- Inside a CF, you can create unlimited dynamic columns called **qualifiers** (column names).
- Qualifiers are **not predefined** and can vary row to row.
- No schema change needed to add new qualifier during inserts.
  Example: One user has `personal:linkedin`, another doesn’t — still valid.

Q14) Why are qualifiers “lightweight” compared to column families?

Answer:

- Qualifiers require **no server memory allocation**.
- They do not create new MemStores or HFiles, nor affect WAL structure.
- You can have **millions of qualifiers per row** (common in time-series).
  Example: `orders:2024-01-01`, `orders:2024-01-02`, … thousands of daily qualifiers.

Q15) Are qualifiers globally unique? Explain namespace rule.

Answer:

- Qualifiers are **namespaced by CF**; same qualifier name in another CF is unrelated.
- Their meaning is only inside their family.
- This encourages clean grouping: keep "name" only in `info` or `personal`.
  Example: `info:name` ≠ `orders:name` (they're different columns).

Versions + Timestamps (major exam area)

Q16) What is a "version" in HBase?

Answer:

- Version is identified by a **timestamp**.
- A single cell can store many values over time (not new rows).
- Latest version is returned by default.
  Example: Old email value remains as older version, new email is the latest version.

Q17) "Versions are not new rows." Explain this clearly.

Answer:

- Versions are stored **inside the same row** under same RowKey.
- They differ only by timestamp under same CF + qualifier.
- It enables audit/recovery/time-based queries.
  Example: User name changed — you can still retrieve the previous name version.

Q18) Default version behavior vs custom version count.

Answer:

- Default: HBase keeps **1 version** (latest only).
- You can configure CF to store multiple versions (e.g., 3).
- This is controlled at CF definition (not per random column).
  Example: Keep last 3 profile updates for rollback.

Q19) Server timestamp vs user-supplied timestamp: why both exist?

Answer:

- If you don't provide timestamp, HBase uses **current system time** in ms.
- You can provide your own timestamp for ordering/event-time correctness.
- Useful for CDC, event logs, IoT measurements where event-time ≠ insert-time.
  Example: Late IoT event arrives—store with its original event timestamp.

Q20) Define Unix Epoch and why it appears in HBase versioning discussions.

Answer:

- Unix epoch: **00:00:00 UTC, 1 Jan 1970**.
- Systems count time as seconds since epoch; large numbers are just elapsed seconds.
- HBase version timestamps are essentially time values used to order versions.
  Example: A timestamp like `1732435394` means seconds after 1970, used for version ordering.

Q21) Why are values stored "most recent version first" inside HFiles?

Answer:

- HFiles store values sorted so most recent is first → fastest read for "latest value".
- Supports analytics like "value at time X" and audit logs.
- Makes versioning like a built-in append-only log (no extra logging table).
  Example: "What was GPA last semester?" can be answered by reading an older timestamp version.

LSM Tree + Internal Storage (write-optimized)

Q22) What is an LSM Tree and why is it used in HBase?

Answer:

- LSM = Log-Structured Merge Tree, optimized for **high write throughput**.
- Writes first go to **MemStore (RAM)** and also to **WAL** for durability.

- Flushed to disk as immutable sorted files; later compacted for read efficiency.
  Example: High-volume sensor updates are appended quickly instead of random disk overwrites.

Q23) Explain the MemStore → HFile flush process.

Answer:

- Data is written to MemStore in RAM; when full, it is **flushed to disk** as a new HFile.
- MemStore is per CF per region (each CF has its own MemStore).
- Frequent flushes can create many HFiles → later compaction overhead.
  Example: A burst of writes creates many small HFiles → reads become slower until compaction runs.

Q24) What is an HFile and how is it sorted internally?

Answer:

- HFile is the actual physical file on disk (on HDFS).
- Sorted by RowKey → then CF → then qualifier → then timestamp.
- Sorting supports fast scans and version retrieval.
  Example: Range scan from RowKey “A” to “H” is efficient because data is sorted by key.

WAL (Write-Ahead Log) — scenario favorite

Q25) What is WAL and why does every write go there first?

Answer:

- WAL = Write-Ahead Log; every Put/Delete is appended before data hits MemStore/disk.
- Stored as HLog files in HDFS; ensures **crash recovery**.
- Prevents data loss if RegionServer dies before flush.
  Example: Power failure after 1000 writes: WAL replay reconstructs them.

Q26) Scenario: 1000 records are in MemStore, server crashes before flush. Explain recovery.

Answer:

- Without WAL → records are lost (they were never flushed).
- With WAL: master assigns region to another RegionServer; new RS reads WAL and replays edits.
- WAL replay rebuilds MemStore → flush to HFiles → data restored.
  Example: After crash, the system replays "Put row1 name=Ali …" until all writes are restored.

Regions + Physical Model (very examinable)

Q27) Explain: Table → Regions → RegionServers → HFiles.

Answer:

- Logical table is split into **regions** by RowKey ranges.
- Regions are hosted on **RegionServers**; each region has MemStore + stores/HFiles + WAL.
- This is how HBase scales horizontally.
  Example: Region1: A–H, Region2: H–Q; users in those key ranges go to different servers.

Q28) What is a Region and how is it different from an HDFS block?

Answer:

- Region is a horizontal partition of a table by **RowKey range**.
- Regions are "counterpart" concept to HDFS blocks, but for table rows.
- Regions move/assign across RegionServers for balancing/failure handling.
  Example: If one server overloads, regions can be reassigned to others.

Q29) What does a RegionServer do (3 responsibilities)?

Answer:

- Hosts multiple regions and serves **reads/writes** for them.
- Uses a log (WAL) for durability and recovery.

- Manages stores per CF (MemStore + HFiles) per region.
  Example: Client Put to `row1` is served by the RegionServer hosting the region containing `row1`.

Q30) What does the HBase Master do?

Answer:

- Coordinates region servers (slaves) and assigns regions.
- Detects failures and reassigns regions on crash.
- Handles admin functions (schema operations).
  Example: If a RegionServer dies, master moves its regions to healthy servers.

Compaction + Compression + Cache

Q31) What is compaction and why is it needed?

Answer:

- Frequent small writes create many HFiles; reads must check all → slow.
- Deleted/old versions still occupy space; compaction merges and cleans.
- Compaction merges HFiles into fewer larger ones to speed reads and reclaim space.
  Example: After heavy updates, compaction removes expired versions and tombstones.

Q32) Minor vs Major compaction: compare with 3 differences.

Answer:

- Minor: merges a few HFiles, keeps old versions, triggered automatically when HFiles exceed threshold (~3).
- Major: heavier; removes tombstones + expired versions (TTL/VERSIONS), often daily or manual.
- Major produces one large HFile per CF per region; can impact performance.
  Example: Minor compaction reduces file count; major compaction actually cleans deleted rows.

Q33) What is compression in HBase and what it does NOT do?

Answer:

- Compression shrinks HFile size to reduce disk usage and I/O.
- Applied **per column family** when MemStore flush creates HFiles.
- It does **not** merge files or remove old versions (that's compaction).
  Example: Use Snappy to speed I/O; still need compaction to clear old versions.

Q34) Compare LZ4/Snappy vs GZIP in one answer (ratio vs CPU vs speed).

Answer:

- LZ4/Snappy: moderate compression, **very fast** read/write, low CPU.
- GZIP: high compression, slower reads/writes, high CPU usage.
- Choose based on workload: write-heavy low latency vs archival storage.
  Example: Time-series ingestion uses LZ4; cold historical archive can use GZIP.

Q35) What is Block Cache and why LRU matters?

Answer:

- Block cache stores hot HFile blocks (e.g., 64KB blocks) for fast reads.
- When cache is full, LRU evicts least-recently-used blocks.
- Helps avoid disk reads for frequently accessed data.
  Example: Popular user profiles stay in block cache, making reads millisecond-level.

Q36) What do Bloom Filters do in HBase reads?

Answer:

- Bloom filters help quickly decide if a row/column exists in an HFile without reading it fully.
- Reduces unnecessary disk seeks and speeds lookups.
- Especially useful when many HFiles exist (before compaction).
  Example: If a row definitely isn't in an HFile, Bloom filter prevents wasted disk access.

HBase vs HDFS + Write/Read Strategy

Q37) HBase vs HDFS: 3 clear differences and when to use each.

Answer:

- HDFS is great for **batch scans over big files**; not good for record lookup or updates.
- HBase is excellent for **fast random reads by row key** and high throughput.
- HBase is not for "query anything anywhere" (non-key ad-hoc queries are slow).
  Example: Store raw logs in HDFS; serve "get user profile by ID" from HBase.

Q38) Why are small writes problematic in HBase? What's the best practice?

Answer:

- Each small write has RPC overhead and triggers frequent MemStore flushes.
- Many tiny HFiles are created → heavy compaction overhead.
- Best practice: buffer small updates and write in bulk batches.
  Example: Collect 10k events then bulk write instead of writing each event one by one.

Q39) Explain 3 HBase write strategies: Random Writes vs Bulk Load vs Incremental Load.

Answer:

- Random writes: good because LSM-tree writes to MemStore then flushes.
- Bulk load: prepare/sort data into HFiles offline and load directly into regions (fast ingestion).
- Incremental load: stream smaller updates in real-time; each incremental write becomes a new version.
  Example: Nightly ETL uses bulk load; live clickstream uses incremental load.

Q40) Compare Get vs Scan and give a situation where each is correct.

Answer:

- **Get:** retrieves a single row by exact RowKey; returns highest version by default.
- **Scan:** range queries across multiple rows using RowKey ranges.
- HBase is optimized for key-based access; scans on random non-key patterns become expensive.

Example: Get `user_001` profile (Get). Fetch `user_001` to `user_050` activities (Scan with range).