0) Big Data Architecture “must include” ideas (the meta that connects everything)

Your guideline says you can be asked about **policies, replication, fault tolerance, leader selection** etc.
So whenever you explain any system, keep these **4 lenses** ready:

1. **Replication** (copies of data): why? fault tolerance + read scaling
2. **Fault tolerance**: what happens if a node dies? does system continue?
3. **Consensus / leader selection**: who is “in charge” for ordering writes?
4. **Consistency vs availability tradeoff**: can reads be slightly stale or must be latest?

You will literally reuse these 4 points in: **Mongo replica set**, **ZooKeeper**, **Kafka (broker leadership + offsets)**, **HBase (region/replication)**.

---

1) Kafka (basic but must be crystal clear)

Your guide: “producer consumer, offset… Kafka kaam kis tarah karta hai, datapoint hai kya karun.”

What Kafka is (in 1 line)

Kafka is a **distributed commit log**: a system that stores events in **ordered sequences** and lets many systems read them.

The story (how one datapoint moves)

Imagine an e-commerce “OrderPlaced” event:

1. **Producer** (website/backend) sends message `{orderId, userId, price, timestamp}`.
2. Kafka stores it in a **topic** (e.g., `orders`).
3. Topic is split into **partitions** (parallel lanes). Each partition is an ordered log.
4. Every message gets an **offset** (0,1,2,3…) inside *that partition*.
5. **Consumers** read messages by offsets.
6. Consumers save their progress (offset) so if they crash, they continue from the last offset.

Key terms you must be able to define

- **Topic**: category/stream name (`orders`, `clicks`, `payments`)
- **Partition**: parallel sub-stream; order is guaranteed **within** a partition
- **Offset**: position of a message in a partition
- **Consumer group**: a team of consumers sharing the work; each partition assigned to one consumer in the group (for parallelism)

Exam-style scenario lines (write these)

- If you want **order preserved** for each user → choose partition key = `userId`
- If you want **scalable processing** → increase partitions
- If consumer crashes → it restarts and continues from last committed **offset**

---

2) ZooKeeper (Lecture 14) — VERY IMPORTANT

Your guide screams: **Znodes, tree, what node contains, leader selection algorithm scenario, watches, cluster/leader-follower, atomic broadcast**.

2.1 What ZooKeeper is

ZooKeeper is a **coordination service** for distributed systems.
It stores small but critical shared info: **configuration, leader info, membership, locks**.

2.2 ZooKeeper data model: the tree + znode

ZooKeeper is like a tiny filesystem:

- `/` root
- `/config`
- `/leaders`
- `/members`

A **znode** is a node in this tree. It stores:

- **small data** (usually < 1MB),
- **metadata** (version, ACL, timestamps),
- and can have **children** like directories.

What can be inside node data?

- config string (`"db=prod;timeout=5s"`)
- leader address (`"10.0.0.7:9092"`)
- service registry info

- lock owner id

2.3 ZooKeeper cluster = “ensemble” + quorum

ZooKeeper runs as a small cluster called **ensemble** (usually 3 or 5).
It needs **majority (quorum)** to operate:

- 3 servers → need 2 alive
- 5 servers → need 3 alive

2.4 Roles: Leader, Followers, Observers

- **Leader**: handles writes, orders updates
- **Follower**: serves reads, forwards writes, votes in elections
- **Observer**: serves reads but **does NOT vote** (read scaling without slowing writes)

Why observers matter: adding more voters increases quorum size → **writes become slower** because leader waits for more acknowledgements. Observers avoid that.

2.5 Types of znodes (EXAM FAV)

- **Persistent**: stays until deleted manually/programmatically
- **Ephemeral**: exists only while client session is alive; disappears if the client dies; cannot have children
- **Sequential**: ZooKeeper appends a sequence number (e.g., `node-00005`)
- **Ephemeral-sequential**: used for leader election fairness (below)

2.6 Watches (NOTIFICATIONS) — super important

A **watch** is a **one-time trigger** a client sets on a znode. When that znode changes, ZooKeeper **pushes a notification**.
Why? Avoid constant polling.

What can trigger a watch?

- node data change
- node deletion
- node creation
- children list change

Exam scenario you can write:

“Instead of asking repeatedly ‘has leader changed?’, client sets a watch on `/leader` and ZooKeeper notifies instantly.”

2.7 Leader election algorithm (you MUST write steps)

Mechanism in slides: **ephemeral-sequential nodes** under an election path.

Algorithm (write like this in exam):

1. Each server creates an **ephemeral-sequential** znode:
   `/election/server-00005`
2. ZooKeeper assigns increasing numbers.
3. The server with the **smallest number becomes leader**.
4. If leader dies → its node is **ephemeral**, so it disappears automatically.
5. Next smallest becomes leader — no fighting, clean handover.

2.8 Zab = ZooKeeper Atomic Broadcast (replication/consensus idea)

Your slides literally say Zab is the protocol that ensures **ordered replication**.

Story of a write in ZooKeeper:

1. Client sends write → goes to **leader**.
2. Leader broadcasts to followers.
3. **Majority confirms**.
4. Leader commits.
5. All nodes apply the same change in the same order.

Guarantees to memorize (they'll ask):

- total order of updates
- atomicity of updates
- single system image
- reliability once quorum acknowledges

---

3) Hive (Lecture 10) — only the basics your guide wants

Guide: focus on **partitioning, bucketing, external table, HDFS link**; skip repetitive DAG/data types/HQL examples.

What Hive is

Hive is the **SQL layer on Hadoop**. It lets you write SQL-like queries on big data stored in **HDFS**.

The crucial concept: "table is metadata, data stays in HDFS"

In Hive:

- The **data** is in files in **HDFS**
- The **table** is mostly **metadata** (schema, file format, location)

That's why your guide says: "hdfs files tables mai kese link hoti hai."

External vs Managed (Internal) table

Managed/Internal table

- Hive "owns" the data
- If you `DROP TABLE`, data files are also deleted

External table

- Hive only points to data location
- If you `DROP TABLE`, HDFS files remain (safe for shared datasets)
- This is super common in big data pipelines

Partitioning (MOST IMPORTANT)

Partitioning = physically organizing data in folders by a column.

Example: partition by date
HDFS layout:

- `/warehouse/sales/dt=2025-12-01/`
- `/warehouse/sales/dt=2025-12-02/`

Why it's powerful:

- Query with `WHERE dt='2025-12-02'` only scans that folder → huge speedup

Bucketing (also important)

Bucketing = splitting data into a fixed number of files based on hashing of a column.

Example:

- bucket by `userId` into 8 buckets
- rows with same hash go to same bucket

Why it helps:

- faster joins (bucketed join)

- sampling
- more even parallelism

Partition vs Bucket (one-liner difference)

- Partition = directory-level split (coarse, based on common filter columns)
- Bucket = file-level split inside table/partition (fine, hash-based)

---

4) Spark (Lecture 12 basics + Lecture 13 important) — focus on DAG, transformations/actions

Guide: Spark is important, but Lecture 12: “basic concepts, skip repetitive, no need to remember code.” Also: “DAG main cheez… parallel paths… transformations/actions.”

4.1 Why Spark exists (the story)

MapReduce writes to disk after each step → slow.
Spark keeps data **in memory** → much faster for iterative + multi-step workloads.

4.2 Spark architecture (must know words)

- **Driver**: the brain (your main program). Creates DAG, schedules work.
- **Executors**: workers that run tasks.
- **Cluster manager**: allocates resources (Standalone/YARN/Mesos/K8s).
- **Job** → split into **stages** → split into **tasks**.

4.3 Transformations vs Actions (EXAM LOVES)

Your guide: “chand hi tou hain, yaad karlen.”

Transformation = creates a new dataset (lazy, doesn’t run immediately)

- `map`, `filter`, `select`, `withColumn`, `join` (conceptually)
- Spark builds the plan but doesn’t execute yet

Action = triggers computation (runs the DAG)

- `count`, `collect`, `show`, `save`, `take`

Lazy evaluation (the trick)
Spark delays execution until an action. That allows optimization.

4.4 DAG (Directed Acyclic Graph) — the heart

DAG is the execution plan of transformations: nodes = operations, edges = data flow.
Spark uses the DAG to:

- pipeline operations
- decide stages
- optimize execution

4.5 Parallel paths, partitions, shuffle (what they mean)

- **Partition**: chunk of data processed in parallel (one task per partition)
- **Parallel paths**: if dataset has many partitions, many tasks run at once
- **Shuffle**: data movement across machines (expensive)
    - happens in operations like groupBy/join/sort
    - causes stage boundaries

4.6 Cache/Persist (why Spark becomes “in-memory”)

When you reuse the same dataset multiple times (like multiple actions), caching avoids recomputation.

Use-case:

- iterative ML
- repeated analytics on same cleaned dataset

---

5) HBase (Lecture 13) — IMP + scenario + read/write flow

Your guide is extremely specific: **data structure, row key, column families, multidimensional map, storage linking (sstable/mtable), versions/timestamps, client-server architecture, read/write flow**.

5.1 What HBase is

HBase is a **distributed NoSQL database** built on Hadoop storage.
It’s designed for:

- huge tables (billions of rows)
- fast random read/write by key
- sparse data (not every row has every column)

5.2 HBase data model (THIS IS THE EXAM CORE)

HBase is often described as a **multidimensional map**:

You can think of a cell address as:

```
(rowKey, columnFamily:qualifier, timestamp) -> value
```

Meaning **a row** is not like SQL rows; it's like a key-space.

RowKey (the "primary key")

Everything is accessed by RowKey. RowKey design decides performance.

What can be inside RowKey? (your guide asks this)

- userId
- deviceId
- region + userId
- reverse timestamp trick (if you want recent-first scanning)
- composite keys like `customerId#orderId`

Benefits of good RowKey

- even distribution across regions (load balancing)
- avoids hotspotting
- supports range scans (because keys are sorted)

Column Family (storage unit)

Column families are **physical storage groups**. Your guide says "Column Family storage." All qualifiers under a family are stored together.

Example:

- `profile:name`
- `profile:age`
- `activity:lastLogin`

Why families matter:

- you define families upfront (schema part)
- families impact disk layout + read efficiency

Qualifier (dynamic column name)

Inside a family, qualifiers can appear anytime (semi-schema-less).

5.3 Versions + timestamps

HBase can store multiple versions of a cell, separated by timestamps. Your guide mentions "versions, timestamp."
Use-case:

- audit history
- time-series
- "what was the value yesterday?"

5.4 Why HBase (distinguishing feature)

If the exam asks "why HBase / which scenario?" you answer:

Use HBase when you need:

- **low-latency random access** by key at huge scale
- **high write throughput**
- sparse, wide tables
- time-versioned values

Not ideal for:

- heavy ad-hoc joins
- complex SQL analytics (that's Spark/Hive)

5.5 Client-server architecture + flows (must be able to narrate)

Your guide: "Hbase client server architecture. Flow of data. Query read/write."

Write flow (story)

1. Client writes `(rowKey, cf:qualifier, value)`
2. RegionServer receives it
3. Write is appended to **WAL** (write-ahead log) for durability
4. Then written to **MemStore** (in-memory)
5. When MemStore fills, it flushes to disk as immutable files (HFiles / SSTable-like idea)
6. Background **compaction** merges files for efficiency

Read flow (story)

1. Client requests rowKey/scan
2. RegionServer checks:
    - MemStore (newest)

    - BlockCache (cached blocks)
    - HFiles on disk
3. Merges versions (based on timestamp rules)
4. Returns result

If you write this cleanly in exam with arrows, it looks perfect.

---

6) MongoDB (Lectures 15 & 16) — distributed, scenario-based, read/write concerns, oplog, elections, arbiter, sharding

Your guide: “Distributed MongoDB important… primary secondary… reading writing op log… read concern, write concern… arbiter… sharding, key, load balancing.”
Also: **skip MongoDB data types**.

6.1 Replica Set (primary + secondaries)

A typical deployment: **1 primary + 2 secondaries**, optional arbiter/hidden/delayed nodes.

How writes work (default)

- Client writes to **primary**
- Primary replicates to secondaries via **oplog** (operations log)
- Secondaries apply operations in order

6.2 Automatic failover (elections)

If primary dies:

- Secondaries detect via heartbeats
- election starts (Raft-like consensus mentioned)
- one becomes new primary
- clients reconnect
- downtime about 2–5 seconds

6.3 Write Concern (consistency guarantee for writes)

Write concern = how many nodes must acknowledge the write.

- `w:1` → only primary confirms (fast, weaker)
- `w:majority` → majority confirms (safer)

- `w:0` → fire-and-forget

Scenario line for exam:

- Banking/critical updates → `w:majority`
- Logging/metrics where speed matters → `w:1` or even weaker

6.4 Read Concern (what version of data reads can see)

Read concern controls consistency/isolation of reads.

- `local` → may see data not majority-committed
- `majority` → only majority-committed
- `linearizable` → strongest, slowest (read guaranteed latest committed)

6.5 Arbiter (what it is and why)

Arbiter is a node that **votes** but holds **no data**.
Purpose: keep odd number of votes to achieve election quorum (your lecture 16 also mentions arbiters + quorum idea).

6.6 Sharding (horizontal scaling)

Lecture 16 highlights sharding topics: mongos, shard keys, range sharding, cardinality, avoid hotspotting.

What is sharding?
Splitting one large collection across multiple machines (horizontal scaling).

Core components

- `mongos` query router
- config servers store metadata (chunk locations)
- shards (each shard is a replica set)

How data is split
MongoDB breaks data into **chunks** (64MB default).
Balancer moves chunks to keep distribution even.

6.7 Shard key (this decides performance)

A good shard key:

- high cardinality
- evenly distributed

- queried often
- not monotonically increasing (or use hashed)

Bad keys (examples):

- timestamp (monotonic → hotspot)
- boolean (only 2 values)
- low-cardinality category

Hotspotting definition (lecture 16)
Too many reads/writes hitting same shard/chunk, overloading one shard while others idle.

6.8 Reads/Writes in sharded cluster (exam scenario)

- Write: client → mongos → correct shard → primary of that shard
- Read:
    - if query includes shard key → goes to one shard (fast)
    - without shard key → scatter-gather to all shards (slow)

6.9 Transactions & consistency tradeoff

MongoDB can be tuned CP-like vs AP-like; better consistency = slower.
Multi-document ACID transactions across shards exist (v4.2) but slower; use only when needed.

---

7) The BIG 20-mark question: Lambda Architecture design (Databricks mention)

Your guide: “Final Lambda Architecture design… real-time streaming application… teen layer… serving layer, speed layer, dash layer… module and flow (20 marks).”

7.1 What Lambda Architecture is (simple definition)

Lambda architecture is a way to build systems that need:

- **real-time results** (seconds)
- **accurate historical results** (minutes/hours, recomputable)
- and a **single view** for queries/dashboard

So it combines:

1. **Batch layer** (accurate, full history)
2. **Speed layer** (real-time, fast approximate)

3. **Serving layer** (where queries read from)
   (+ in your guide you must also include dashboard/flow/specs)

7.2 The “exam diagram” template (write this in exam)

Pick any scenario: e-commerce, ride-hailing, IoT sensors, banking fraud, clickstream.

Then draw:

Data Sources → Ingestion → (Batch + Speed) → Serving → Dashboard

Now plug your course tech into it (your guide demands that).

7.3 A very strong Databricks-style Lambda (easy to write, looks professional)

Use this template in the exam:

(A) Data Sources

- Website clicks, orders, payments
- Mobile app events
- IoT sensors

(B) Ingestion

- **Kafka** as event pipeline (producers/consumers/offsets)
- **ZooKeeper** (if they want classic Kafka coordination concepts: leader election, watches, reliability ideas)

(C) Batch Layer (accurate master dataset)

Goal: store *all data*, compute truth from full history.

- Storage: **HDFS** (raw immutable logs)
- Query/warehouse: **Hive external tables** pointing to HDFS partitions
- Processing: Spark batch jobs (Databricks notebooks/jobs)

Output of batch:

- daily aggregates: revenue per day, user cohorts, product ranking
- stored in **serving store** (could be HBase or Mongo depending on use)

(D) Speed Layer (real-time processing)

Goal: compute quick, up-to-date views from latest stream.

- Spark **Structured Streaming** / Spark streaming conceptually (you can just say “Spark streaming”)
- Reads from Kafka
- Performs transformations (filter/map/windowed counts)
- Writes live results to low-latency store

(E) Serving Layer (fast query layer)

This is what dashboards and APIs hit.

Two good choices (both from your exam topics):

1. **HBase** for fast key-based lookups (e.g., `userId → latest stats`)
2. **MongoDB** for flexible dashboards/documents, and sharded scaling if needed

(F) Dashboard Layer (explicitly asked)

- BI dashboard / web dashboard / alerting system
- Shows:
    - real-time metrics (from speed outputs)
    - historical metrics (from batch outputs)

7.4 The key “Lambda trick” they love

You write this line:

Batch layer recomputes the accurate truth from full history, and periodically corrects/merges the speed layer’s fast but potentially incomplete results.

That’s the whole reason Lambda exists.

7.5 Give “many types” of architectures you can choose in exam

Here are **4 ready variants** you can pick depending on the scenario:

Variant 1: E-commerce real-time analytics (most common)

- Kafka → Spark Streaming (speed)
- HDFS/Hive + Spark batch (batch)
- Serving: MongoDB (dashboard docs) OR HBase (fast lookup)
- Dashboard: “Live sales, trending products, fraud alerts”

Variant 2: Fraud detection (strong narrative)

- Kafka events: card swipes
- Speed: Spark streaming runs rules/ML scoring → alert store

- Batch: nightly Spark recomputes models/features from full history
- Serving: HBase for `customerId → riskScore`, Mongo for investigation cases

Variant 3: IoT smart campus (sensor streams)

- Kafka ingestion from sensors
- Speed: streaming computes “current temperature anomalies”
- Batch: daily aggregates, seasonality baselines
- Serving: HBase time-series rowkeys (deviceId#reverseTime)

Variant 4: Ride-hailing / delivery ETA

- Speed: live driver pings compute ETAs
- Batch: historical travel times per route/hour
- Serving merges both: “ETA = baseline (batch) + adjustment (speed)”

In exam, you only need **one**, but having these in your head makes you unstoppable.