1) What is HBase? (Start of lecture)

The simple idea

HBase is a **distributed database** made for **very large data** where you want **fast access**.

Your slide says:

- HBase is a **distributed column-oriented data store** built on top of **HDFS**
- It's an Apache project to provide storage for Hadoop distributed computing
- Data is logically organized into **tables, rows, columns**

VERY IMPORTANT: Where does it live?

HBase is not independent storage. It stores its files inside HDFS. The diagram says:

- **HBase is built on top of HDFS**
- **HBase files are internally stored in HDFS**

Story:
Think of HDFS as the **warehouse** (file storage).
HBase is the **inventory system** that makes the warehouse searchable and updatable quickly.

---

2) HBase is NOT an RDBMS (THIS COMES IN EXAM)

The slide says:

HBase looks like a table, but it is **NOT a relational table**.

So:

- No SQL joins like MySQL (joins aren't supported)
- No "search any column anytime" style querying
- HBase is built for **key-based access**

---

3) The HBase Data Model (MOST IMPORTANT CONCEPT)

Your slide gives the most famous HBase definition:

HBase is a **sparse, distributed, multidimensional, sorted map**
(RowKey, ColumnFamily, ColumnQualifier, Timestamp) → Value

Let's unpack each word slowly:

(A) "Map" meaning

Map means like a dictionary:

- input → output
  Here, input is 4-part key, output is the value.

(B) "Multidimensional" meaning

Because the key has **multiple dimensions**:

1. RowKey
2. ColumnFamily
3. ColumnQualifier
4. Timestamp

(C) "Sorted" meaning

Sorted primarily by RowKey (lexicographically).

(D) "Sparse" meaning

HBase stores only what exists.
If a column doesn't exist in a row, no memory/disk is wasted.

Big exam line: HBase is best when different rows can have different columns.

---

4) Row Key (Your teacher LOVES this)

Slide says:

- Every row is identified by a single unique **row key**
- Data is sorted **lexicographically** by row key (not numeric sorting)
- Range scans are extremely fast due to this sorting
- RowKey is a byte array: `byte[]` (everything converted to bytes)

Lexicographic sorting (super important trap)

Lexicographic means dictionary-like sorting.

Slide gives example:
If numbers stored as strings:

- "1", "2", "10", "20"
  Sorted becomes:
- "1", "10", "2", "20"

But if stored as fixed-width binary (Bytes.toBytes(int)):

- 1, 2, 10, 20 properly sort numerically.

What can be a RowKey?

Slide lists: String, Integer, Long, Timestamp, UUID, composite keys, hashes, encoded binary keys — everything becomes byte[].

Why byte arrays?

Slide answers:

- Flexibility (client can encode any datatype)
- Performance (byte comparison is fast)
- No overhead (HBase doesn't interpret values)
- Efficient lexicographic sorting (regions split easily)

✅ **Exam tip:** If they ask "why byte[]?", write these 4 points.

---

5) Column Families (second huge pillar)

Slide says:

- HBase groups columns into **column families**
- CFs are fixed at schema definition (can't add/remove frequently)
- Data in the same CF is stored together on disk (same HFile) — **very important**
- Each column belongs to a CF: `cf:column`

What does "stored together" mean?

Slide explains:
For row user_001:

- CF personal stored in one HFile
- CF orders stored in another HFile
- CF activity stored in another HFile
  You can scan only one CF without touching others.

Story:
Imagine CF is like “folders”:

- personal folder
- orders folder
- activity folder
  If you only need “personal”, you don’t open the other folders → faster reads.

Creating CFs (syntax in slide)

You define CFs at table creation:
```
create 'users', 'personal', 'orders', 'activity'
```

CF impact on reads/writes (why CF is heavyweight)

Slide says each CF has its own:

- MemStore
- HFiles
- BlockCache
- Bloom filters

Also:

- Performance tuning is per-CF
- Compaction is CF-wise (big CF → heavy compaction)

Column family settings (must know)

Slide table:

- TTL: auto-delete after X seconds
- Max versions
- Compression (Snappy/GZip/LZ4)
- Block size (default 64KB)
- Bloom filters reduce disk seeks

Column family isolation (why split CFs)

Slide gives example:

- personal: small rarely updated → no compaction pressure
- orders: large frequently updated → separate compaction needed
- activity: high-write time-series → short TTL

How many column families?

Slide says: **DO NOT make many CFs**
Best practice:

- Max 2–3 CFs
- up to 5 risky
- 1–2 ideal

Why many CFs are bad:
each CF has separate MemStore/HFiles/WAL edits/flushes/compactions → wastes RAM and causes compaction storms.

---

6) Column Qualifiers (how HBase becomes “schema-less”)

Within a column family, you can create **unlimited dynamic columns** called qualifiers.

Slide says:

- qualifiers are not predefined; can vary row to row
- gives schema-less flexibility within each family

Another slide adds:

- you can create new qualifiers anytime while inserting (no schema change)
- different rows can have different qualifiers
- HBase stores only what exists → sparse table

Qualifiers are also stored as byte[] and sorted lexicographically.

Qualifiers are “lightweight”:

- don’t create new HFiles
- don’t create new MemStores
- don’t affect WAL structure
- can be millions per row (time-series)

✅ **Exam line:** Column families are heavyweight; qualifiers are cheap.

---

7) Versions + Timestamps (HBase's special power)

This is one of the most testable parts.

Cell uniqueness formula (again)

(row, cf, qualifier, timestamp) → value

Slide says:

- Versions are not new rows
- they are stored under same row key and same column, with different timestamps

What is a version?

A version is identified by timestamp.
Latest version returned by default.

Default versions + changing it

HBase keeps **1 version by default**.
You can change it per CF:
`alter 'users', NAME=>'info', VERSIONS=>3`

Server vs user timestamp

- If you don't provide timestamp → server assigns current time in ms
- You can set your own timestamp (useful for time-series, CDC, event logs, IoT, ordering)

Slides also explain Unix epoch basics (1970 baseline).

How versions stored in HFiles

Slide says:

- Stored sorted with most recent version first (fast read)
- Versioning helps: analytics at time X, audit, recovery, drift tracking, append-only log style

What is a “cell”?

Smallest unit:
(row key, cf, qualifier, timestamp) → value

---

8) HBase internal organization view (logical → internal tree)

Slide shows the structure:
RowKey → CF → Qualifier → Timestamp → Value

---

9) LSM Tree (how HBase achieves fast writes)

Slides explain HBase uses a Log-Structured Merge Tree.

Why LSM?

Optimized for **high write throughput**.

LSM write story

1. Writes first go to RAM (MemStore/MemTable)
2. Also appended to WAL for durability
3. MemStore keeps sorted in memory
4. When full → flush to disk as immutable file (HFiles/SSTables)
5. Over time many files accumulate → periodic merge/compaction

Slide says it’s fast because:

- sequential writes (append WAL, flush)
- minimized disk seeks
- reads aided by indexes, Bloom filters, compactions

---

10) Major benefits (why HBase is used)

Slide lists:

- sparse (different cols no storage penalty)
- wide (millions of columns)
- flexible schema (dynamic qualifiers)
- high write throughput (LSM)
- scalable (regions split and spread across region servers)

---

11) Physical model (how HBase stores things in cluster)

Slide says:
HBase Table → Regions → RegionServers → HFiles

(A) Region

A region is a **horizontal partition** of the table based on row-key range.
Example:

- Region1 rows "A" to "H"
- Region2 rows "H" to "Q"

Each region physically has:

- MemStore (RAM)
- Store (disk)
- HFiles
- WAL

(B) RegionServer

A RegionServer hosts many regions.
Each region has 1 or more stores (one store per column family).

(C) Store (important)

Column Family = Unit of storage.
For each CF, HBase creates a store with:

- MemStore
- HFiles on HDFS

---

12) Write path (THIS IS EXAM GOLD) — Put → WAL → MemStore → HFile

Slides explain this clearly:

When client writes (Put):

1. Write goes to **WAL** (crash recovery)
2. Write goes to **MemStore** (RAM)
3. ACK sent to client
4. When MemStore hits threshold → flush to HFiles on HDFS

MemStore is per column family per region.

What is WAL exactly?

Slide says:

- every write goes to WAL before hitting MemStore
- stored in HDFS as HLog files
- used to replay writes if RegionServer crashes
- prevents data loss

Crash scenario (very testable)

Slide gives scenario:

- inserted 1000 records, still in MemStore
- regionserver crashes
  Without WAL → lost.
  With WAL:
- master assigns region to another RS
- new RS reads WAL for that region
- WAL replay rebuilds MemStore then flushes to HFiles
- data restored

✅ If exam asks "why WAL?" write that story.

HFile ordering (important detail)

HFile is sorted:
RowKey → ColumnFamily → ColumnQualifier → Timestamp

---

13) Compaction (why reads stay fast over time)

Motivation slide says:

- frequent small writes → many HFiles in a column family
- reads check all HFiles → slow
- deleted/old versions still occupy space
  Compaction merges and cleans HFiles

Minor compaction

- merges a small number of HFiles
- keeps old versions
- triggered when HFiles exceed threshold (default 3)

Major compaction

- heavier, can impact performance
- removes tombstones (deleted markers)
- removes expired versions (TTL/versions)
- creates a single large HFile per CF per region

---

14) Compression (different from compaction)

Slide says:

- compression reduces disk usage/I/O by storing data compressed inside HFiles
- codecs: LZO, LZ4, Snappy, GZIP
- applied per column family

- done when writing HFiles (flush)
- does NOT merge files or remove old versions

And it gives comparison table (ratio vs speed vs CPU).

---

15) Cache (BlockCache + MemStore) + eviction policy

Slide says:

- Block Cache stores hot data blocks read from HFiles (64KB blocks)
- MemStore caches data written but not flushed
- eviction policy: LRU (least recently used)

---

16) HBase Architecture components (Master, RegionServer, Client)

Slide says 3 major components:

- HBaseMaster (one master)
- HRegionServer (many region servers)
- HBase client

Master responsibilities:

- coordinates slaves
- assigns regions, detects failures
- admin functions

RegionServer:

- manages regions
- serves reads/writes
- uses log (WAL)

---

17) HBase vs HDFS (where each is good)

Slide says:
HDFS:

- good for batch processing (scans over big files)
- not good for record lookup, small incremental adds, updates

HBase:

- excellent for fast random reads by row key (ms lookup)
- high throughput for large datasets

Not good for:

- ad-hoc lookups by non-key columns
- secondary indexes (not built in)
- random scans over large ranges (slow/expensive)

One clean line from slide:
HBase is good for key-value pattern lookups, not “query anything anywhere.”

---

18) Write strategies (Random, Bulk load, Incremental)

Slide says:

Random writes

HBase is well suited for random writes due to LSM:
MemStore → flush to HFiles.

Bulk load

For huge ingestion:

- prepare sorted HFiles offline
- directly load into regions
- efficient for ETL pipelines

Incremental load

Small updates in real-time:

- events streamed incrementally
- each write results in a new version

Also slide warns: small writes cause RPC overhead, frequent flushes, many tiny HFiles, compaction overhead → better buffer updates and write in bulk.

---

19) Read strategy (Get, Scan + caches + bloom filters)

Slide says:

- **Get** retrieves a single row by row key
- **Scan** retrieves multiple rows in a row-key range

Block cache:

- check cache first before disk

Bloom filters:

- quickly determine if row/column exists in HFile without reading entire file (reduces disk seeks)

---

20) Operations (Put / Delete) + Versions

Slide says:

- Put can insert new record or update existing key
- implicit timestamp (server) or explicit timestamp

Delete:

- marks cells as deleted (tombstones)
- can delete at multiple levels: entire CF, all CFs of row, etc.

Joins:

- HBase does not support joins
- can be done in application layer using get/scan

Exam-ready “3 point” answers (most likely questions)

Q1) Why HBase uses RowKey sorting?

1. Rows sorted lexicographically by RowKey
2. Fast point lookup using exact key
3. Range scans become extremely fast due to sorting

Q2) Explain Column Family vs Column Qualifier

- CFs are fixed in schema and stored together in one HFile (heavyweight).
- Qualifiers are dynamic, schema-less, cheap, can be millions.
- CF design affects compaction/caching/performance.

Q3) Explain write path (Put)

1. Client sends Put to RegionServer → append to WAL
2. Write stored in MemStore (RAM)
3. Later flush MemStore → HFile on HDFS; WAL enables replay on crash

Q4) Minor vs Major Compaction

- Minor: merge few HFiles, reduce file count, keeps old versions
- Major: heavy, merges all, removes tombstones + expired versions (TTL/versions)
- Needed because frequent writes create many small HFiles and slow reads