BIG DATA ANALYTICS (CS-431)

Dr. Sriparna Saha
Associate Professor

Website: https://www.iitp.ac.in/~sriparna/
Google Scholar: https://scholar.google.co.in/citations?user=Fj7jA_AAAAAJ&hl=en
Research Lab: SS_Lab
Core Research AREA: NLP, GenAI, LLMs, VLMs, Multimodality, Meta-Learning, Health Care, FinTech, Conversational Agents

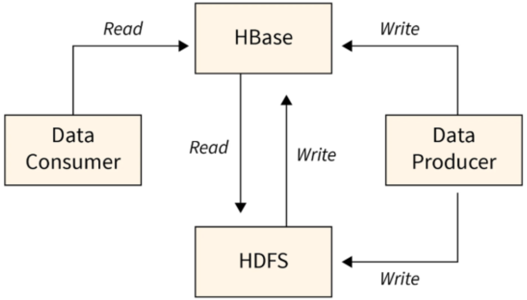
TAs: Sarmistha Das, Nitish Kumar, Divyanshu Singh, Aditya Bhagat, Harsh Raj

What is Apache HBase?

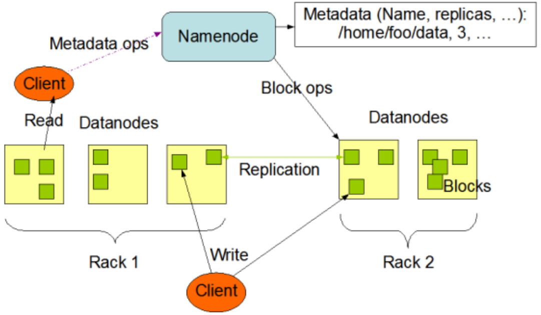
HBase is a **NoSQL, column-oriented** database that runs on top of HDFS. It provides **real-time read/write** access to large datasets.

Key Points:

- Built on top of HDFS (HDFS is the primary **distributed file system** used by Hadoop to store large volumes of data across multiple machines. It is designed for **high-throughput** access to large files.), uses it for storage.
- Ideal for **random, real-time access** to big data
- Suitable for use cases like messaging platforms, analytics engines, etc.
- Schema-less, supports millions of rows and columns.

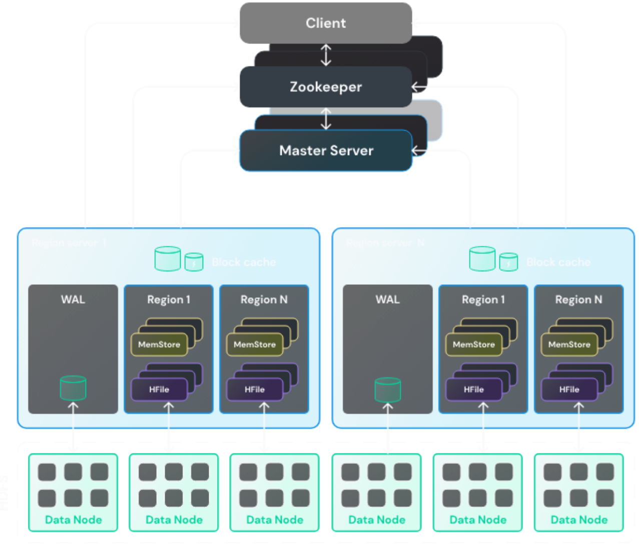


HDFS Architecture



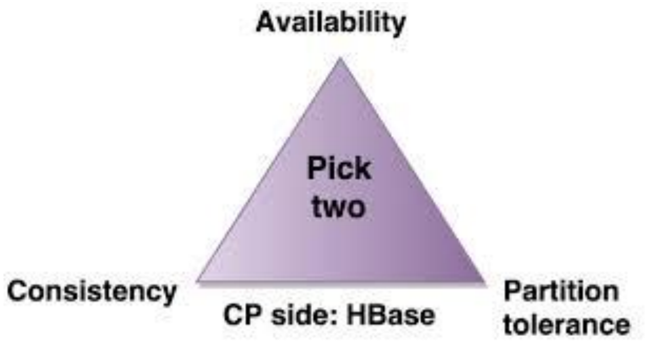
HBase architecture

- **HBase architecture** is based on the following components:
  a. **Region Server.** It serves one or more Regions — ranges of rows, stored together. Each Region is served only by one Region Server. Region Servers are also called HRegionServers. A Region Server contains multiple components, some of them work on top of HDFS, using it as a persistent data storage.
  b. **Master server.** It is a main server responsible for managing an HBase cluster. It is similar to a NameNode in HDFS. A Master server manages distribution of Regions between Region Servers, maintains the registration of Regions, etc. It is also called HMaster. You can deploy several Master servers in your cluster: one active and one or more standby.
  c. **ZooKeeper.** It is a special service designed to manage configurations and synchronization of services. It is used to coordinate actions between HBase services.



HBase and the CAP Theorem

- HBase is a **CP (Consistency + Partition Tolerance)** system.
- It guarantees that a client will always read the most recently written value.
- This strong consistency is achieved by having a single RegionServer responsible for a given data range.
- This makes it a good fit for Big Data use cases where data accuracy is a strict requirement.



The HBase Data Model

HBase stores data in a **table-like structure**, but it's very different from traditional databases.

- **Structure:**

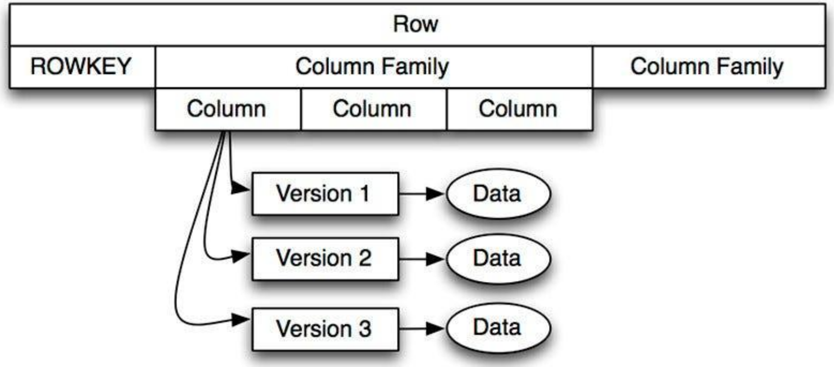
```
Table → Rows → Column Families → Columns → Versions (timestamps)
```

- **Row**: Identified by a unique **Row Key**.
- **Column Family**: A group of related columns stored together.
- **Version**: Each data cell can have multiple versions (by timestamp).

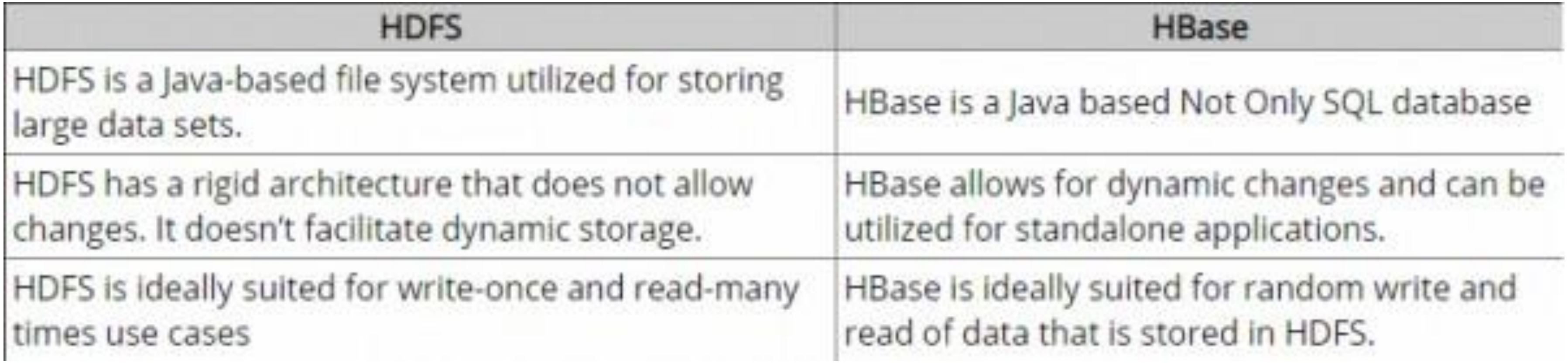
Designed for **scalability** and **sparse datasets** — good when not every row has all columns.

The Importance of the Row Key

- The **Row Key** uniquely identifies a row.
- Data is **sorted** by Row Key for fast lookups.
- It controls **data distribution** across the cluster.
- A **bad Row Key design** can slow down performance (e.g., timestamp-only keys = hotspotting).
- The **Row Key** helps HBase find data quickly because all rows are **sorted by Row Key**, enabling fast lookups and efficient data access.



HBase vs. HDFS



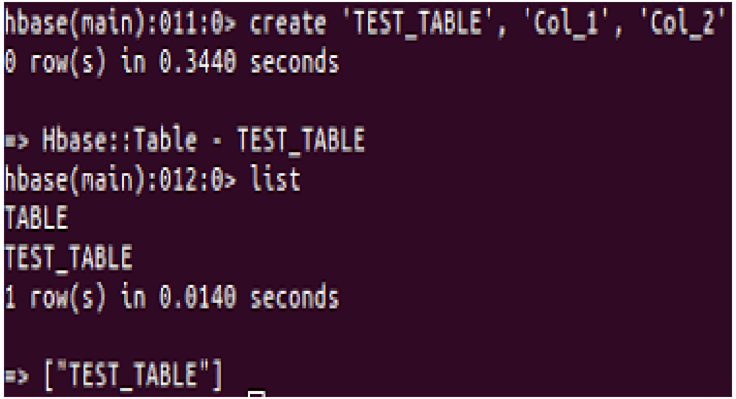
HDFS	HBase
HDFS is a Java-based file system utilized for storing large data sets.	HBase is a Java based Not Only SQL database
HDFS has a rigid architecture that does not allow changes. It doesn't facilitate dynamic storage.	HBase allows for dynamic changes and can be utilized for standalone applications.
HDFS is ideally suited for write-once and read-many times use cases	HBase is ideally suited for random write and read of data that is stored in HDFS.

Interacting with HBase

You can work with HBase through:

- `hbase shell` (command line)
- **Java API** (main programmatic access)
- **REST/Thrift/Avro** (for web or external systems)
- Python (e.g., using `happybase`)

Use commands like: `put`, `get`, `scan`, `delete`.



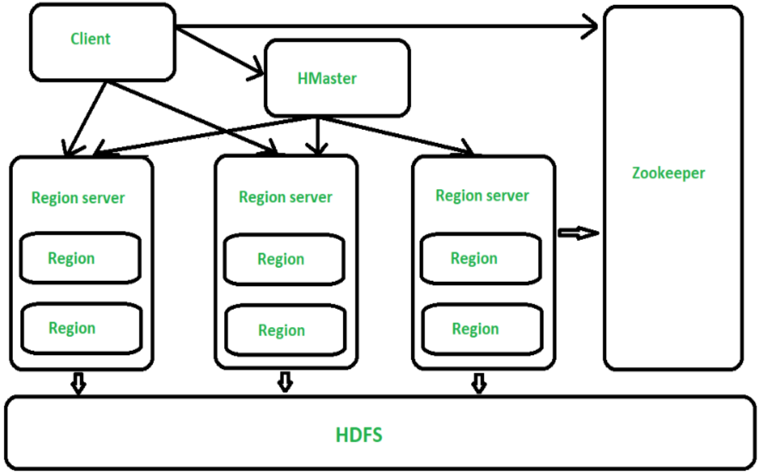
```
hbase(main):011:0> create 'TEST_TABLE', 'Col_1', 'Col_2'
0 row(s) in 0.3440 seconds

=> Hbase::Table - TEST_TABLE
hbase(main):012:0> list
TABLE
TEST_TABLE
1 row(s) in 0.0140 seconds

=> ["TEST_TABLE"]
```

Master-Slave Architecture

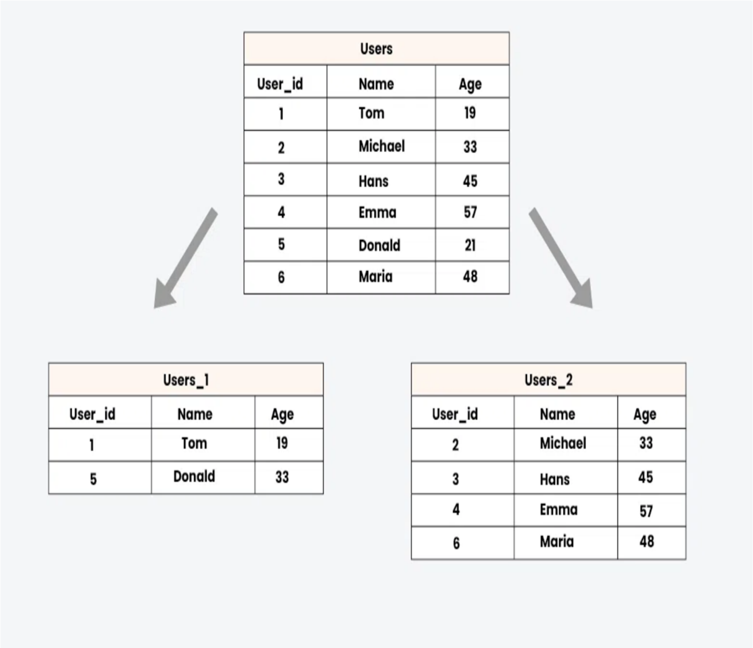
- HBase uses a master-slave architecture, coordinated by ZooKeeper.
- The **HMaster** is responsible for administrative tasks and assigning data regions to servers (Manages metadata and region assignments).
- The **RegionServers** are the workhorses; they handle all client read and write requests.
- This clear division of labor is a classic architecture for managing large-scale data systems.
- Only **one active HMaster** at a time; others are **standby** for failover.



Regions: Horizontal Data Partitioning

- A **Region** stores a range of rows from a table.
- Tables are **split horizontally** into Regions as they grow.
- Each **RegionServer** can serve multiple Regions.

This allows HBase to **scale out** as data increases.



Users

User_id	Name	Age
1	Tom	19
2	Michael	33
3	Hans	45
4	Emma	57
5	Donald	21
6	Maria	48

Users_1

User_id	Name	Age
1	Tom	19
5	Donald	33

Users_2

User_id	Name	Age
2	Michael	33
3	Hans	45
4	Emma	57
6	Maria	48

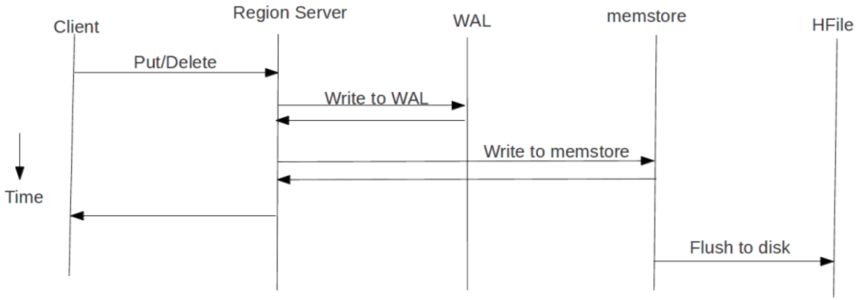
The Write Path: WAL (Write-Ahead Log)

WAL (Write-Ahead Log)

- Logs every write **before** saving data.
- Stored in **HDFS** to keep data safe if a server fails.
- Helps **recover data** after crashes.
- Fast because it only **adds to the end** of the log.

Purpose: If a RegionServer crashes, WAL ensures no data is lost.

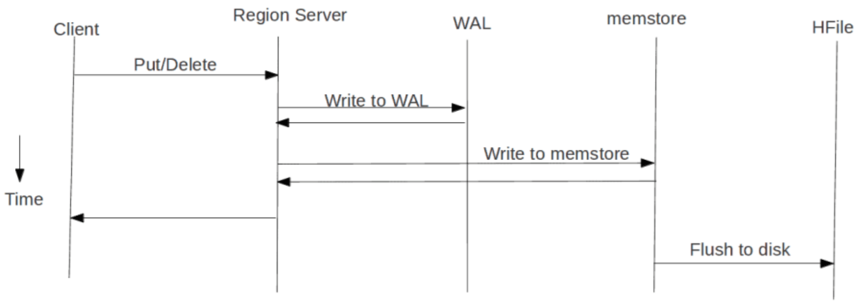
WAL = No data loss



HBase Write Path

The Write Path: MemStore

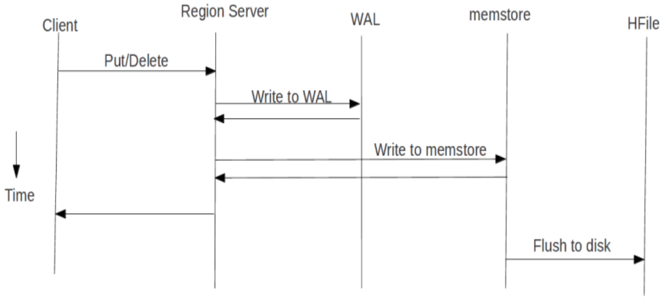
- Temporary storage **in memory** for new data.
- Keeps data **sorted** for quick access.
- When full, data is saved to disk as an HFile.
- Once MemStore reaches a size limit, it's **flushed** to disk as an HFile.
  - First, HBase checks if the latest data is in **memory** (MemStore).
  - If found, it's returned directly (fastest access).



HBase Write Path

The Write Path: HFiles (on HDFS)

- Permanent data files stored on **HDFS**.
- Created by flushing MemStore.
- Files are **sorted and unchangeable** for fast reads.
- Multiple files get merged over time to improve speed.
- Over time, multiple HFiles are **compacted** to reduce read overhead.
    - If not found in MemStore or BlockCache, HBase reads from **HFiles** on **HDFS**.
    - Data is merged from **multiple HFiles and versions** if needed.
    - May involve **Bloom Filters** and **Block Indexes** to speed up lookups.



HBase Write Path

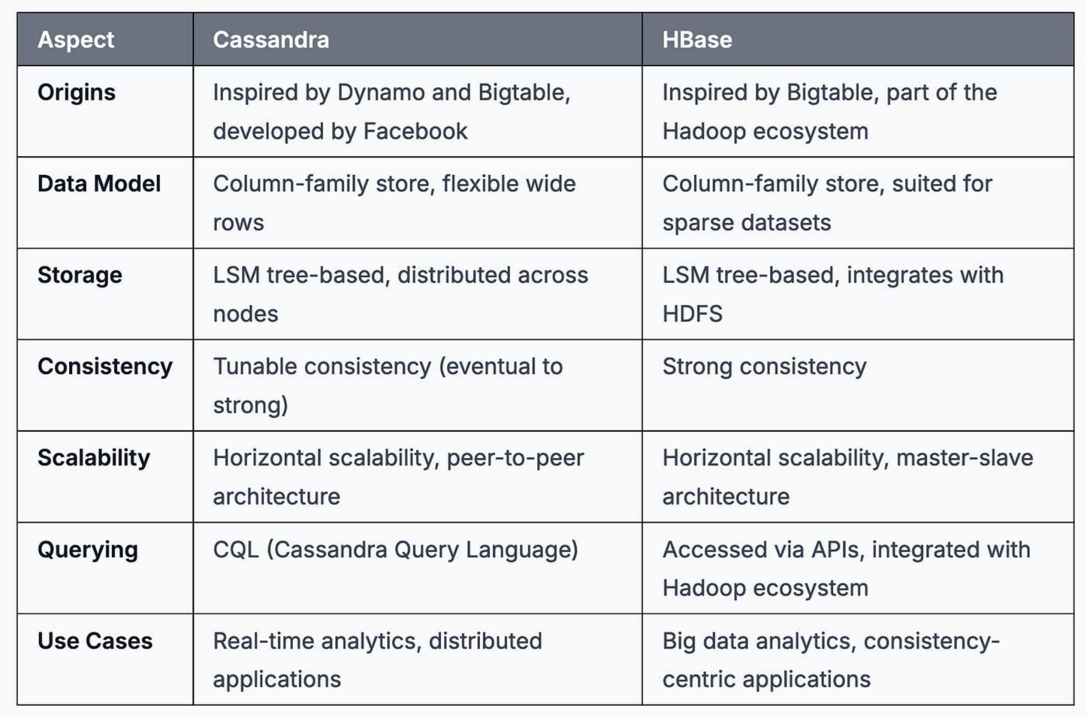
The Read Path

HBase reads data in this order: **memory → cache → disk**, ensuring the most recent and fastest access possible.

- To fulfill a read request, a **RegionServer first checks the MemStore** for the latest data.
- If the data isn't in MemStore, it checks the **BlockCache**, which stores frequently accessed data blocks from HFiles.
- If not found there, HBase reads from **HFiles on HDFS** using **Bloom Filters** and **block indexes** to speed up access.

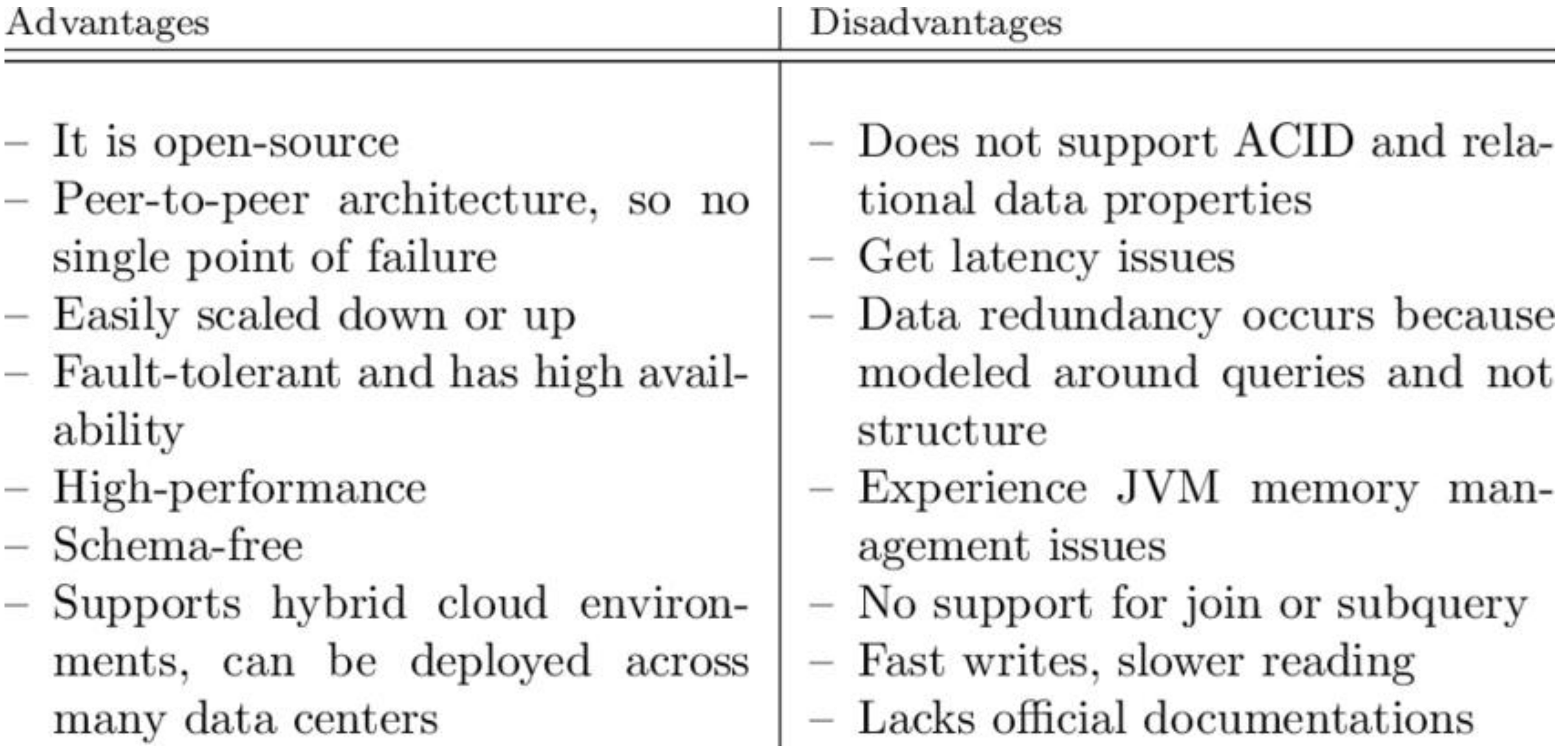
This **multi-layered read path** ensures **low-latency performance**, making HBase ideal for **real-time applications** like analytics dashboards.

Cassandra vs. HBase



Aspect	Cassandra	HBase
Origins	Inspired by Dynamo and Bigtable, developed by Facebook	Inspired by Bigtable, part of the Hadoop ecosystem
Data Model	Column-family store, flexible wide rows	Column-family store, suited for sparse datasets
Storage	LSM tree-based, distributed across nodes	LSM tree-based, integrates with HDFS
Consistency	Tunable consistency (eventual to strong)	Strong consistency
Scalability	Horizontal scalability, peer-to-peer architecture	Horizontal scalability, master-slave architecture
Querying	CQL (Cassandra Query Language)	Accessed via APIs, integrated with Hadoop ecosystem
Use Cases	Real-time analytics, distributed applications	Big data analytics, consistency-centric applications

HBase: advantages and disadvantages



Advantages	Disadvantages
– It is open-source	– Does not support ACID and relational data properties
– Peer-to-peer architecture, so no single point of failure	– Get latency issues
– Easily scaled down or up	– Data redundancy occurs because modeled around queries and not structure
– Fault-tolerant and has high availability	– Experience JVM memory management issues
– High-performance	– No support for join or subquery
– Schema-free	– Fast writes, slower reading
– Supports hybrid cloud environments, can be deployed across many data centers	– Lacks official documentations