Comprehensive Lecture Notes: Apache HBase

Distributed, Scalable, Big Data Store

Contents

1 Overview of HBase

1.1 Definition and Ecosystem Position

Apache HBase is a distributed, column-oriented, non-relational database management system modeled after Google's Bigtable. It is an open-source project designed to run on top of the Hadoop Distributed File System (HDFS).

- **Storage Layer:** While HDFS provides a file system for storing large files (Write-Once-Read-Many), HBase provides a logical layer to organize this data into tables, rows, and columns, enabling random real-time read/write access.
- **Ecosystem Integration:** HBase is a native part of the Hadoop ecosystem. It often sits below processing layers like MapReduce, Hive, and Pig, and above the storage layer (HDFS). ZooKeeper provides the coordination service required to manage the distributed state.

1.2 HBase vs. HDFS vs. RDBMS

To understand HBase, it is crucial to understand what it is *not*.

1. **HBase vs. HDFS:**
   - **HDFS** is optimized for batch processing and high-throughput streaming of large files. It does not support fast record lookups or incremental updates.
   - **HBase** facilitates fast random reads and writes (record lookups) and manages updates via versioning, making it suitable for real-time applications.
2. **HBase vs. RDBMS:**
   - HBase is **not** a relational database. It does not support SQL (natively), joins, or complex transactions spanning multiple rows.
   - Instead of a fixed schema, HBase offers a flexible data model (dynamic columns) and scales horizontally to billions of rows across thousands of nodes.

2 The HBase Data Model

HBase is logically viewed as a table, but physically, it is a sparse, distributed, multi-dimensional, sorted map. The data is indexed by a specific key structure:

$$(\text{RowKey}, \text{ColumnFamily}, \text{ColumnQualifier}, \text{Timestamp}) \rightarrow \text{Value}$$

2.1 Row Key

The Row Key is the unique identifier for a row in an HBase table.

- **Byte Arrays:** Row keys are stored as raw byte arrays (`byte[]`). This allows any data type (String, Integer, Long, complex objects) to be used as a key once serialized.
- **Lexicographical Sorting:** Data in HBase is sorted lexicographically (byte-by-byte) by the Row Key. This is the primary indexing mechanism.
- **Performance Implication:** Because data is sorted, range scans (e.g., scan rows from "User_A" to "User_Z") are extremely fast. However, poor key design (e.g., monotonically increasing keys) can lead to "hotspotting," where all writes go to a single server.

2.2 Column Families (CF)

Columns in HBase are grouped into Column Families.

- **Physical Storage Unit:** All columns within a single Column Family are stored together physically on the disk in the same files (HFiles).
- **Schema Definition:** Column Families must be defined when the table is created and cannot be changed frequently.
- **Tuning:** Configuration settings such as compression, data retention (TTL), and caching are applied at the Column Family level.
- **Best Practice:** A table should have a small number of Column Families (ideally 1 to 3). Too many CFs can degrade performance due to write buffer splitting and compaction storms.

2.3 Column Qualifiers

Inside a Column Family, data is addressed using Column Qualifiers.

- **Dynamic Schema:** Unlike Column Families, Qualifiers are not defined in the schema. They can be created on the fly during data insertion.
- **Flexibility:** Different rows can have completely different sets of Column Qualifiers.
- **Syntax:** They are referenced as `Family:Qualifier` (e.g., `info:name`, `info:email`).

2.4 Cells and Versioning

A unit of data in HBase is called a **Cell**.

- **Versioning:** A cell is not just a single value; it can hold multiple versions of a value, indexed by a **Timestamp**.
- **Timestamp:** This is a `long` integer representing the time of insertion (usually milliseconds since the Unix Epoch). It can be system-generated or user-defined.
- **Ordering:** Within a cell, versions are stored in descending order of timestamp. By default, a read operation returns the latest version.
- **Configuration:** Users can define how many versions to keep (e.g., `VERSIONS => 3`) or a Time-To-Live (TTL) for automatic deletion of old data.

3 HBase Physical Architecture

HBase follows a Master-Slave architecture.

3.1 HMaster (The Master)

- Responsible for administrative operations.
- Assigns regions to RegionServers.
- Handles load balancing and failure recovery (detecting crashed RegionServers).
- It is *not* part of the read/write path for client data, preventing it from becoming a bottleneck.

3.2 HRegionServer (The Worker)

- Handles the actual data storage and retrieval.
- Clients communicate directly with RegionServers to read or write data.
- A single RegionServer manages multiple **Regions**.

3.3 Regions

- **Horizontal Sharding:** A table is horizontally partitioned into Regions based on Row Key ranges (e.g., Row A to Row H).
- **Scalability:** As a table grows, regions split automatically and are distributed across different RegionServers.

3.4 ZooKeeper

- Acts as the coordination service ("the glue").
- Maintains the state of the cluster (which servers are alive).
- Stores the location of the `hbase:meta` table (which tells clients where to find specific user data).

4 Storage Internals and The Write Path

HBase utilizes an **LSM Tree (Log-Structured Merge Tree)** architecture, which is optimized for high-throughput random writes.

4.1 The Write Process

When a client issues a `Put` request:

1. **WAL (Write-Ahead Log):** The data is first appended to the WAL on HDFS. This ensures durability; if the server crashes, the log can be replayed to recover lost data.
2. **MemStore:** The data is then written to the MemStore, an in-memory buffer. Here, data is sorted by Row Key.
3. **Acknowledgement:** Once in the WAL and MemStore, the write is acknowledged to the client.
4. **Flush:** When the MemStore reaches a configured threshold (e.g., 128 MB), it is flushed to disk, creating a new immutable file called an **HFile**.

4.2 HFiles

- HFiles are the physical storage files in HBase, stored in HDFS.
- They contain sorted key-value pairs.
- Because HFiles are immutable (cannot be modified), updates and deletes are handled as new records (appends) with newer timestamps or "tombstone" markers for deletion.

5 The Read Path and Optimization

Reading from HBase involves merging data from the MemStore (recent writes) and HFiles (historical data).

5.1 Read Optimization Mechanisms

- **BlockCache (L1 Cache):** Frequently accessed HFile blocks are cached in the RegionServer's RAM (BlockCache). A read request checks the BlockCache first.
- **MemStore Check:** Since the latest data might not be on disk yet, the system checks the MemStore.
- **Bloom Filters:** These are probabilistic data structures that allow HBase to quickly determine if a specific row or column *definitely does not exist* in an HFile. This prevents unnecessary disk seeks.

6 Compaction

Because writes (and updates/deletes) create many small HFiles, performance degrades over time (reads have to check too many files). Compaction is the maintenance process to fix this.

6.1 Minor Compaction

- Merges a few small adjacent HFiles into a single larger HFile.
- Does not drop deleted records or expired versions.
- Low impact on performance.

6.2 Major Compaction

- Merges *all* HFiles in a region into a single file per Column Family.
- **Cleanup:** Physically removes deleted records (tombstones) and expired versions.
- **Performance Cost:** This is a resource-intensive operation and is usually scheduled during off-peak hours.

7 Data Loading Strategies

7.1 Standard Writes (Random Access)

Using the ‘Put()‘ API. While fast, putting millions of records one by one involves significant RPC (Remote Procedure Call) overhead and pressure on the MemStore.

7.2 Bulk Loading

For massive data ingestion (e.g., initial migration), Bulk Loading is preferred.

1. Data is prepared and sorted offline using a MapReduce job.
2. The output is formatted directly into HFiles.
3. These HFiles are "moved" directly into the live HBase region.
4. This bypasses the Write path (WAL and MemStore) entirely, saving CPU and RAM.

8 Summary of Operations

- **Get():** Returns attributes for a specific row key. Fast point lookup.
- **Put():** Adds a new row or updates an existing row. (Updates are just new versions).
- **Scan():** Iterates over a range of rows. Used for analytics or retrieving multiple records.
- **Delete():** Writes a "tombstone" marker. Data is not physically removed until a Major Compaction occurs.

9 Best Practices and Design Considerations

1. **Row Key Design:** Crucial for performance.
    - Avoid monotonically increasing keys (like timestamps 1, 2, 3...) to prevent all writes hitting one RegionServer (hotspotting).
    - Use hashing or salting to distribute writes evenly.
2. **Schema Design:**
3. Keep Column Family names short (they are stored in every cell, adding overhead).
4. Use fewer Column Families (1-3) to ensure efficient flushing and compaction.
5. **Updates:** Buffer small updates and batch them to reduce RPC overhead.