GUIDE

Organization of Dynamic Update and Delete operations on HDFS files in Multi-DataNode Big Data system

Contents

1. Essence of HDFS file system and possibility to dynamic change the content

At its core, the Hadoop Distributed File System (HDFS) is designed for "write-once, read-many" processing. It's built to handle massive datasets by spreading them across clusters of commodity hardware, prioritizing high throughput over low latency. Big Data Systems (BDS) using mainly Apache Hadoop software is an open source framework allows for the distributed storage and processing of large datasets across clusters of computers (multi nodes as DataNodes) using simple programming models. BDS-Hadoop is designed to scale up from a single computer to thousands of clustered nodes-computers, with each machine offering local computation and storage. In this way, BDS-Hadoop can efficiently store and process large datasets ranging in size from gigabytes to petabytes of data. Hadoop Distributed File System (HDFS) is a file system that manages large data sets that can run on commodity hardware. HDFS is the most popular data storage system for Hadoop and can be used to scale a single Apache Hadoop cluster to hundreds and even thousands of nodes. Because it efficiently manages big data with high throughput, HDFS can be used as a data pipeline and is ideal for supporting complex data analytics.

There is one big advantage of the HDFS file system - Fault tolerance and fast recovery from hardware failures. Because one HDFS instance might consist of thousands of servers, failure of at least one server is always a possibility. HDFS has been built to detect faults and automatically recover quickly. Data replication with multiple copies across many nodes helps protect against data loss. HDFS keeps at least one copy on a different rack from all other copies. This data storage in a large cluster across nodes increases reliability. In addition, HDFS can take storage snapshots to save point-in-time (PIT) information. HDFS is intended more for batch processing versus interactive use, so the emphasis in the design is for high data throughput rates, which accommodate streaming access to data sets. Because the data is stored virtually, the costs for file system metadata and file system namespace data storage can be reduced.

HDFS uses a cluster architecture to help deliver high throughput. To reduce network traffic, the Hadoop file system stores data in DataNodes where computations take place, rather than moving the data to another location for computation. With both horizontal and vertical scalability features, HDFS can be quickly adjusted to match an organization's data needs. A cluster might include hundreds or thousands of nodes.

The essence of HDFS file system architecture can be presented as follow. HDFS isn't a "real-time" file system like the one in general purpose computer. It operates on a few fundamental principles:

- Streaming Data Access: It's optimized for batch processing rather than interactive use. The goal is to move the computation to the data, not the other way around.
- Fault Tolerance: Generally, the hardware can and will fail. It handles this by splitting files into blocks (typically 128MB or 256MB) and replicating those blocks across different nodes.

- Master/Slave Architecture:
    - NameNode: The "brain" that manages metadata (where files are located).
    - DataNodes: The "muscle" that stores the actual data blocks

In the figure 1 is presented the workflow of creation of HDFS file, while in figure 2 is presented the workflow for reading HDFS file.

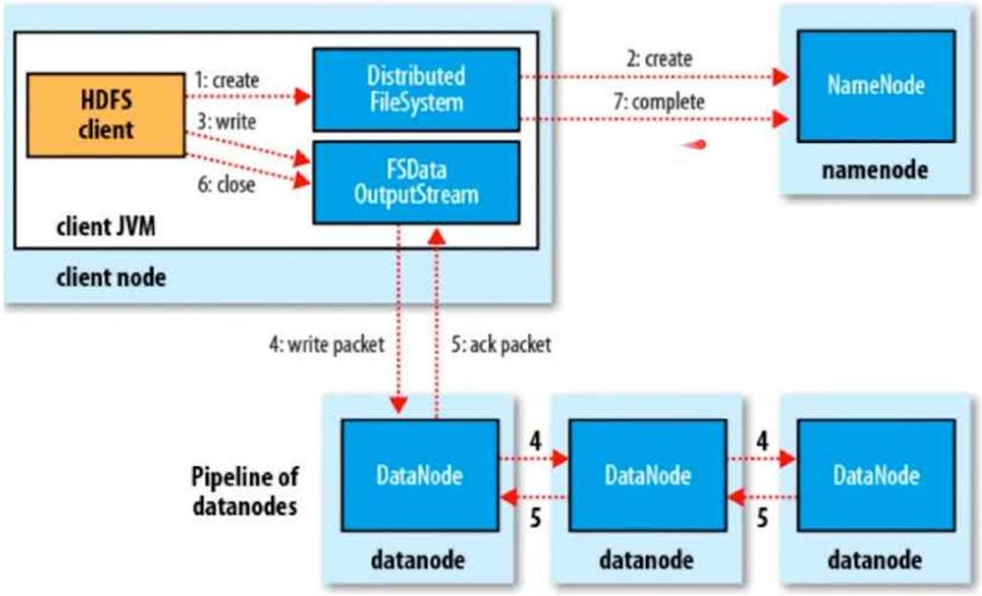

Figure 1

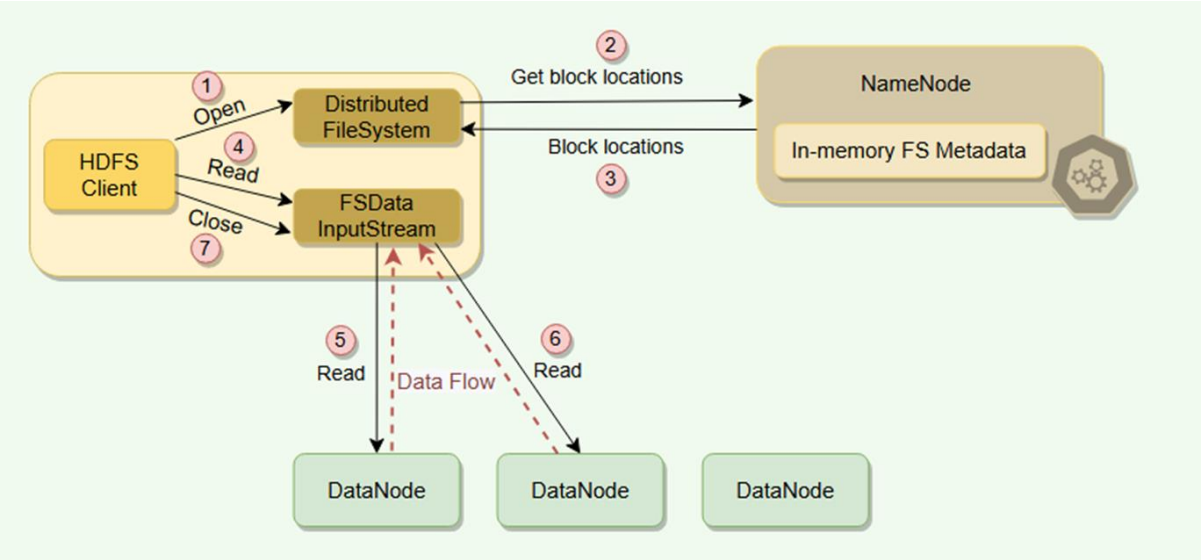

Figure 2

In HDFS file system there is no possible to dynamically change the content in the traditional sense. Unlike a standard Linux file system where it is possible to open a file and edit a line in the middle, HDFS has strict limitations:

1. Immutability: Once a file is written and closed, its existing content cannot be modified. You cannot perform "in-place" updates.
2. Appends Only: HDFS does support appends. It is possible to add new data to the end of an existing file, but cannot go back and fix a typo or change a value in the first 10MB of a 1GB file.
3. The "Rewrite" Workaround: If it is required to change the content, the standard practice in the Hadoop ecosystem is to:
    - Read the original file.
    - Apply the changes in memory or during a process like MapReduce/Spark job.
    - Write a brand-new file and delete the old one.

This design choice is intentional. Maintaining data consistency across a distributed cluster is incredibly difficult. By making files immutable (or append-only), HDFS avoids the massive overhead of locking mechanisms that would be required if multiple users tried to edit the same block simultaneously.

2. Example of updating a HDFS file for video data where a record is a few terabytes

When a **single HDFS "record" is many terabytes** (e.g., the HDFS file is a multi-TB video file), the idea of "updating" it becomes fundamentally different from updating a normal database record. HDFS is **append-only**, **immutable**, and optimized for **large sequential reads**, not in-place updates.

To make this concrete, here's how an update typically looks **as a process** and **as time**, in real-world architectures used for multi-TB video data.

2.1. Why cannot update a multi-TB HDFS file in place

HDFS stores files in **128 MB blocks by default** (or 256/512 MB in large clusters). If a multi-TB video is:

- 10 TB video
- Block size: 256 MB
- This leads to about 40,000 HDFS blocks

If it is required to "update" even **1 second** in the middle of the video, HDFS forces to:

- rewrite the entire file
- or rewrite all blocks after the changed region

This is why **in-place updates do not exist** generally.

2.2. What "update" means in practice for multi-TB video on HDFS

There are two realistic patterns:

2.2.1. A. Full rewrite (brute-force)

Process

1. Read the original video from HDFS
2. Apply the update (e.g., re-encode, replace segment)
3. Write a new full video file to HDFS
4. Swap metadata pointer (atomic rename)
5. Delete old file asynchronously

Time cost

1. Rewrite time ≈ size / throughput

   In this case: 10 TB / 2 GB/s = the update will be about 1.4 hours

2. Plus, encoding time if needed

This is the simplest but extremely expensive – for processing and for time point of view.

2.2.2. B. Chunked storage + partial rewrite (recommended)

Instead of storing a 10 TB video as one file, it is possible to store it as many chunks:

The following table demonstrate possible identification of a separate chunk.

Video_file__id	chunk_id	hdfs_path
42	0	/v/42/0
42	1	/v/42/1
...	...	...

Process for update

1. Identify which chunk(s) contain the region to update
2. Rewrite only those chunks
3. Update metadata in other Hadoop software component - HBase / Hive
4. Leave all other chunks untouched

Time cost

1. Rewrite time ≈ size of affected chunks
2. If each chunk is 100 MB and the change requires update in 3 chunks → rewrite 300 MB
3. Metadata update will be in milliseconds

This is the only standard scalable way to support updates.

2.3. Best practice for HDFS file architecture of multi-TB video

This architecture is used by Netflix, YouTube, and large research clusters handle multi-TB video.

Storage organization:

- HDFS stores chunks
- Chunk size: 64–256 MB
- Directory per single video
- Optional used supporting software: Parquet/ORC for metadata

Metadata organization:

- Software component - HBase
- Stores the following components: chunk index, offsets, durations, timestamps

Access layer - A service that read:

- maps timestamps → chunks

- assembles a few chunks for playback
- handles updates atomically of that chinks

Service - Update pipeline

- Chunk or chunks extraction
- Re-encoding
- Write new chunk/chunks
- Update manifest

3. Similar to index sequential organization for HDFS files

In HDFS, there is no 1:1 "out of the box" equivalent to a traditional Indexed Sequential Access Method (ISAM) because HDFS is a write-once, append-only file system designed for high-throughput streaming rather than random access. However, there are several architectures and file formats that provide the same benefits - namely, the ability to read data sequentially or jump to specific records using an index.

3.1. Columnar Formats with Built-in Indexes (The Standard)

The most common way to achieve "indexed sequential" behavior in HDFS is by using **Apache Parquet** or **Apache ORC**. These formats organize data into "Row Groups" or "Stripes."

- **Sequential Benefit:** Data within a column is stored contiguously, making it incredibly fast for sequential scans.
- **Index Benefit:** They contain metadata (min/max values, offsets) at the footer of the file. This allows the execution engine to skip entire blocks of data that don't match your query criteria.

A) Apache Parquet

Apache Parquet is a columnar storage format designed to bring efficiency to big data processing. While HDFS is just a distributed file system that sees data as "blocks," Parquet organizes that data internally to allow for high-performance querying, specifically mimicking some behaviors of indexed sequential access. Here is the breakdown of its essence regarding data organization.

- Columnar Layout (The Core Shift)

Unlike traditional "Row-Oriented" formats (like CSV or Avro), Parquet stores values of the same column together. This is the "sequential" part of its efficiency—reading a single column means the disk head moves linearly through that specific data type, rather than skipping over irrelevant fields.

- The Internal Hierarchy

Parquet doesn't just dump columns into a file; it organizes them into a structured hierarchy that facilitates "indexing-like" behavior:

- Row Groups: The file is split into large horizontal chunks. Each row group contains data for a subset of rows.
- Column Chunks: Within a row group, data for each column is stored together.
- Pages: Column chunks are further divided into pages. This is the smallest unit of compression and reading.

- Metadata: The "Virtual Index"

Parquet doesn't use a separate index file (like a B-Tree in a SQL database). Instead, it embeds Metadata at the end of the file (the "Footer"). This metadata contains:

- Min/Max Statistics: For every column chunk and page, Parquet stores the minimum and maximum values.
- Offset Indexes: Tells the reader exactly where a specific column starts in the file.

The data organization for Apache Parquet is presented in figure 3.

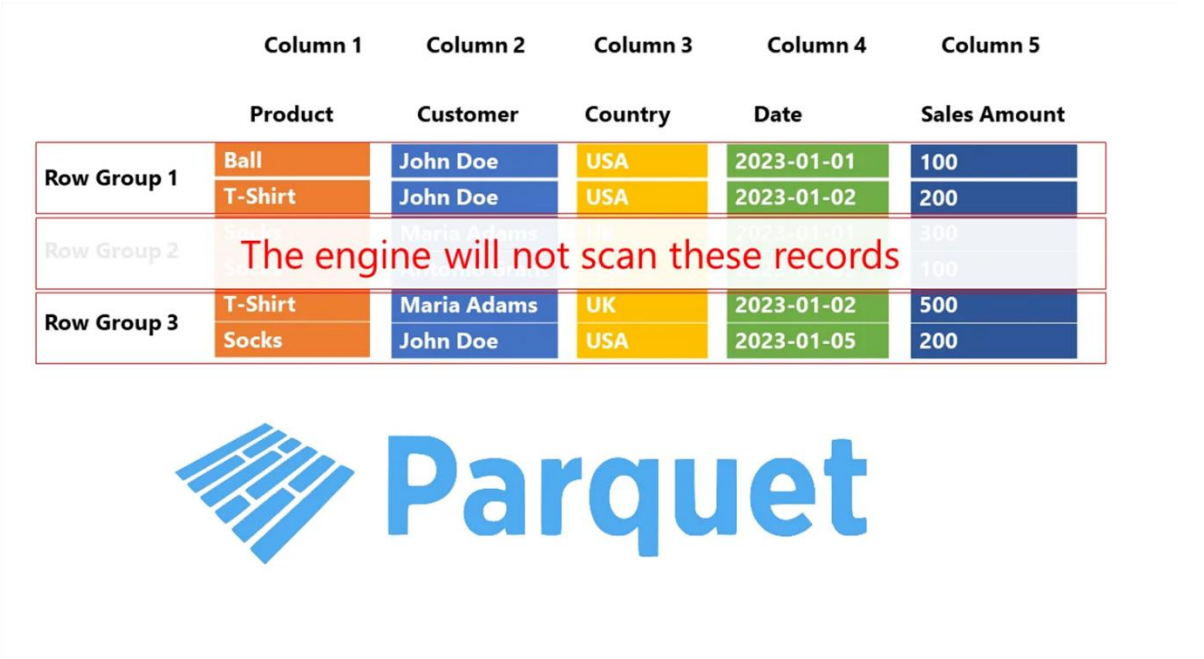

Figure 3

Generally, HDFS is optimized for large, sequential reads. Parquet aligns perfectly with this because: Since values in a column are of the same type (e.g., all integers), they compress much better than mixed-row data. Reduced I/O: By using metadata to skip data (indexing) and only reading required columns (sequential), you drastically reduce the network traffic between the HDFS DataNodes and the compute engine (Spark/Presto). In this way, Apache Parquet is good for HDFS file where the stricture of each record are similar. A typical example of not-suitability is for HDFS file containing video/audio records.

B) Apache ORC

While Apache Parquet is the generalist of the big data world, Apache ORC (Optimized Row Columnar) is the specialist, specifically engineered for the Hadoop/Hive ecosystem. Its "essence" lies in being type-aware and offering even more aggressive indexing and compression for structured data.

ORC organizes data into large horizontal chunks called **Stripes**, which are the equivalent of Parquet's "Row Groups" but typically larger (default 250 MB). This size is optimized for HDFS to enable large, streaming reads.

- Stripes: Each stripe contains its own index data, row data, and a footer. This makes stripes independent and easily "splittable" for parallel processing across a cluster.
- Row Groups: Within a stripe, rows are grouped (usually every 10,000 rows) to create entry points for fine-grained skipping.
- Postscript: A small section at the very end of the file that acts as the "key" to the whole file, telling the reader the compression type and footer length.

Unlike Parquet, which uses a more generalized approach, ORC is type-aware. This means it chooses the best compression strategy based on the specific data type of a column:

  - Integers: Uses variable-width encoding (smaller numbers take less space) and Run-Length Encoding (RLE) for repeated values.
  - Strings: Uses a Dictionary Encoding that is sorted. This not only compresses better but allows for faster filtering because the engine can search the dictionary instead of the raw text.

ORC provides three levels of indexing that allow it to skip data more aggressively than almost any other format:

1. File Level: Global statistics (min/max/null count) for the entire file.
2. Stripe Level: Statistics for each ~250MB stripe. If the query is looking for a date in 2024 and the stripe min/max is all 2023, the entire 250MB is skipped.
3. Row Group Level: Statistics for every 10,000 rows. This allows the reader to "jump" within a stripe, reading only the specific blocks needed.

The Result: On HDFS, this "index sequential" behavior means a query might only touch 1% of the actual data on disk, drastically reducing I/O and network congestion.

C) ORC vs. Parquet: The Essential Differences

Feature	Apache ORC	Apache Parquet
Best For	Apache Hive & transactional (ACID) data.	Apache Spark & broad tool compatibility.
Compression	Generally higher due to type-specific encoders.	High, but more generalized.

ACID Support	Built-in. Supports Update/Delete in Hive.	Not natively supported (requires Delta Lake/Iceberg).
Indexing	Extremely granular (down to every 10k rows).	Granular (at the Page level).

3.2. Hadoop Archive (HAR) Files

For the purpose to manage many small files sequentially, HAR files are the closest legacy equivalent. They create an index file that maps original file paths to their location within a larger master archive file, reducing the memory pressure on the NameNode. But for bigger HDFS file sizes, this approach is not too good.

3.3. HBase Real-Time Indexed Layer for accessing data

For the purpose of true indexed access with the ability to update records (which HDFS doesn't allow), Apache HBase is one of the possible good answer.

- It sits on top of HDFS.
- It stores data in SSTables (Sorted String Tables).
- Data is sorted by a Row Key, creating a distributed, indexed, and sequential storage layer.

The essence of Index Sequential organization (ISAM) is **predictability through structure.** Imagine a physical encyclopaedia. You have the main content (the data) stored in alphabetical order, and a thumb index (the index) that tells you exactly which page to flip to.

- How it works: Data is stored in sorted order based on a primary key. An index file stores the starting address of blocks of data.
- The Workflow: To find a record, the system searches the small index first to find the right "neighborhood," then jumps to that specific spot in the data file.
- The Catch: While great for reading, it struggles with inserts. If it is required to add a name starting with "B" but the "B" section is full, the system has to use "overflow areas," which eventually makes the index messy and slow.

HBase (built on top of Hadoop) is a distributed, versioned, column-oriented NoSQL database. Its essence is massive scalability. While ISAM is like a single well-organized book, HBase is like a warehouse spanning several city blocks where the shelves can grow indefinitely. The essence of this organization lies in the LSM-Tree (Log-Structured Merge-Tree) architecture and the Row Key design.

Traditional databases (like MySQL) often use B+ Trees, which require random disk writes. HBase avoids this "seek" penalty by treating data as a sequential stream.

- MemStore: All incoming writes are first sorted in memory. This ensures that even if data arrives out of order, it is organized into a sequence before hitting the disk.

- HFiles: Once the MemStore reaches a certain size, it is "flushed" to HDFS as an HFile. Because the data was already sorted in memory, the HFile is a perfectly indexed, sorted sequence of KeyValues.
- Compaction: Over time, HBase merges these smaller HFiles into larger ones, maintaining the global sort order and removing deleted items.

Key Characteristics:

- Sparse & Column-Oriented: Unlike a traditional table where every row must have the same columns, HBase allows rows to have completely different columns. If a cell has no data, it takes up zero space.
- The Row Key is King: Like ISAM, HBase stores data sorted by a Row Key. This is the only way to retrieve data quickly.
- LSM-Trees (The Secret Sauce): Unlike ISAM's static files, HBase uses Log-Structured Merge-Trees. It writes updates to memory first and then flushes them to disk as sorted files. This makes it incredibly fast for writing data, which is where ISAM usually fails.

In HBase, there is only one "real" index: the Row Key. Everything—Columns, Column Families, and Timestamps—is stored relative to this key.

- Lexicographical Sorting: HBase stores rows in alphabetical (lexicographical) order based on the Row Key.
- Search Efficiency: Because the data is stored in a sorted sequence, HBase can perform a Range Scan very efficiently. If you know the start and end of a sequence (e.g., all logs from `2026-01-01` to `2026-01-02`), HBase simply jumps to the start point and reads sequentially.

To handle massive scale, HBase breaks the "index sequence" into chunks called Regions.

While HDFS is an append-only file system, meaning files cannot be modified once written, HBase allows to "update" and "delete" records constantly. HBase pulls this off by never actually changing the original data. Instead, it uses a concept called LSM-Tree versioning and Tombstones.

When is required to update a cell in HBase, the system doesn't search for the old value to overwrite it. *The Process*: HBase simply writes a new version of that cell with a newer timestamp. *Storage*: Both the old version and the new version exist simultaneously in different HFiles on HDFS. *Retrieval*: When is required to perform a Get, HBase looks at all versions and, by default, returns the one with the highest (most recent) timestamp.

Since HBase cannot go back into a closed HFile to erase a row, it uses a Tombstone. The *Process*: When is issued a delete command, HBase writes a special record called a Delete Marker (Tombstone) into the MemStore. *The Mask*: This marker is sorted right along with the data. During a read operation, if HBase encounters a tombstone, it knows to "hide" any data with that key that has an older timestamp. *Physical Existence*: The "deleted" data is still physically taking up space on HDFS until a specific cleanup process occurs.

To prevent HDFS from becoming cluttered with old versions and tombstones, HBase performs Compaction. This is the only time data is actually removed.

For example, if there is 10 billion records (like Facebook likes or sensor data) in the file and is needed to find one specific interaction in milliseconds, HBase can make this job. It takes the "sorted key" concept of ISAM and stretches it across thousands of computers.

Apache HBase is actually very well-suited for index-sequential access, updates, and deletes, but for reasons that are quite different from traditional ISAM/VSAM or B-tree-based systems. HBase Works Well for Index-Sequential Access, Updates, and Deletes.

3.3.1. Index-Sequential Access (ISA)

HBase is fundamentally a sorted, lexicographically ordered key–value store. This makes it ideal for sequential scans over a key range.

What makes it suitable:

- Sorted RowKeys: All data is stored in sorted order by row key.
- Region Splits: Sequential ranges map to contiguous regions, enabling efficient range scans.
- HFiles: Immutable, sorted files on HDFS optimized for sequential reads.
- Bloom filters: Reduce unnecessary disk reads during scans.

Result: It is fast, predictable range scans, especially when is required row key design to be good (e.g., prefix-based, salted, or composite keys).

3.3.2. Updates

HBase is designed for high-throughput random writes, which is exactly what updates are.

Why updates are fast:

- LSM-tree architecture:
  - Writes go to MemStore (in-memory)
  - Then flushed to HFiles
  - No in-place updates → no random disk I/O
- MVCC (Multi-Version Concurrency Control):
  - Each update is a new version
  - Readers never block writers
- Append-only design:
  - Updates are just new cells with higher timestamps

Result: Updates are cheap, fast, and scalable, even at millions of ops/sec.

3.3.3. Deletes

Deletes in HBase are not immediate physical removals. They are tombstones, which fit perfectly with the LSM design.

How deletes work:

- A delete writes a tombstone marker

- During compaction, tombstones remove older versions
- No random disk writes → consistent write performance

Types of deletes:

- Delete specific cell
- Delete entire column family
- Delete entire row
- Delete up to a timestamp

Result: Deletes are as fast as writes, and compactions clean up storage later.

In the following table are compared the features of HBase for this file access.

Operation	Why HBase Is Suitable	Internal Mechanism
Index-Sequential Access	Fast range scans, sorted storage	Sorted row keys, HFiles, region splits
Updates	High-throughput random writes	LSM-tree, MVCC, append-only writes
Deletes	Cheap, non-blocking	Tombstones + compaction

3.3.4. When HBase Is *Not* Suitable

Just to keep the architecture honest:

- Not ideal for **complex secondary indexes** (unless using Phoenix or custom indexing)
- Not ideal for **ad-hoc SQL queries**
- Not ideal for **small datasets** (HBase shines at scale)

Suitability of HBase for different file operations and types in presented in the next table.

Operation / File type	Suitability	Notes
Index-Sequential Access	⋆ ⋆ ⋆ ⋆ ⋆	Perfect for chunk/time-based indexes
Updates	⋆ ⋆ ⋆ ⋆ ⋆	LSM-tree + MVCC = fast updates
Deletes	⋆ ⋆ ⋆ ⋆ ⋆	Tombstones + compaction
Storing raw audio/video	⋆ ⋆	Technically possible but strongly discouraged
Random access inside media	⋆	Use HDFS + byte-range reads instead

In summary, HBase is highly suitable for the *indexing layer* of audio/video systems, especially when is needed:

- Fast sequential scans
- High-throughput updates
- Cheap deletes

- Scalable metadata storage
- Time-based or chunk-based indexing

But it is not suitable for storing the media files themselves.

3.4. Hive Indexing / Partitioning

While Hive dropped support for "compact" and "aggregate" indexes in newer versions (favoring ORC/Parquet), Partitioning remains the primary way to organize HDFS data sequentially. By storing data in directories like /year=2024/month=02/, it is possible to create a coarse-grained index that the system uses to navigate directly to the relevant sequential files.

The data organization of Hive indexing is presented in figure 4.

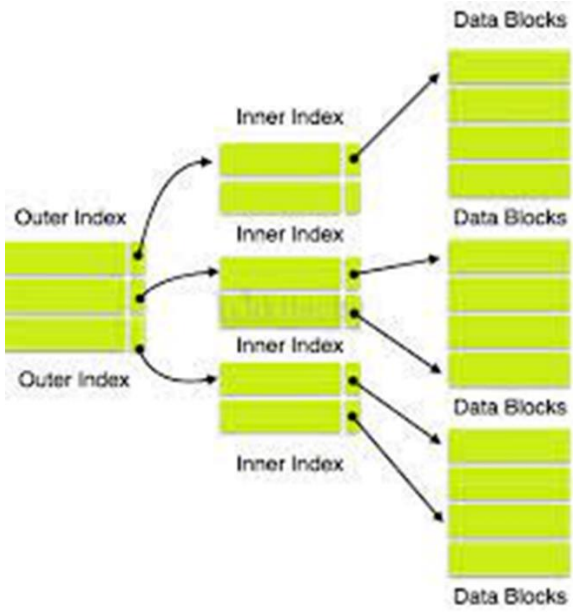

Figure 4

A summary of the presented comparative approaches is given in the next table

Feature	HDFS (Raw)	Parquet / ORC	HBase
Organization	Flat Blocks	Row Groups / Stripes	Sorted Key-Value
Search Method	Full Scan	Metadata Filtering (Pushdown)	Row Key Index
Write Type	Append-only	Write-once	Random Updates
Best Use Case	Raw Logs	Analytics / Data Warehousing	Real-time Lookups

4. Organization of HDFS file with direct access, update and delete

4.1. General approach

HDFS (Hadoop Distributed File System) was famously designed for a write-once, read-many model. By its nature, it is an immutable file system meant for high-throughput streaming rather than low-latency random access or in-place updates.

HDFS splits files into large Blocks (defaulting to 128MB or 256MB) and distributes them across a cluster.

- NameNode: The "brain" that stores metadata (where blocks are located).
- DataNode: The "muscle" that stores the actual data blocks.

Strictly speaking, HDFS does not support "direct access" in the way a Solid State Drive does. However, it provides a mechanism to seek specific data:

- The Mechanism: Using the FSDataInputStream, a client can call seek(long pos) to jump to a specific byte offset.
- The Catch: While you can jump to a position, performance is best when reading sequentially. Frequent random seeks are inefficient because HDFS must coordinate with the NameNode to find the correct block and then establish a connection with the specific DataNode holding it.

Update uses the "Append-Only" rule. HDFS does not support in-place updates. It is not possible to change a byte in the middle of a 500MB file.

- Why? Maintaining data consistency across replicated blocks during an in-place edit would be a networking nightmare and would tank performance.
- The Solution: HDFS allows Appends. It is possible to add new data to the end of an existing file. If it id required to "update" a record, the standard practice is to:
    1. Read the original file.
    2. Apply the change in memory or via a MapReduce/Spark job.
    3. Write a brand new file and delete the old one.

Deletions in HDFS are designed to be safe and efficient:

- Soft Delete (Trash): By default, when you delete a file, it isn't immediately erased. It is moved to a /.Reserved//.Trash directory.
- NameNode Processing: The NameNode removes the file from its namespace immediately. The physical blocks on the DataNodes are only deleted later during a garbage collection process once the "Trash" interval expires.
- Efficiency: Deleting a file is very fast because it only involves updating metadata on the NameNode; the DataNodes eventually clear the space in the background.

Deletion of a record (set of bytes) from HDFS file is similar to update principle explained above.

Feature	Support Level	Implementation Method
Direct Access to data in a file	Partial	Via seek() to a byte offset.
Update data inside a file	Limited	Append-only; no in-place modification.
Delete data from inside a file	Limited	Append-only; no in-place modification.
Delete a file	Full – all data from a file	Metadata removal followed by background block cleanup.

4.2. Three-layered architectural solution

To make a solution for HDFS files direct access update and delete, a software solution can be applied, using 3 layers and implementing additionally software as the components HBase, Hive and Aceberg plus additional software code. The three layers are:

1. Storage Layer (HDFS)

Stores immutable blocks. No updates here.

2. Index + Metadata Layer

This is where the magic happens. It has to maintain:

- Primary index → maps logical IDs to physical HDFS block offsets
- Delta logs → track updates and deletes
- Compaction service → periodically merges deltas into base files

Common implementations:

- HBase (random access + updates)
- Hive ACID (delta files + compaction)
- Iceberg (modern table formats with metadata trees)

3. Access Layer

Provides APIs for:

- Direct record lookup
- Update (insert new version into delta)
- Delete (mark tombstone)
- Scan with predicate pushdown

This architecture can work in 2 options:

Option A: using HBase (true random access)

HBase stores data in HFiles on HDFS but exposes:

- Get(rowkey) → O(log n)
- Put(rowkey) → update
- Delete(rowkey) → tombstone

This is the closest to “direct access” that can get on HDFS.

Option B: using Iceberg

These systems maintain:

- Manifest files
- Column-level statistics
- File-level indexes
- Bloom filters

Direct access is achieved by skipping irrelevant files and reading only the necessary blocks.

In the following table are compared 3 additional software components working on top of HDFS files – HBase, Hive and Iceberg in comparison with pure HDFS files, from which the predominant value of HBase for direct and sequential access of data in HDFS file is focused.

Feature	HDFS Alone	HBase	Hive ACID	Iceberg
Direct access	✗	✓	⚠ (slow)	⚠(fast with indexes)
Update	✗	✓	✓	✓
Delete	✗	✓	✓	✓
Random reads	✗	Excellent	Medium	Good
Compaction	N/A	Automatic	Required	Required
Best for	Logs, big files	OLTP-like access	Batch tables	Lakehouse, mixed workloads

Here is good to make the essence and difference of HBase and Hive for the purpose of direct access of data in HDFS files.

Apache Hive: Mainly used for SQL-on-Hadoop Warehouse. Hive is a data warehouse software built on top of Hadoop for providing data query and analysis. It allows you to write HQL (Hive Query Language), which is very similar to SQL, and translates those queries into MapReduce or Spark jobs. Spwcially used mainly for Analytical processing (OLAP), complex joins, and generating reports on massive historical datasets.

Apache HBase: Mainly used for NoSQL Database. HBase is a distributed, scalable, Big Data store. It is a NoSQL Key-Value store that runs on top of the Hadoop Distributed File System (HDFS). Unlike Hive, it doesn't use SQL; it uses Java APIs or Thrift/REST for data access. Specially used for: Real-time, random read/write access (OLTP) and handling "sparse" data (tables with many columns but few values).

The major differences between HBase and Hive are presented in a table below:

Feature	Apache Hive	Apache HBase
Type	Data Warehouse / SQL Engine	NoSQL Key-Value Store
Primary Use	Batch Analytics (OLAP)	Real-time Operations (OLTP)
Latency	High (Seconds to Minutes)	Low (Milliseconds)
Data Format	Structured (Schema-on-write)	Semi-structured (Schema-less)
Storage	HDFS (Files)	HDFS (Key-Value pairs)
Transactions	Limited / Batch-oriented	Row-level atomicity

In Hive, if it is required to find one specific customer in a 10-petabyte table, Hive has to scan through massive chunks of data (or partitions) to find them. This is like searching for a specific quote by reading an entire library of books. It's slow for one record, but great for counting how many times "Love" appears in every book. In HBase, if it is required to provide a "Row Key" (like a library index card), and it jumps directly to that specific record in milliseconds. It doesn't care if there are trillions of rows; it knows exactly where that needle is in the haystack.

4.3. Best practices recommended architecture, using additional supporting software components

In many enterprise architectures for Big Data Systems with tens/thousands DataNodes, HBase and Hive are complementary.

The first recommended architecture uses HBase to capture real-time streaming data from an app, and then use Hive to run a "heavy" end-of-the-month analytical report on that same data.

The second recommended architecture consist of HDFS file + Iceberg + Optional HBase Index. The reason for this is:

- Iceberg gives ACID, schema evolution, delete/update, manifest-based indexing, snapshot isolation.
- HBase can serve as a secondary index for ultra-fast point lookups.
- Giving HDFS as the cheap, scalable storage layer.

The term ACID means: **A**tomicity (A transaction is all or nothing - if an update touches 10 files, either all 10 succeed or none, preventing partial writes and corrupted states); **C**onsistency (The system moves from one valid state to another valid state, where schema rules are preserved, constraints are respected and metadata stays coherent); **I**solation (Concurrent transactions do not interfere with each other, where Readers see a consistent snapshot, Writers don't block readers and Writers don't corrupt each other's work); and **D**urability (Once a transaction is committed, it is permanently stored, even if A node fails, A service restarts or A network partition occurs).

The third recommended architecture is the simplest possible architecture using:

- Store base files in HDFS (Parquet/ORC),

- Maintain a small key→offset index in HBase or RocksDB,
- Store updates/deletes in delta files.
- Run compaction periodically.

This architecture gives: Direct access, Update, Delete, without adopting a full lakehouse framework.

5. MetaHDFS file system for dynamic update and delete operations in Multi-DataNode Big Data system

A new approach is developed to have Dynamic Update and Delete operations over HDFS files, where the principles are presented here, while the implementation can vary depending of used number of DataNodes in Big Data System (Hadoop) and supporting software components available for Big Data Systems, which leads to different level of parallel high performance computing, different execution time and different possible supporting software components. About the additional supporting software components – they can vary from pure codding like in Python, till using Apache HBase and/or Apache Hive.

Generally, the HDFS files are not designed for dynamic update and delete operations – they are designed for append functioning, which means read only with possibility to add some records at the end of the file. In this general philosophy, any change of the file content somewhere in the middle of the file, requires total rewriting of the file as a new file and after that deleting of the old version of the file.

The current offered approach for Dynamic Update and Delete of HDFS file called "MetaHDFS file system for dynamic update and delete operations ", is based on 2 file components – First component: the file – Main file with all its records and the Second component: Meta file serving as entry for the Main file, managing where to read and what to write at the end of the Main file in the operations update and delete. This structure of this MetaHDFS file is presented in the next figure 5.

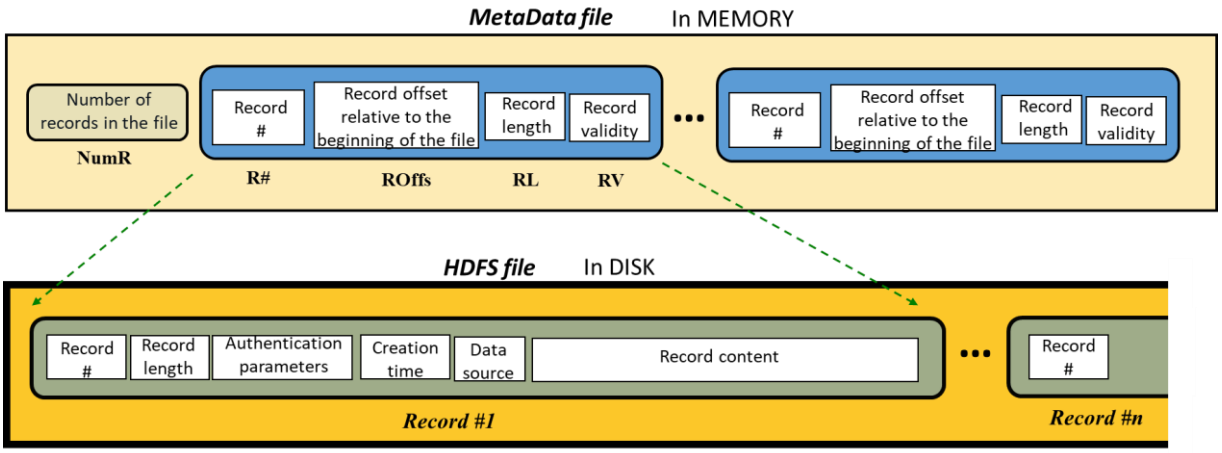

Fig. 5

The MetaHDFS file consist of HDFS file (the Main file - standard HDFS file for Big Data Systems – Hadoop) and MetaData file (Meta file). There is a way of implementation in which the Main file - HDFS file is stored on disks in different DataNodes of Hadoop system – as low level HDFS file, or is stored as higher level organization on top HDFS file as Hive, HBase or Spark type of files. The MetaFile is stored in the Memory during the MetaHDFS file is in operational mode - reading, writing, updating, deleting. It is stored on disk, when the MetaHDFS file is in non-operational mode. To keep the high-reliability of the MetaHDFS file, equal to all files in Big Data Systems-Hadoop, which means to have many copies of the same file, HBase or Hive can be used to store that file. When the MetaHDFS file is moving to operational mode, the MetaDatafile is reading from its storage environment (for example from Have or HBase), it is loaded in memory for quick operation of update and delete and this action is equivalent to the process of OPEN the MetaHDFS file. After finishing the operation, the records - data-content of the Main file are staying in HDFS file, and the MetaData file is also store from the memory to disk environment (as Have file, or HBase file, or as traditional HDFS file).

In HDFS file, each record consists of the following components: Record #(number), Record length, Authentication parameters, Creation time, Data source (in IP address or computer-name) and the Record content. Each record in the MetaHDFS file is numbered from 1 to n. The Athentication parameters are used for high-level authentication between users for data exchange between users and for type of possible operation of the counterpart users – only for reading, writing, updating, deleting and combination of them. The Record content consists of binary data, being structured or non-structured data, as well as audio and video.

The MetaData file consists of: first record containing the maximum number of records in the MetaHDFS file and after that Meta-records – one per record in HDFS file. The Meta-record contains: Record # (for which record in the MetaHDFS file is related this Meta-record), Record Offset from the beginning of the HDFS file (the position in bytes of the specific record in HDFS file according to the beginning of the HDFS file), Record length and Record validity-RV (with value “YES” for normal use of the specified record in the HDFS file; with value “Upd” for record, which is already updated and the new updated content is stored somewhere at the end of the HDFS file as new content with the same record number and with Meta-record with the same record number stored in MetaData file with RV (validity)=YES; with value “Del” for record which is logically deleted from the MetaHDFS file, it is accepted as already deleted and there is no Meta-record in the MetaData file with number equal to the specified number).

During the reading process, the Meta-record for each record in HDFS file will show how many bytes from the beginning of the HDFS file is the beginning of the giving record (from “Record Offset related to the beginning of the file”) and fast reading in HDFS file will ensure quickly to move to the beginning of the record from where to start reading. The “Record length” in Meta-record will specify how many bytes is the record length and this will lead to the reading of that exact number of bytes equal to the exact record content. This reading process is valid for records, which Meta-record is with RV (validity)=YES.

The process of Dynamic Update in MetaHDFS file is presented in figure 6.

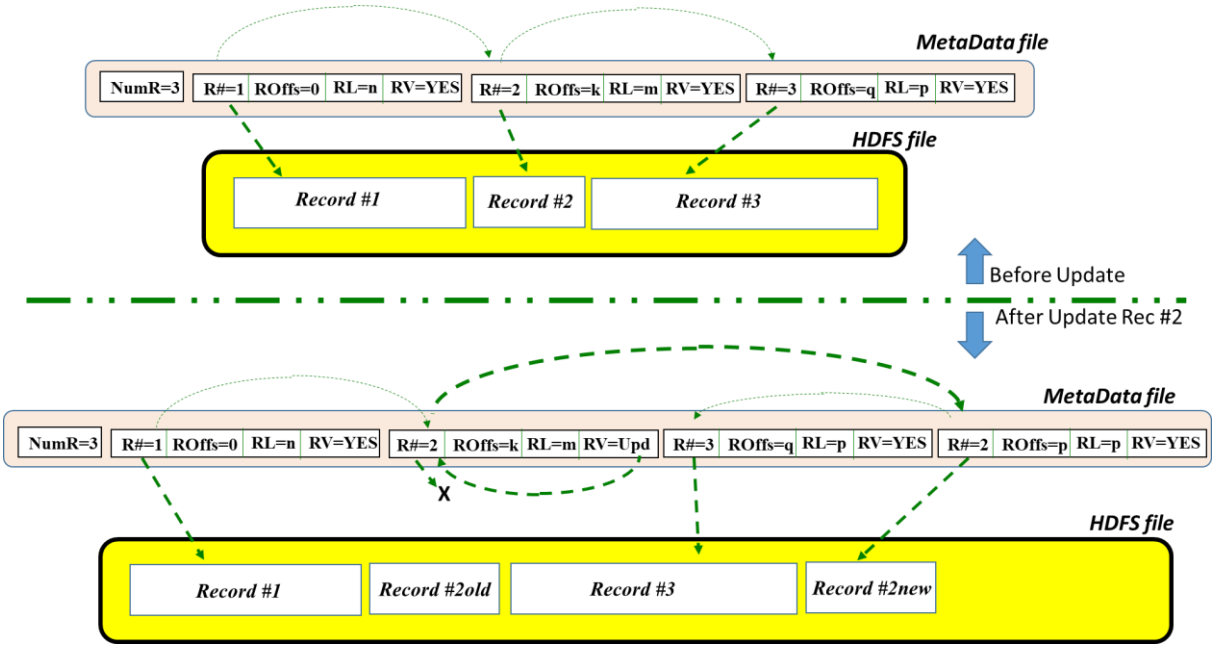

Fig. 6

In that figure above, before the Update, a MetaHDFS file is presented with MetaData file and HDFS file. After the Update of the second record, a new second record is written at the end of the HDFS file, the Meta-record for the second record is modified with RV (validity)=Upd and a new Meta-record for second record is created at the end of MetaData file with R#=2 and RV (validity)=YES. In this way, during the reading process, the second Meta-record will tell that a new Meta-record that has to be find another Meta-record with R#=2, which will lead to the new second record.

The process of Dynamic Delete in MetaHDFS file is presented in figure 7.

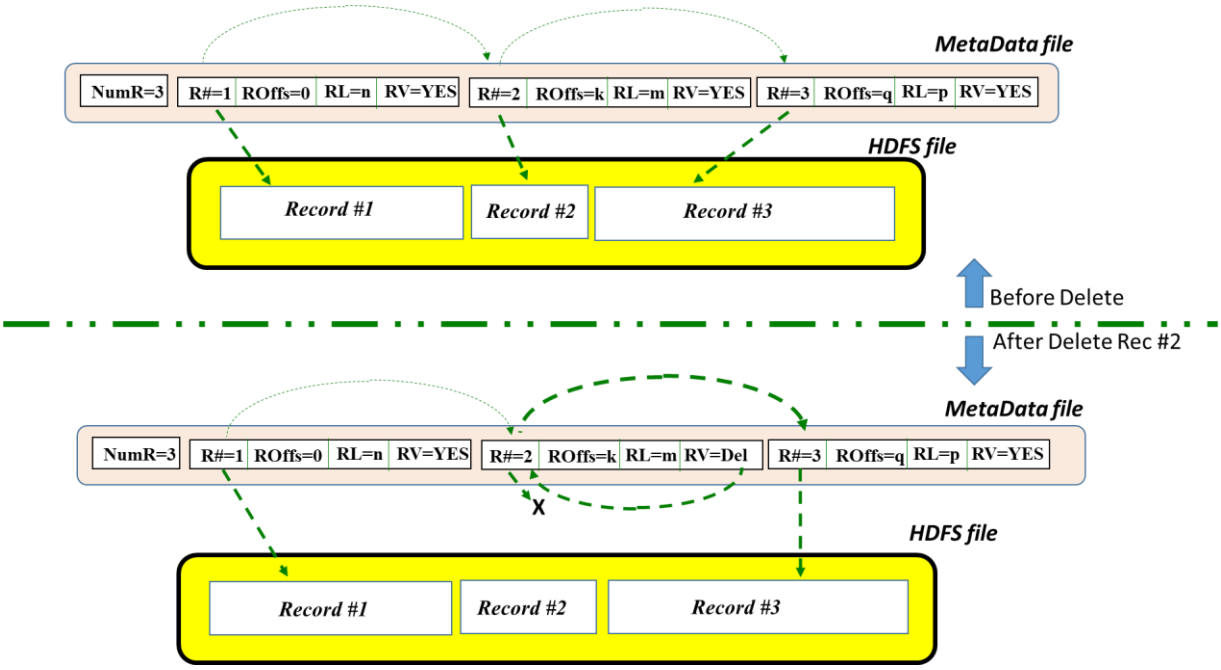

Fig. 7

In this figure, the second record in HDFS file is deleted, where the Meta-record for the second record is changed with RV (validity)=Del, which means, during the reading process, the Meta-record for second record will define that the second record is logically deleted and will be not read from the HDFS file, independently that the content is there.

References:

1. Hadoop Distributed File System (HDFS), https://www.databricks.com/glossary/hadoop-distributed-file-system-hdfs
2. What is Hadoop Distributed File System (HDFS)?, https://www.ibm.com/think/topics/hdfs
3. How to access files in Hadoop HDFS?, https://stackoverflow.com/questions/36290755/how-to-access-files-in-hadoop-hdfs
4. HDFS Architecture Guide, https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
5. Guide to Hadoop HDFS data access, https://cerexio.com/blog/guide-to-hadoop-hdfs-data-access
6. Accessing Data from HDFS, https://docs.cloudera.com/cdsw/1.10.5/import-data/topics/cdsw-accessing-data-from-hdfs.html
7. Apache HBase – Hadoop database, https://hbase.apache.org/
8. What is HBase?, https://www.ibm.com/think/topics/hbase
9. Apache Hive - Distributed Data Warehouse at Massive Scale, https://hive.apache.org/
10. What is Apache Hive?, https://www.databricks.com/glossary/apache-hive
11. Indexing a HDFS sequence file, https://stackoverflow.com/questions/6537435/indexing-a-hdfs-sequence-file
12. What is the best way to index and search files in HDFS?, https://www.quora.com/What-is-the-best-way-to-index-and-search-files-in-HDFS
13. What are good ways to search files or folders on HDFS?, https://www.quora.com/What-are-good-ways-to-search-files-or-folders-on-HDFS