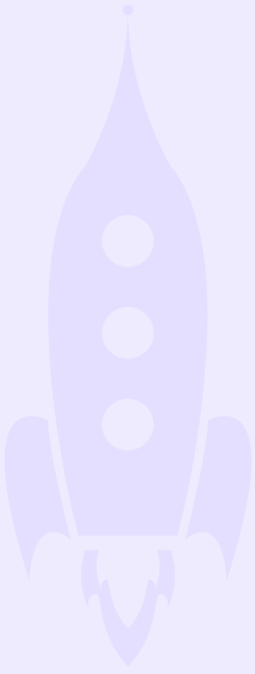
ONEHOUSE

Apache hudi

FROM ZERO TO ONE



Shiyan Xu
Apache Hudi PMC member

Contents

Preface

Apache Hudi is a revolutionary open-source framework that transforms the way data engineers and scientists interact with large-scale datasets. Hudi offers the capabilities of a traditional database, including efficient upserts, deletions, and incremental data processing, all by creating and managing metadata alongside a traditional data lake architecture. The combination of new data management capabilities, on top of data lake underpinnings, is referred to as a data lakehouse.

Hudi was the first data lakehouse project - though when it was introduced, in 2016, it was referred to as a transactional data lake. Since then, Hudi has been joined by Apache Iceberg and Delta Lake. All three are widely used open source data lakehouse projects.

Hudi's ability to perform efficient upserts, handle incremental updates smoothly, and its rich services layer overall, continue to serve as distinguishing features for Hudi. (See our comparison blog post for details.) And this year has seen the introduction of Apache XTable (Incubating), which provides read-write interoperability across the three projects.

This ebook, which originated as a series of Substack blog posts, provides a comprehensive introduction to Hudi's storage format, emphasizing its transactional nature and how it facilitates ACID properties on large-scale datasets. Through explorations of read and write operations, indexing strategies, and table services such as compaction and clustering, the book provides a foundation for understanding Apache Hudi's core mechanics and its impact on the design and efficiency of data lakehouses.

Chapter 1:
A First Glance at Hudi's Storage Format

Hudi Overview

Hudi is a transactional data lake platform that brings database- and data warehouse-like capabilities to the data lake. At its core, Hudi defines a data table format that organizes the data and metadata files within storage systems, allowing for ACID transactions, efficient indexing, and incremental processing. The Hudi stack is depicted in Figure 1-1.

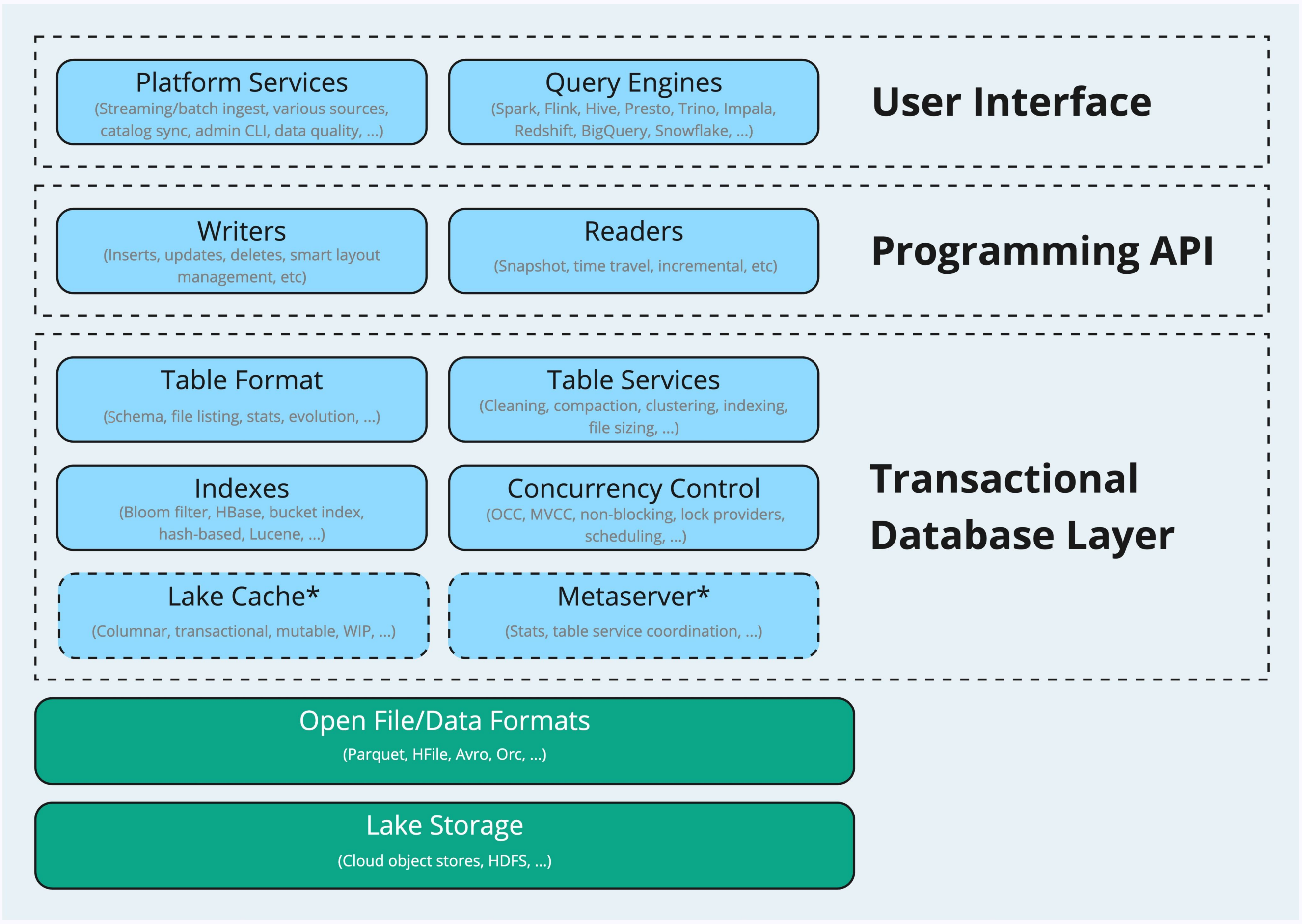

Platform Services
(Streaming/batch ingest, various sources, catalog sync, admin CLI, data quality, ...)
Query Engines
(Spark, Flink, Hive, Presto, Trino, Impala, Redshift, BigQuery, Snowflake, ...)
User Interface
Writers
(Inserts, updates, deletes, smart layout management, etc)
Readers
(Snapshot, time travel, incremental, etc)
Programming API
Table Format
(Schema, file listing, stats, evolution, ...)
Table Services
(Cleaning, compaction, clustering, indexing, file sizing, ...)
Indexes
(Bloom filter, HBase, bucket index, hash-based, Lucene, ...)
Concurrency Control
(OCC, MVCC, non-blocking, lock providers, scheduling, ...)
Lake Cache*
(Columnar, transactional, mutable, WIP, ...)
Metaserver*
(Stats, table service coordination, ...)
Transactional Database Layer
Open File/Data Formats
(Parquet, HFile, Avro, Orc, ...)
Lake Storage
(Cloud object stores, HDFS, ...)

Figure 1-1: The Hudi stack

The remainder of this chapter will explore the format details, showcasing the structure of a Hudi table on storage and explaining the roles of different files.

Storage Format

Figure 1-2 depicts a typical data layout of a Hudi table under the table's base path in storage.

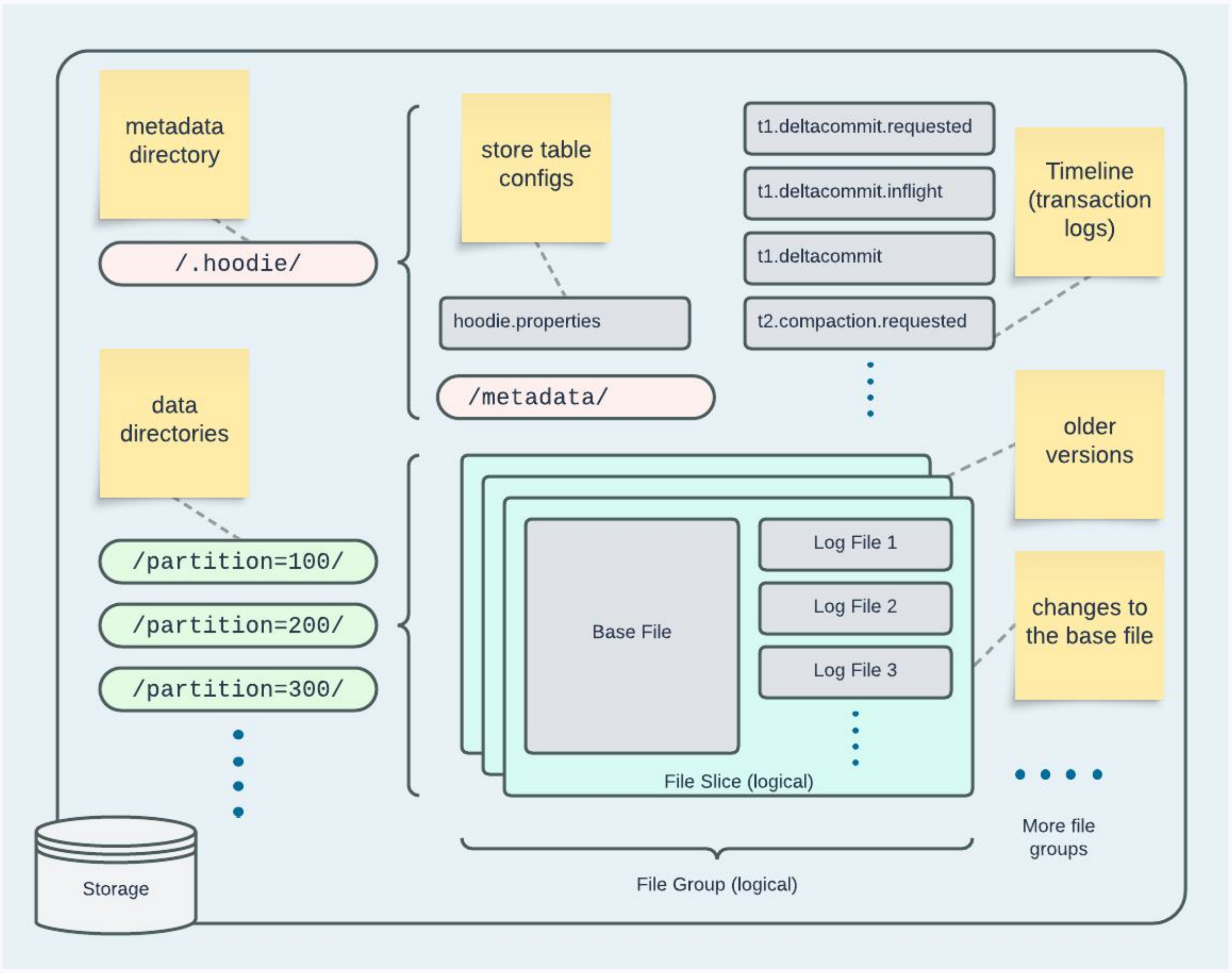


Figure 1-2: A typical data layout of a Hudi table

There are two main types of files:

- Metadata files, located in **<base_path>/.hoodie/**
- Data files, which are stored within partition paths for partitioned tables, or under the base path for non-partitioned tables

Timeline

A Hudi timeline is an important type of metadata which records transactional actions for a Hudi table. The timeline metadata files contain information about the changes that should be applied to the table and those that have already been applied.

Keeping these transaction logs makes it possible to recreate the table's state at any point, achieve snapshot isolation, and reconcile writer conflicts through concurrency control mechanisms.

The timeline metadata files follow this naming pattern:

```
<action timestamp>.<action type>[.<action state>]
```

Each field in the file-naming pattern is described as follows:

- The **<action timestamp>** marks when an action was first scheduled to run and uniquely identifies an action in the timeline. It increases monotonically across different actions in a timeline.
- The **<action type>** shows what kind of changes were made by the action. The following are common action types (additional action types will be discussed in more detail in future chapters):
  - Write actions, such as **.commit** and **.deltacommit**, indicate new write operations (**INSERT**, **UPDATE**, or **DELETE**) that occurred on the table
  - Table service actions, such as compaction and clean
  - Recovery actions, such as savepoint and restore
- The **<action state>** indicates the status of the action, which can be **requested**, **inflight**, or **completed** (without a suffix). As the names suggest, **requested** indicates being scheduled to run, **inflight** means execution is in progress, and **completed** means that the action is done.

Here's an example logging **.deltacommit** actions in timeline metadata files:

```
20230827233828740.deltacommit.requested
20230827233828740.deltacommit.inflight
20230827233828740.deltacommit
```

Other Metadata Files

Alongside the table's timeline metadata files, there are additional files and directories under the **.hoodie**/ directory. Examples include, but are not limited to:

- The **hoodie.properties** file, which contains essential table configurations, such as table name and version, which are used by both writers and readers of the table
- The **metadata**/ directory, which contains additional metadata related to timeline actions, and serves as an index for readers and writers
- The **.heartbeat**/ directory, which stores files for heartbeat management
- The **.aux**/ directory, reserved for various auxiliary purposes such as storing checkpoint metadata

Data

Hudi categorizes physical data files into two categories, base files and log files, which are optimized for write vs. read:

- Base files contain the main records in a Hudi table, are optimized for reads, and are typically formatted as columnar files (e.g., Apache Parquet).
- Log files contain record changes for an associated base file, are optimized for writes, and are typically formatted as row-based files (e.g., Apache Avro).

Within a partition path of a Hudi table (as shown in the previous layout diagram), a single base file and any associated log files are grouped together as a file slice. Multiple file slices constitute a file group. File slice and file group are logical concepts designed to enclose physical files, simplifying access and manipulation for both writers and readers.

Each file slice is tied to a specific timeline actions timestamp, and the file slices within a file group track how the contained records evolved over time, allowing Hudi to support versioning across commit actions.

Table Types

Hudi defines two table types, copy-on-write (CoW) and merge-on-read (MoR):

- CoW tables store data in columnar format, and each update creates a new file version during a write. CoW is the default storage type.
- MoR tables store data using a combination of columnar and row-based formats. Updates are logged to row-based delta files and are compacted to create new versions of the columnar files. Throughout this chapter, we have been using MoR as the example.

CoW can be treated as a special case of MoR, where records in a base file and associated changes are implicitly merged into a new base file during each write operation. Unlike MoR, CoW has no log file, and write operations result in **.commit** actions instead of **.deltacommit**.

When selecting a table type, consider your read and write patterns:

- CoW is well suited for read-heavy, analytical workloads or small tables. It has high write amplification due to rewriting records in the new file slices for every write, while read operations will always be optimized.
- MoR is best for larger tables with relatively frequent upserts. MoR has low write amplification because changes are buffered in log files and batch-processed to merge and create new file slices. However, read latency is increased, because inflight merging of log files with the base file is required for reading the latest records.

Users may also opt to only read base files of an MoR table to obtain efficiency while sacrificing result freshness. We'll discuss Hudi's different read modes in subsequent chapters. As the Hudi project evolves, the merging costs associated with reading from MoR tables have been optimized across releases. It is foreseeable that MoR will become the preferred table type for most workload scenarios.

Recap

In this chapter, we introduced Hudi as a comprehensive lakehouse platform offering features across various dimensions. We explored the fundamentals of Hudi's storage format to show how data is structured within Hudi tables. We also briefly explained the different table types and their trade-offs.

Chapter 2:
Read Operation Flow and Query Types

We'll now explore how read operations work in Hudi.

There are several analytical query engines integrated with Hudi, such as Spark, Presto, and Trino. Although the integration APIs may differ, the fundamental process in distributed query engines remains consistent. In general, the process is:

1. Interpret the input SQL
2. Create a query plan for execution on worker nodes
3. Collect the results to return to users

We're going to use Spark as the example engine to illustrate the flow of read operations and provide code snippets to showcase the usage of various Hudi query types.

In the next section, we'll introduce Spark queries with a primer, delve into Hudi-Spark integration points, and explain the different query types.

Spark Query Primer

Spark SQL is a distributed SQL engine that performs analytical tasks for large-scale data. A typical analytics query begins with user-provided SQL, aiming to retrieve results from a table on storage. Spark SQL then takes this input and proceeds through several steps, as depicted in Figure 2-1.

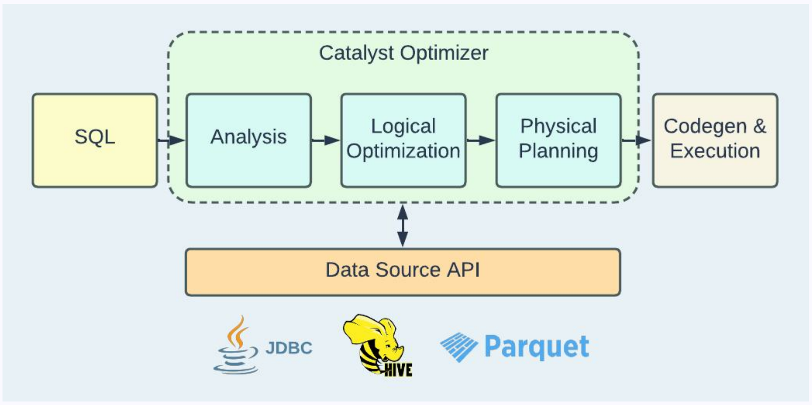


Figure 2-1: Spark SQL query planning flow

There are three distinct steps:

1. During the analysis step, the input is parsed, resolved, and converted into a tree structure that works as an abstraction of the SQL statement. The table catalog is consulted for information, such as table names and column types.

2. In the logical optimization step, the tree is evaluated and optimized at the logical layer. Some common optimizations include predicate pushdown, schema pruning, and null propagation. This step generates a logical plan that outlines the necessary computations for the query, but which lacks the specifics needed for running on actual nodes.
3. Physical planning serves as the bridge between the logical layer and the physical layer. A physical plan specifies the precise manner in which computations will be executed. For instance, in a logical plan, there may be a join node indicating a join operation, whereas in the physical plan, the join operation could be specified as a sort-merge join or a broadcast-hash join, depending on size estimates from the relevant tables. The highest-rated physical plan is selected for code generation and execution.

The three phases are features provided by the *Catalyst optimizer*. To learn more about the Catalyst optimizer, see the videos, A Deep Dive into Spark SQL's Catalyst Optimizer with Yin Huai and A Deep Dive into Query Execution Engine of Spark SQL - Maryann Xue.

During execution, a Spark application operates on a foundational data structure called a *resilient distributed dataset (RDD)*. RDDs are collections of Java virtual machine (JVM) objects that are immutable, partitioned across nodes, and fault-tolerant, due to the tracking of data lineage information. As the application runs, the planned computations are performed, and RDDs are transformed and acted upon to produce results. This process is commonly referred to as *materializing* the RDDs.

Data Source API

While the Catalyst optimizer handles the formulation of query plans, the *Spark Data Source API* connects to the data source, enabling optimizations to be pushed down. The API is designed to integrate with a wide range of data sources. Some data sources, such as JDBC, Hive tables, and Parquet files, are supported out-of-the-box. Hudi tables, due to their specific data layout, qualify as a custom data source.

Spark-Hudi Read Flow

In this section, we'll describe the Spake-Hudi read flow. Figure 2-2 illustrates key interfaces and method calls in the Spark-Hudi read flow common to all Hudi query types with Spark:

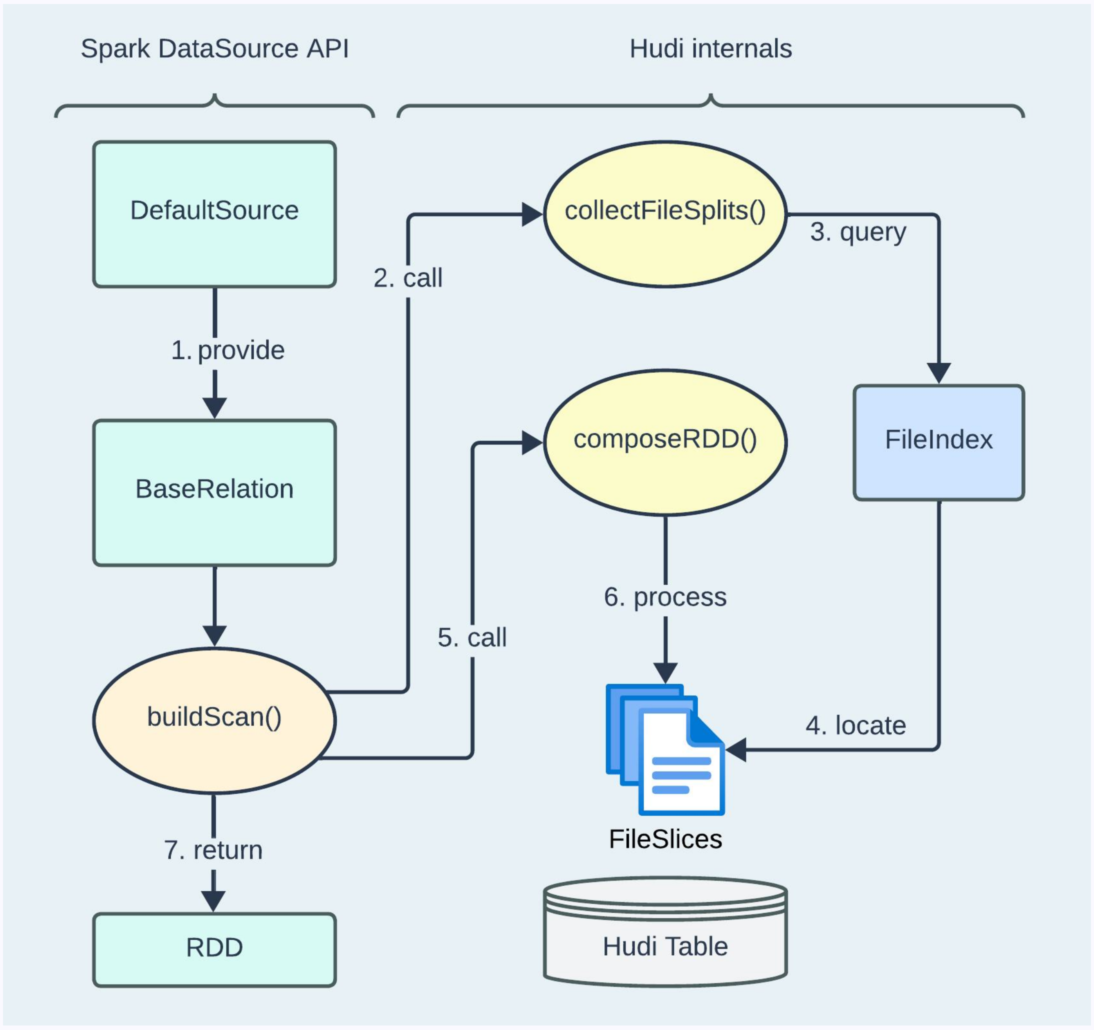


Figure 2-2: The Spark-Hudi read operations data flow

Here's a high-level overview of the process:

1. **DefaultSource** serves as the entry point of the integration, defining the data source's format as **org.apache.hudi** or **hudi**. It provides a **BaseRelation**, which Hudi uses to implement the data extraction process.
2. **buildScan()** is a core API to pass filters to data sources for optimizations. Hudi defines **collectFileSplits()** for gathering relevant files.
3. **collectFileSplits()** passes all the filters to a FileIndex object that helps identify the necessary files to read.
4. **FileIndex** locates all the relevant file slices for further processing.
5. **composeRDD()** is invoked after file slices are identified.
6. File slices are loaded and read out as RDDs. For columnar files, such as base files in Parquet, this read operation minimizes the transferred bytes by reading only the necessary columns.
7. RDDs are returned from the API for further planning and code generation.

Please note that this is a high-level overview of the read flow, omitting details such as support for schema-on-read and advanced indexing techniques, such as data skipping using a metadata table.

In the following sections, we'll discuss how various query types work. All types, except for read-optimized queries, are applicable to both CoW and MoR tables.

Snapshot Query

A *snapshot* query is the default query type when reading Hudi tables. It retrieves the latest records from the table, capturing a snapshot of the table at the time of the query. When a snapshot query is performed on an MoR table, log files must be merged with the base file to complete the query, which causes some degree of impact on performance.

The following example SQL query shows how to set up an MoR table, with one record inserted and updated after launching a spark-sql shell with Hudi as a dependency:

```
create table hudi_mor_example (
  id int,
  name string,
  price double,
  ts bigint
) using hudi
tblproperties (
  type = 'mor',
  primaryKey = 'id',
  preCombineField = 'ts'
) location '/tmp/hudi_mor_example';

set hoodie.spark.sql.insert.into.operation=UPSERT;
insert into hudi_mor_example select 1, 'foo', 10, 1000;
insert into hudi_mor_example select 1, 'foo', 20, 2000;
insert into hudi_mor_example select 1, 'foo', 30, 3000;
```

The next example shows a snapshot query that will retrieve the latest value of the record by running a **SELECT** statement:

```
spark-sql> select id, name, price, ts from hudi_mor_example;
1      foo     30.0    3000
Time taken: 0.161 seconds, Fetched 1 row(s)
```

Read-Optimized Query

Read-optimized queries (RO) are designed specifically for use with MoR tables, as they have lower read latency, but with potentially older results. When conducting queries such as **collectFileSplits()**, an RO query will fetch only base files for file slices.

In the previous snapshot query code example, the code automatically generates a catalog table named **hudi_mor_example_ro**, which specifies a property **hoodie.query.as.ro.table=true**. This property instructs query engines to always perform RO queries.

Running the following **SELECT** statement returns the original value of the record, because subsequent updates have not yet been applied to the base file.

```
spark-sql> select id, name, price, ts from hudi_mor_example_ro;
1      foo     10.0    1000
Time taken: 0.114 seconds, Fetched 1 row(s)
```

Time Travel Query

A *time travel query* allows users to request a historical snapshot of a Hudi table by providing the appropriate timestamp. As previously discussed in Chapter 1, file slices are associated with specific commit times and, therefore, support filtering. In a time travel query, the file index only locates the file slices that correspond to the specified time. If there is no exact match, the query returns the closest matches that are older than the specified time.

Here's an example of a time travel query:

```
spark-sql> select id, name, price, ts from hudi_mor_example timestamp as
of '20230905221619987';
1      foo     30.0    3000
Time taken: 0.274 seconds, Fetched 1 row(s)

spark-sql> select id, name, price, ts from hudi_mor_example timestamp as
of '20230905221619986';
1      foo     20.0    2000
Time taken: 0.241 seconds, Fetched 1 row(s)
```

In the first example, the **SELECT** statement executes a time travel query at the **.deltacommit** time of the latest insert, providing the most recent snapshot of the table. In the second example, the query sets a timestamp earlier than that of the latest insert, resulting in a snapshot as of the second-to-last insert.

The timestamp in each example follows the Hudi timeline's format of **yyyyMMddHHmmssSSS**. However, the timestamp can also be formatted as **yyyy-MM-dd HH:mm:ss.SSS** or **yyyy-MM-dd**.

Incremental Query

With an *incremental query*, users can retrieve changed records within a specified time window. The time window is specified with a starting timestamp and an optional ending timestamp. (If no ending timestamp is set, the window returned will include all records since the starting timestamp.) Hudi also offers full change data capture (CDC) capabilities by enabling additional logs on the writer's side and activating CDC mode for incremental reads. See Chapter 8 for more information.

Recap

In this chapter, we provided an overview of Spark's Catalyst optimizer, explored how Hudi implements the Spark Data Source API for reading data, and introduced four distinct Hudi query types.

Chapter 3: Understanding Write Flows and Operations

We'll now delve into write flows, with Spark as the example engine. When it comes to writing data to Hudi, there are numerous customizable configurations and settings. Therefore, this chapter does not aim to serve as a complete usage guide.

Instead, the primary goal is to describe internal data flows and break down the steps involved during a write operation. This will provide readers with a deeper understanding of running and fine-tuning Hudi applications. For practical usage examples, see Hudi's docs.

Write Operation Flow

The following diagram illustrates the typical high-level steps involved in a Hudi write operation within the context of an execution engine. A Hudi write operation involves several high-level steps, as illustrated in Figure 3-1.

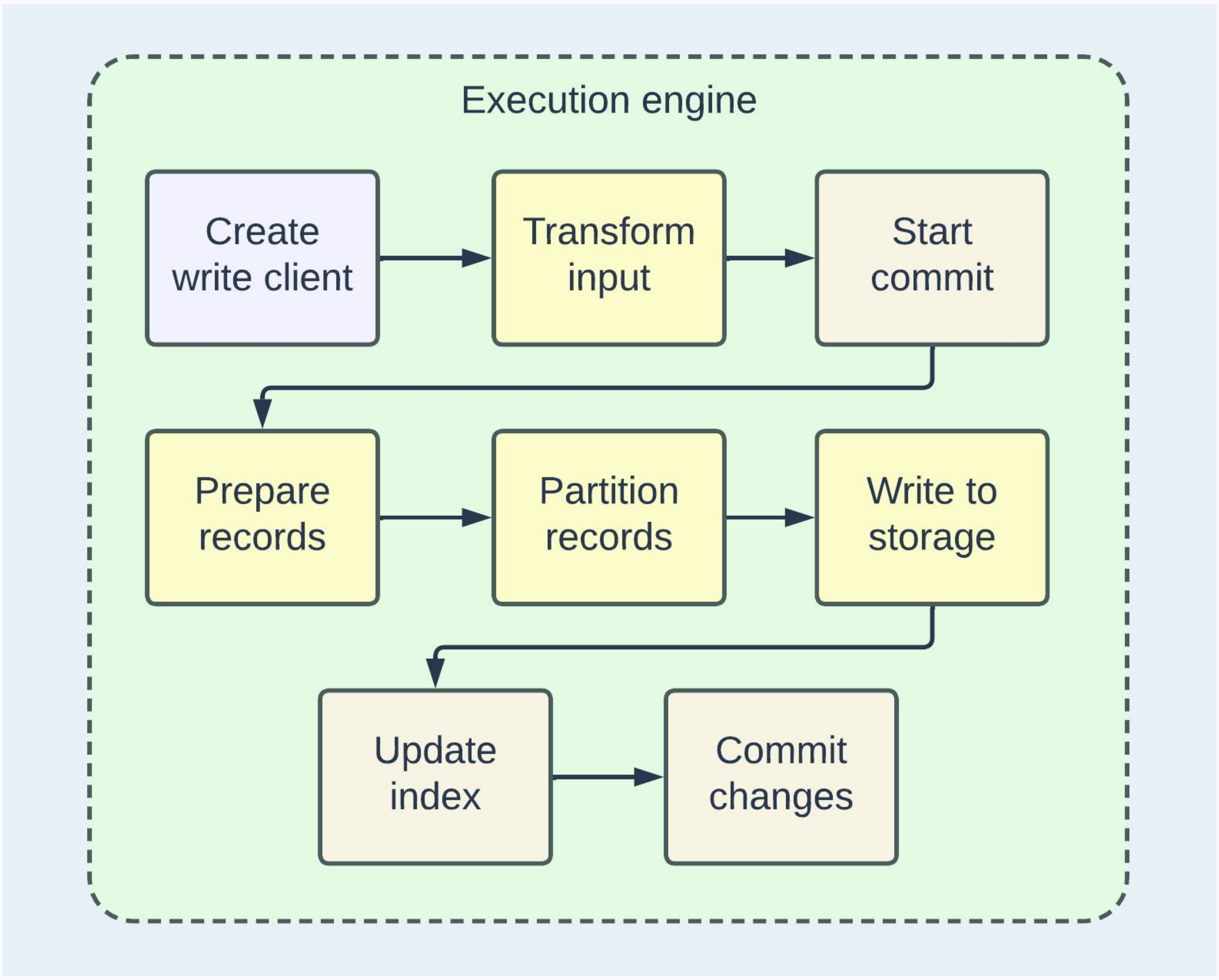


Figure 3-1: Hudi write operation data flow

1. Create write client

A Hudi write client serves as the entry point for write operations, and Hudi write support is achieved by creating an engine-compatible write client instance. For example, Spark uses the **SparkRDDWriteClient**, Flink employs the **HoodieFlinkWriteClient**, and Kafka Connect generates the **HoodieJavaWriteClient**. Typically, this step involves reconciling user-provided configurations with the existing Hudi table properties, then passing the final configuration set to the client.

2. Transform input

Before a write client processes the input data, several transformations occur, including the construction of a **HoodieRecord** (see Figure 3-2), which is a fundamental model in write paths, and schema reconciliation.

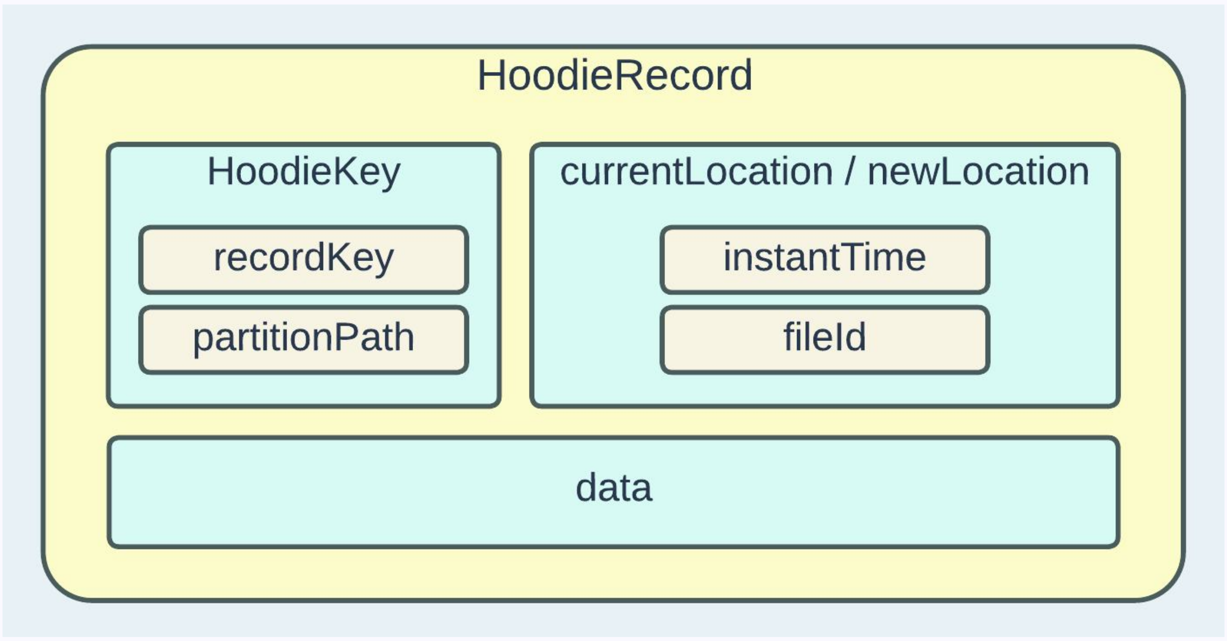


Figure 3-2: The structure of a Hudi record

Hudi identifies unique records using the **HoodieKey** model, which consists of **recordKey** and **partitionPath**. These values are populated by implementing the **KeyGenerator** API. This API offers flexibility in extracting and transforming custom fields into **HoodieKey** based on the input schema. For usage examples, see the docs.

Both **currentLocation** and **newLocation** are composed of a Hudi timeline's action timestamp and a file group's ID. Recalling the logical file group and file slice concepts from Chapter 1, the timestamp points to a file slice within a specific file group. The location properties are employed to locate physical files using logical information. If **currentLocation** is not null, it indicates where a record with the same key exists in the table, while **newLocation** specifies where the incoming record should be written.

The **data** field is a generic type that contains the actual bytes for the record, also known as the *payload*. Typically, this property implements **HoodieRecordPayload**, which guides engines on how to merge an old record with a new one. Starting from release 0.13.0, a new experimental interface, **HoodieRecordMerger**, has been introduced to replace **HoodieRecordPayload** and serve as the unified merging API.

3. Start commit

At this step, a write client checks if there are any failed actions remaining in the table's timeline, and performs a rollback accordingly, before initiating the write operation by creating a "requested" commit action.

4. Prepare records

The provided **HoodieRecords** may optionally undergo deduplication and indexing based on the user configurations and the operation type. If deduplication is necessary, records with the same key will be merged into a single record. If indexing is required, the **currentLocation** will be populated if the record exists. There's a lot more to say about indexing logic, but for the purposes of understanding writer flows, it is important to remember that an index is responsible for locating physical files for the given records.

5. Partition records

This essential pre-write step determines which record goes into which file group and, ultimately, which physical file. Incoming records will be assigned to "update" buckets and "insert" buckets, implying different strategies for subsequent file writing. Each bucket represents one RDD partition for distributed processing, as is the case with Spark.

6. Write to storage

This is when the actual I/O operations occur. Physical data files are either created or appended to using file writing handles. Before that, marker files may also be created in the **.hoodie/.temp/** directory to indicate the type of write operation that will be performed for the corresponding data files. This is valuable for efficient rollback and conflict resolution scenarios.

7. Update index

After data is written to disk, there may be a need to immediately update the index data to ensure read and write correctness. This applies specifically to index types that are not synchronously updated during writing, such as the HBase index hosted in an HBase server.

8. Commit changes

In this final step, the write client will undertake multiple tasks to correctly conclude the transactional write. For example, it may run pre-commit validation if configured, check for conflicts with concurrent writers, save commit metadata to the timeline, reconcile **WriteStatus** with marker files, and so on.

Write Operations

In this section, we'll delve into the **UPSERT** flow for a CoW table in detail, followed by a brief overview of all other supported write operations.

UPSERT

In a Hudi **UPSERT**, records are first tagged as inserts or updates and optimized for storage before being written. Figure 3-3 describes the Hudi UPSERT flow for a CoW table:

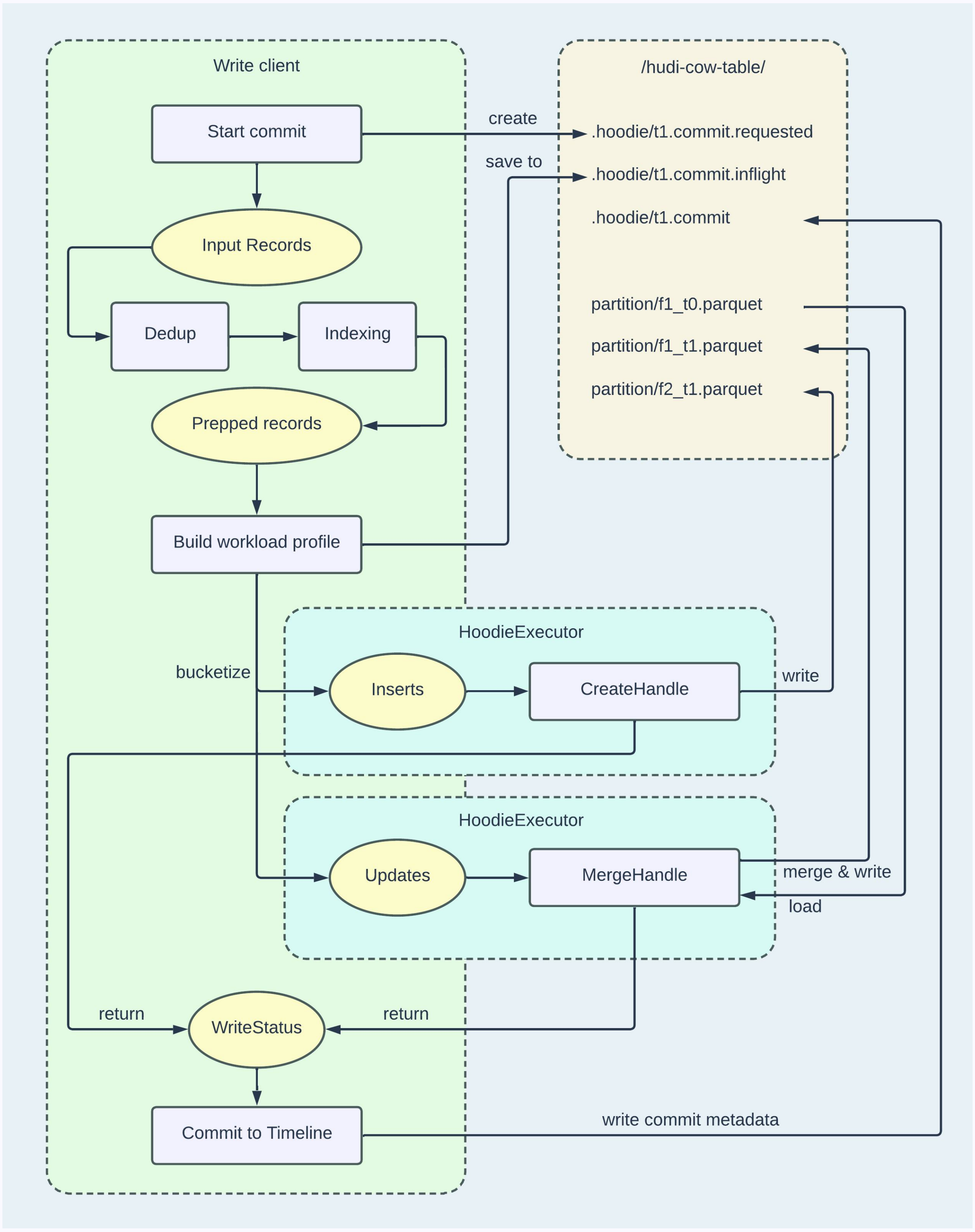


*Figure 3-3: The **UPSERT** flow for a Hudi CoW table*

An **UPSERT** to a Hudi CoW table involves the following steps:

1. The write client starts the commit and creates the "requested" action in the timeline.
2. Input records undergo the preparation step, in which duplicates are merged and target file locations are populated by the index. At this point in the process, we have the exact records to be written, and we know which of those exist in the table, along with their respective locations (i.e., file groups).

3. Prepared records are categorized into "update" and "insert" buckets. Initially, a **WorkloadProfile** is constructed to gather information on the number of updates and inserts in the relevant physical partitions. This data is then serialized into an "inflight" action in the timeline. Subsequently, based on the **WorkloadProfile**, buckets are generated to hold the records. For updates, each file group is assigned as an update bucket. In the case of inserts, any base file smaller than a specified threshold (determined by **hoodie.parquet.small.file.limit**) becomes a candidate for accommodating the inserts, with its enclosing file group being designated as an update bucket. If no such base file exists, "insert" buckets will be allocated, and new file groups will be created for them later.
4. The bucketized records are then processed through file writing handles for actual persistence to storage. In the case of records in the "update" buckets, "merge" handles are used, resulting in the creation of new file slices within the existing file groups (achieved by merging with data from the old file slices). For records in the "insert" buckets, "create" handles are utilized, leading to the creation of entirely new file groups. This process is done by **HoodieExecutors**, which employ a producer-consumer pattern for reading and writing records.
5. Once all data is written, the file writing handles return collections of **WriteStatus** that contain metadata about the writes, including the number of errors, the number of inserts performed, the total written size in bytes, and more. This information is sent back to the Spark driver for aggregation. If no errors have occurred, the write client will generate commit metadata and persist it to indicate a completed action in the timeline.

An **UPSERT** to an MoR table follows a very similar process, with a different set of conditions to determine the types of file-writing handles used for updates and inserts.

INSERT

The **INSERT** flow is very similar to **UPSERT**, with the key difference being the absence of an indexing step. This implies that the entire **INSERT** process is faster than **UPSERT**, especially if deduplication is turned off. However, this may result in duplicates in the table, causing data files to be larger than they need to be.

BULK_INSERT

BULK_INSERT follows the same semantics as **INSERT**, meaning it can also result in duplicates, due to the absence of indexing. Overall, **BULK_INSERT** is generally more performant than **INSERT**, but may require additional configuration tuning to address small-file issues. The records partitioning strategy is determined in one of two ways:

- Setting **BulkInsertSortMode**
- Implementing **BulkInsertPartitioner** for customization

By default for Spark, **BULK_INSERT** also enables row-writing mode, bypassing Avro data model conversion at the input transformation step and working directly with the engine-native data model **Row**. This supports even more efficient writes.

DELETE

The **DELETE** operation can be viewed as a special case of **UPSERT**. The primary difference is that, during the input transformation step, input records are transformed into **HoodieKeys** and passed on to subsequent stages, as these are the minimum required data for identifying the records to be deleted. It's important to note that this process results in a hard delete, meaning that the target records will not exist in the new file slices of the corresponding file groups.

DELETE_PARTITION

DELETE_PARTITION follows a completely different flow compared to the operations previously introduced, including **DELETE**. Instead of input records, **DELETE_PARTITION** takes a list of physical partition paths, which is configured via **hoodie.datasource.write.partitions.to.delete**. Because there are no input records, processes such as indexing, partitioning, and writing to storage do not apply. **DELETE_PARTITION** saves all file group IDs for the target partition paths in a **.replacecommit** action in the timeline, ensuring that subsequent writers and readers treat them as deleted.

INSERT_OVERWRITE

INSERT_OVERWRITE completely rewrites partitions with the provided records. This flow can be effectively seen as a combination of **DELETE_PARTITION** and **BULK_INSERT**; it extracts affected partition paths from the input records, marks all existing file groups in those partitions as deleted, and creates new file groups to store the incoming records.

INSERT_OVERWRITE_TABLE

INSERT_OVERWRITE_TABLE is a variation of **INSERT_OVERWRITE**. Instead of extracting affected partition paths from input records, it fetches all partition paths of the table for the purpose of overwriting.

Recap

In this chapter, we explored the high-level steps of Hudi write operations, using an **UPSERT** to a CoW table as the main example, and discussed other available write operations.

Chapter 4:
All about Writer Indexes

We'll now discuss one of the most crucial steps in the Hudi write process in detail: indexing. Indexing verifies the existence of records in the table and helps achieve efficient update and delete operations. This chapter will introduce the writer indexing APIs, and explore various types of indexes and their internal flows. Please note that the indexes covered in this chapter are intended for writers, not for readers.

Indexing APIs

Writer-indexing abstractions are defined in **HoodieIndex**. Some key APIs involved in indexing are:

- **tagLocation()**: This API is invoked when a set of input records is passed to the index component during writing. The API tags each record, determining whether it is present in the table, and then associates it with its location information. The resulting set of records is referred to as *tagged records*. In the **HoodieRecord** model introduced in Chapter 3, the **currentLocation** field is populated by this tagging process.
- **updateLocation()**: After data is written to storage, **updateLocation()** provides certain indexes with required location information to be updated to synchronize with the data table. This process is only executed during the post-I/O phase for those applicable index types.
- **isGlobal()**: Hudi categorizes indexes into *global* and *non-global* types. Global indexes identify unique records across all table partitions, and are therefore global in relation to the table. Non-global indexes, on the other hand, validate uniqueness at the partition level. Typically, non-global indexes exhibit better performance due to their smaller scan space. However, they are not suitable for tables with records that can shift between partitions. This API determines if an index is global or not.
- **canIndexLogFiles()**: Due to the implementation specifics, certain indexes are able to index on log files for MoR tables. This characteristic affects how writers create file-writing handles; when this is true for the configured index, inserts will be routed to log files through **AppendHandle()**.
- **isImplicitWithStorage()**: This API indicates whether the index is implicitly persisted along with data files on storage. Some indexes store their indexing data separately.

Index Types

Hudi offers several out-of-the-box index types to support different traffic patterns and table sizes. Selecting the most appropriate index for each table is a crucial tuning step. In the following sections, we'll describe various writer index types and their internal workings, to further enhance understanding of when and how to use these indexes.

Simple Index

The *simple index* is a non-global index that serves as the default type, and is shown in Figure 4-1. A simple index scans all base files within the relevant partitions to determine whether incoming records match any of the extracted keys. Since the simple indexes tend to load all base files at either the partition level or the table level, they are well suited for traffic patterns having random or evenly-distributed data access.

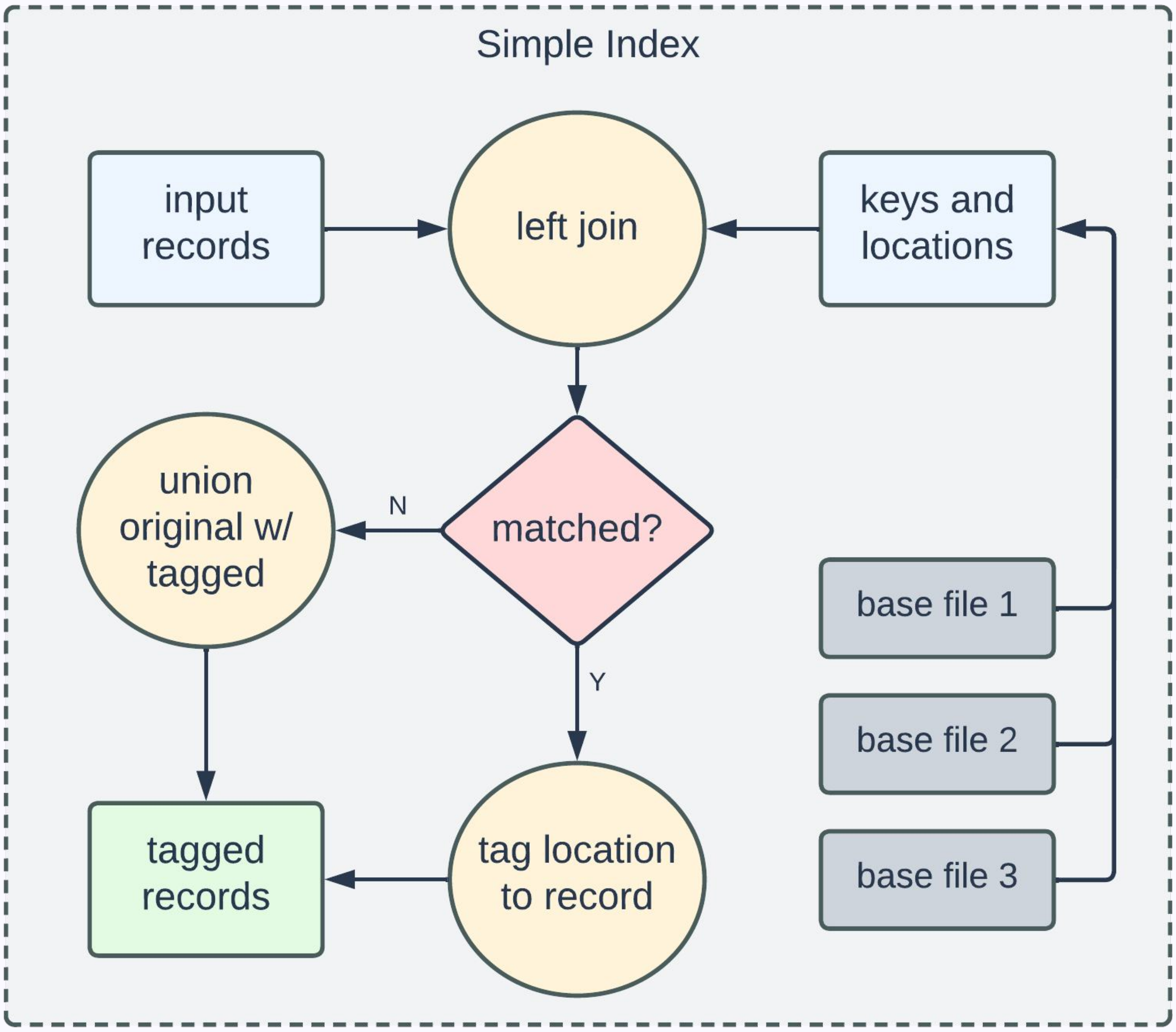


Figure 4-1: The flow of a Hudi simple index

From the left-join operation, if an input record matches an extracted key, the join result will include the location information, which will then be used to populate the **currentLocation** field of the **HoodieRecord**. This produces the tagged records previously described. A union operation is performed on unmatched records and tagged records for further processing.

Global Simple Index

The simple index has a global version known as the *global simple index*, which matches input against base files from all partitions, rather than just the relevant ones. When a record's partition value is updated, the respective file group is loaded, including log files for MoR tables. As an additional tagging step, this index merges the incoming record with its pre-existing version and tags the merged result to the location in the new partition.

Bloom Index

The *Bloom index* follows a similar high-level flow to the simple index (see Figure 4-2). However, the distinguishing concept behind the Bloom index lies in its approach to minimizing the number of keys and files for look-ups while maintaining a low read cost.

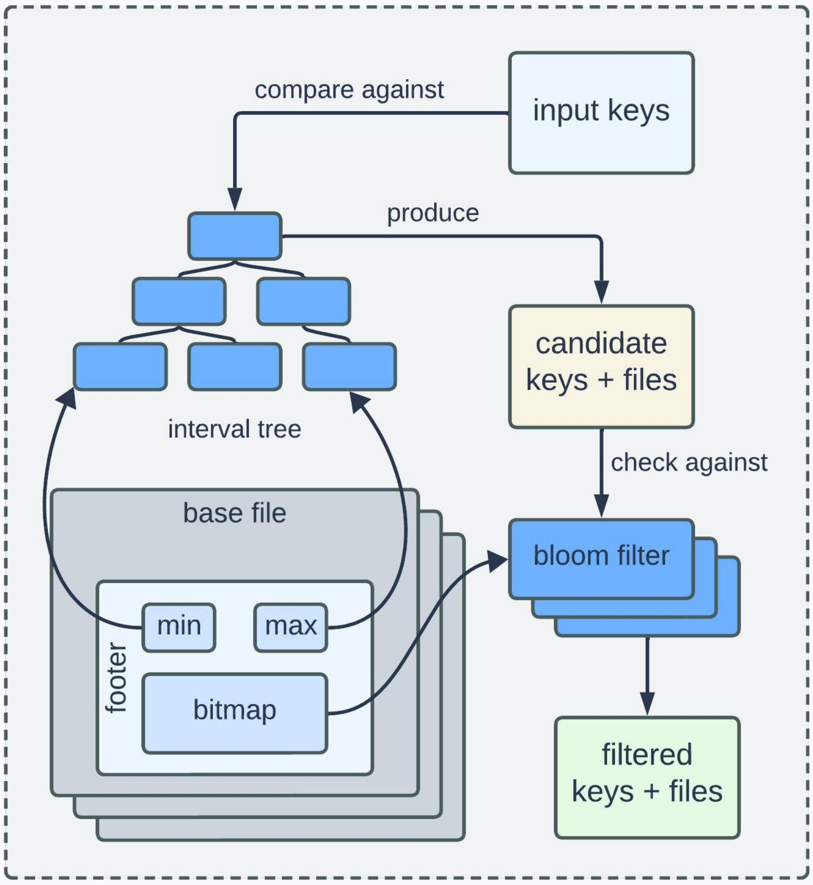


Figure 4-2: The flow of a Hudi Bloom index

The Bloom index employs two-stage filtering to reduce the number of keys and files for look-ups:

- First stage: Input keys are compared to an interval tree constructed using minimum and maximum record key values stored in base file footers. Keys falling out of these ranges represent new inserts, while the remaining keys are considered candidates for the next stage.
- Second stage: This stage checks the candidate keys against deserialized Bloom filters, which help determine the definitively absent keys and the potentially present keys. Actual file look-ups are then carried out using the filtered keys and the associated base files. The result returns the key and location tuples for tagging.

Note that the filtering process before the look-ups only involves reading the file footers, thereby incurring low read costs.

Global Bloom Index

Just like the simple index, the Bloom index also has a global version known as the *global Bloom index*. It operates similarly to the non-global version, albeit at the table level, and it employs the same logic as the global simple index for handling partition-update scenarios.

Bucket Index

The *bucket* index maps a key to a file group using a fixed hashing function, eliminating the need for disk reads and resulting in significant time savings.

The bucket index comes in two variations:

- *Simple bucket index*: This index assigns a fixed number of buckets, each mapping to one file group, which in turn limits the total number of file groups in the table. This leads to limitations on handling data skewness and scaling out.
- *Consistent-hashing bucket index*: This index is designed to overcome the drawbacks of the simple bucket index by dynamically rehashing an existing bucket into sub-buckets when the corresponding file group exceeds a certain size threshold.

HBase Index

The *HBase index* is a global index implemented using an external HBase server. It stores the mappings between a record key and the relevant file group information. This index offers efficient look-ups for tagging and can readily scale out as the table size increases. However, one drawback of this index is the operational overhead involved in managing an additional server.

Record-level Index

The record-level index, available since release 0.14.0, is logically similar to the HBase index. Like the HBase index, it is also a global index that saves the mappings of record keys and file groups. The key difference is that the record index keeps the indexing data local to the Hudi tables, thus avoiding the cost of operating an extra server. To learn more, see this Hudi blog post.

Recap

In this chapter, we discussed Hudi indexing APIs for writers, delved into the detailed flows of the simple index and the Bloom index, and briefly introduced the bucket index, the HBase index, and the record-level index.

Chapter 5: Introducing Table Services - Compaction, Cleaning, and Indexing

Now that we've delved into the details of read and write, we'll discuss table services. We'll first introduce high-level concepts, then cover three specific table services: compaction, cleaning, and indexing.

Overview

Table services are a type of maintenance job that operates on a table without adding new data. When ingesting new records, we often prioritize low latency, which may lead to trade-offs such as extra copies of records, resulting in sub-optimized storage. Table service jobs improve storage layout, paving the way for more efficient read and write processes.

In general, a table service job consists of two steps:

1. *Scheduling*: This step generates an execution plan that defines changes to be made to the table.
2. *Execution*: This step carries out the execution plan and makes the actual changes to the table.

Hudi table services can operate in three modes: inline, semi-async, and full-async (see Figure 5-1). Each mode provides different functionality for flexibility for various real-world scenarios.

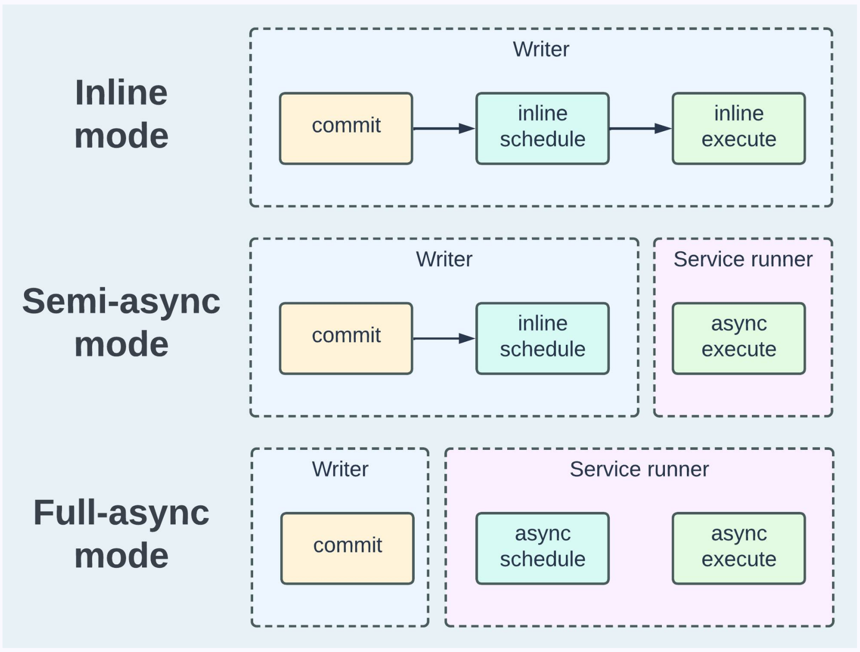


Figure 5-1: The scheduling and execution strategies of Hudi table services

Different modes use different strategies for scheduling and execution, with various trade-offs and benefits:

- *Inline mode*: In this mode, both the "schedule" and "execute" steps occur synchronously after the writer commits; hence, the use of the term "inline." This requires the simplest operational effort, as the two steps are automatically executed in sequence within the existing writer process. However, as a trade-off, significant latency may be introduced to the writing process.
- *Semi-async mode*: This mode maintains inline scheduling, but separates execution. In other words, the execution step is asynchronous to the writer process. In this mode, users have the flexibility to deploy the service runner as a separate job or even to a different cluster, which might be necessary due to high computational requirements of the service execution.
- *Full-async mode*: This mode is the most flexible, as it decouples table services from write processes. This is particularly helpful in managing a large number of tables in a lakehouse project, where a dedicated scheduler can be employed to optimize both scheduling and execution.

Table Services

As of release 0.15.0, Hudi offers four table services: clustering, compaction, cleaning, and indexing. In the following sections, we'll explore three of these services: compaction, cleaning, and indexing. Clustering will be discussed in Chapter 6.

Compaction

Recall from Chapter 1 that CoW tables create a new base file automatically on writes, so there is no need for a separate compaction process. But for MoR tables, the file slice evolves by adding log files alongside the base file; this makes writes faster, but slows performance on reads. (Because the reader has to examine log files as well as the base file in determining the result.) MoR tables require a separate compaction step to merge log files into the then-existing base file, creating a new base file in the process.

Compaction jobs can be quite resource-intensive due to the high write amplification that occurs when re-writing base files, so it's important to run them during less-busy periods where possible. In addition to file compaction, there is also an experimental table service, log compaction. This feature was initially introduced in release 0.13.0 to address the write-amplification issue by only compacting log files into larger ones.

There are quite a few configuration options to manage when scheduling and executing compaction. The documentation provides detailed examples that showcase the usage. In this chapter, our focus is on the generalized internal workflow, as illustrated in Figure 5-2.

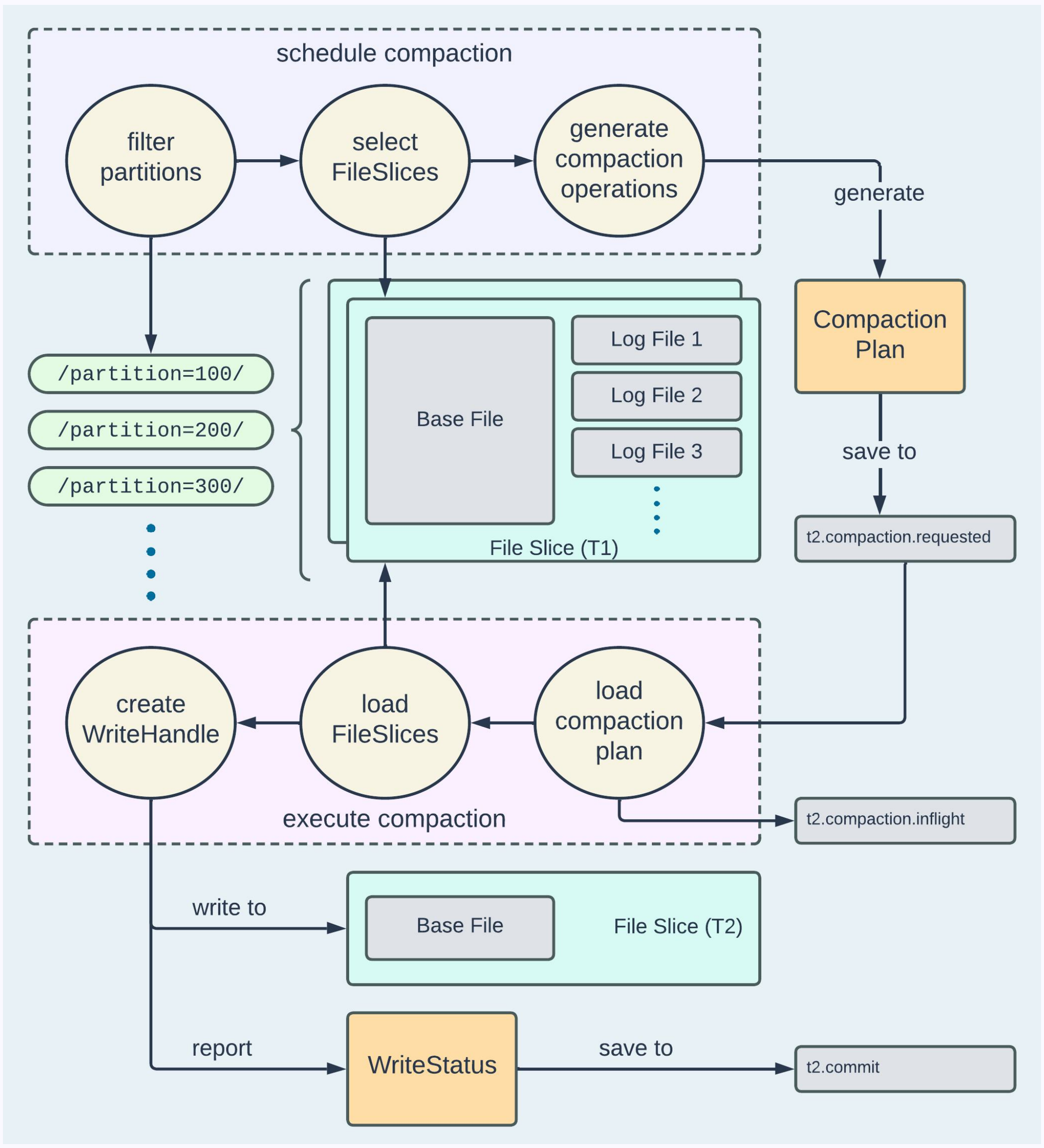


Figure 5-2: The Hudi compaction workflow

Compaction scheduling and execution occurs as follows:

1. The scheduling step determines whether compaction is necessary based on the configurable **CompactionTriggerStrategy**. If deemed necessary, this step generates a compaction plan and saves it to the timeline as a **.compaction.requested** action. Users can set the triggering threshold based on factors such as number of commits or elapsed time. If the criteria are met, a compaction plan generator will scan the table based on the **CompactionStrategy**, which essentially controls which file slices should be compacted. This produces a **CompactionOperation** for each file slice to formulate a plan.

2. The execution step loads the serialized **CompactionOperations** from the plan and runs them in parallel. Depending on the presence of the base file in the target file slice, either **MergeHandle** or **CreateHandle** will be used to write the merged records in a new file slice. Similar to a write process, a group of **WriteStatus** objects will be returned, reporting statistics collected during the execution, and a **.commit** action will be saved in the timeline, marking the success of the compaction.

Users may also monitor performance vs. table write and read load and manually initiate compactions before triggers are reached, reducing the odds that users will be affected significantly by having a compaction operation occurring at a busy time.

Cleaning

For incoming data, Hudi tables continually add file slices to represent newer versions, taking more disk space. *Cleaning* is the table service designed to reclaim storage space by deleting old and unwanted versions, as illustrated in Figure 5-3.

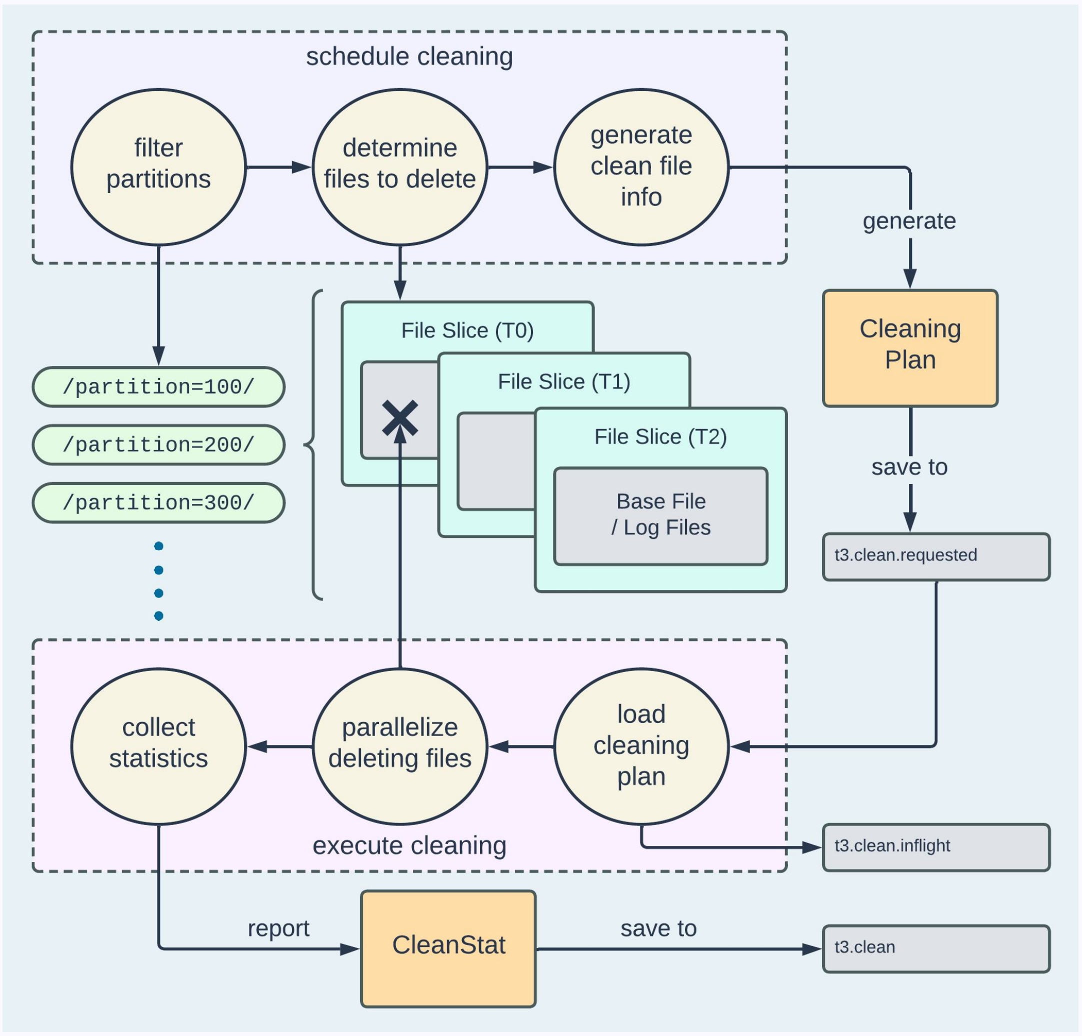


Figure 5-3: The Hudi cleaning workflow

The scheduling and execution steps are as follows:

Similar to compaction, Hudi uses **CleaningTriggerStrategy** to determine if cleaning is required at the time of scheduling, where the specified number of commits is the trigger criteria. After the configured threshold for number of commits is reached, a cleaning planner will scan relevant partitions and determine if any file slice meets the criteria for cleaning, as defined by **HoodieCleaningPolicy**. Physical paths of either base files or log files from the eligible file slices will be used to generate a group of **CleanFileInfo**. A cleaning plan is then formulated based on that and saved into a **.clean.requested** action.

At this time, three cleaning policies are supported: **clean-by-commits**, **clean-by-file-versions**, and **clean-by-hours**. During the execution step, the job performs file-system deletes for the target files in parallel after loading the plan and deserializing the **CleanFileInfo**. Statistics are initially collected at the partition level, and then aggregated and saved into a **.clean** action, indicating process completion.

For more detailed usage information, see the cleaning documentation.

Indexing

The *indexing* table service, initially added in release 0.11.0 as an experimental feature, is designed for building indexes for the metadata table. In this section, we'll provide a brief overview of the design. For a deeper technical dive into the indexing process, see the docs, this Onehouse blog post, and RFC-45.

Recall the indexing API **updateLocation()** described in Chapter 4. This API is required by certain indexes to keep the indexing data in sync with the written data. From a table service perspective, we can view this API as an indexing operation running in inline mode; in other words, the scheduling and execution steps are performed inline. The current indexing service can be considered to run in full-async mode.

The metadata table can be seen as another index type that encompasses multiple indexes, also known as a *multi-modal index*. As the data table size grows, updating the metadata table inline with each write can be time-consuming. Therefore, we need the async table service to maintain high write-throughput, while keeping the indexes current.

Recap

In this chapter, we introduced the general concept of table services, and we discussed three of them: compaction, cleaning, and indexing. The other table service, clustering, will be discussed in the next chapter.

Chapter 6: Demystifying Clustering and Space-Filling Curves

We’ll now explore the concept of *proximity*, focusing on clustering techniques and layout strategies to improve read efficiency. Using the analogy of a 2D plane and the mathematical determination of proximity, we'll establish a foundation for understanding its implications in multidimensional datasets. By illuminating the intricate interplay between proximity, clustering, layout optimization, and query performance, this chapter provides essential insights into enhancing data storage efficiency and query performance.

Proximity Explained

To illustrate the concept of proximity, we'll use the analogy of a 2D plane with X and Y axes to represent a dataset. In this analogy, if the dataset's schema has two columns, X and Y, records are considered to have close proximity when the coordinate pairs (X, Y) are close to each other on the 2D plane. However, in practice, a complex and/or wide schema with numerous columns will require more dimensions to be added. While visualizing high-dimensional spaces is challenging for 3D beings like ourselves, proximity in higher-dimensional spaces can still be determined mathematically, allowing computers to process the information.

Clustering, in the context of data storage, stands as a valuable optimization technique to improve the storage layout by preserving data locality for better read efficiency. The three main motivations for clustering are as follows:

- Clustering improves query performance, especially when low-latency, high-throughput writes result in too many small files being created. Clustering consolidates and rewrites these small files into larger ones, which can effectively address the issue, especially when clustering is executed asynchronously to writing.
- Clustered records tend to show better alignment with file-level statistics such as column minimum and maximum values. During the process of rewriting data files, proximate records are more likely to be clustered in the same files, allowing data files to be skipped more effectively based on given predicates.
- Clustered data exhibits good spatial locality, and can therefore use a block cache, as with HDFS, to increase hit rate, resulting in faster reads.

Clustering Workflow

Similar to other table services mentioned in Chapter 5, clustering can be run in three modes: inline **(hoodie.clustering.inline)**, semi-async **(hoodie.clustering.schedule.inline)**, and full-async **(hoodie.clustering.async.enabled)**. For further information on how to configure these flags, see the docs.

As clustering involves rewriting data, a **.replacecommit** will be generated upon the completion of the table service job, indicating that the eligible file groups have been rewritten into new ones. The clustering workflow, consisting of scheduling and execution, is illustrated in Figure 6-1.

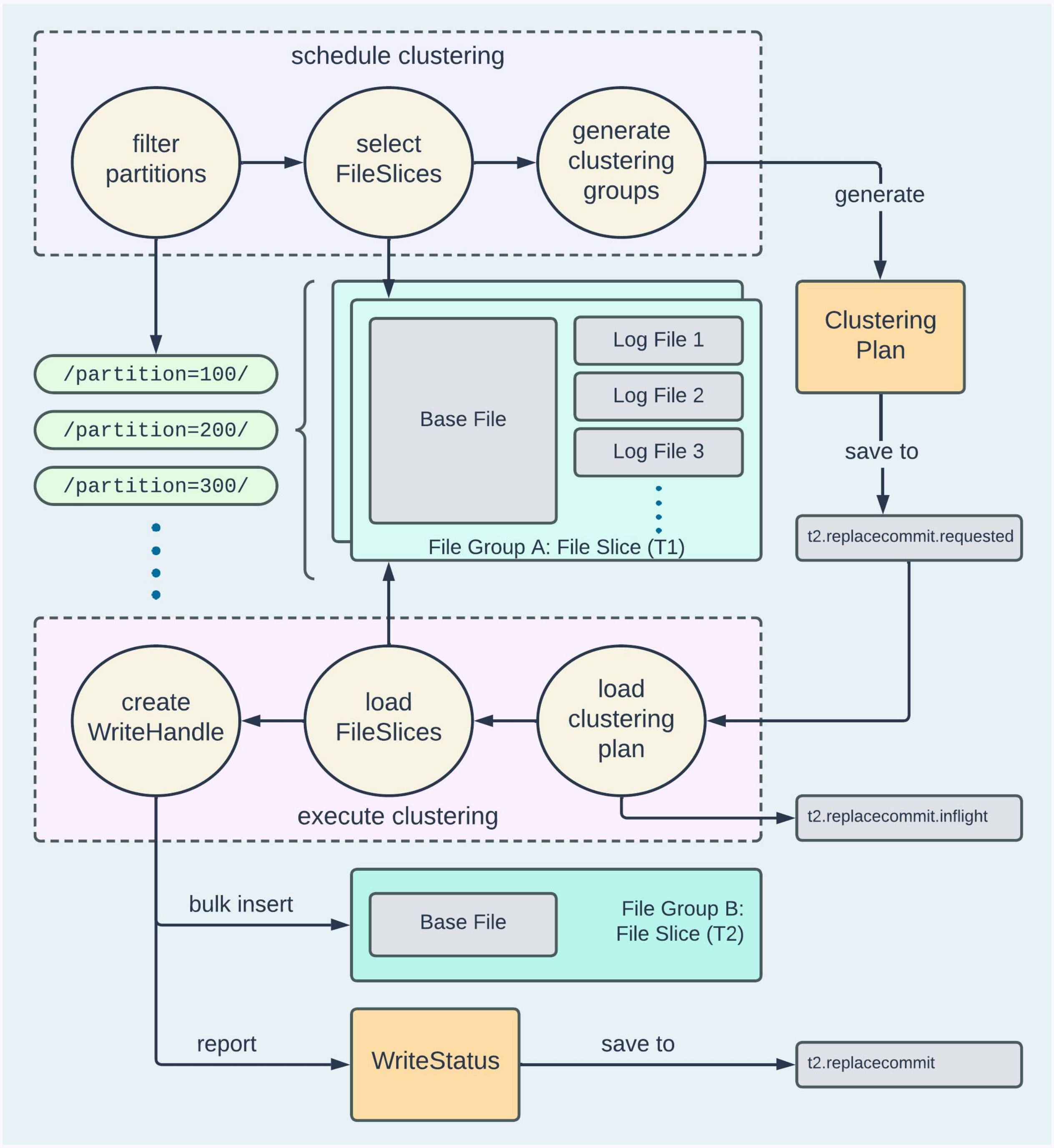


Figure 6-1: The Hudi clustering workflow

The clustering workflow is similar to the compaction workflow, with the following steps:

1. During the scheduling phase, eligible partitions and file slices are selected based on **ClusteringPlanStrategy**. Users have the flexibility to define partition patterns to target specific partitions using regular expressions. Within these partitions, file slices meeting certain criteria (not currently undergoing compaction, not qualifying as small files, etc.) are added to **HoodieClusteringGroups**. These entities store information about the input and output for subsequent clustering execution. Typically, **HoodieClusteringGroup** adheres to size limits, such as the maximum total bytes of file slices to include for rewriting. The total number of **HoodieClusteringGroups** is also capped by default, preventing unintentional submission of resource-intensive clustering jobs.
2. The steps for the execution phase of clustering are as follows:
   a. Deserialize the clustering plan.
   b. Load the designated input file slices.
   c. Merge the loaded records.
   d. Bulk insert the merged records to new file groups.
   e. Report write statistics through the returned **WriteStatus**.

Users can customize the execution step by supplying their own implementation of **ClusteringExecutionStrategy**. By default, each **HoodieClusteringGroup** defined in a clustering plan will be submitted as a separate job to perform parallel rewriting of file slices.

By default, for file groups undergoing a clustering process, writers will fail if updates or deletes on those file groups are intended. However, in the case of running table services, failing writes may not be ideal. Other pluggable strategies exist that allow updates to proceed, followed by resolving conflicts or enforcing dual writes on both the old and new file groups.

The record proximity mentioned in the chapter overview comes into play at the bulk insert step, where records are re-partitioned and sorted according to **hoodie.layout.optimize.strategy**. We'll describe this in detail in the next section.

Layout Optimization Strategies

Hudi offers three layout optimization strategies: linear, Z-order, and Hilbert. Each of these defines how records should be sorted during bulk insert. The default strategy is linear optimization, which performs lexicographical sorting. The other two optimization strategies, Z-order and Hilbert, are known as space-filling curves because they sort and preserve good spatial locality.

The linear strategy is highly effective for datasets in which record proximity relies on just one column. For instance, consider a table containing transaction records with a timestamp column. Analysts often run queries to fetch all records between transaction time A and transaction time B. Given that the records are considered proximally close as long as the transaction timestamps are close, linear sorting by the timestamp is a perfect strategy to preserve locality.

However, the linear strategy may not perform well with datasets that require two or more columns to determine record proximity. For example, consider a house inventory dataset with columns for latitude and longitude. Lexicographical sorting of latitude followed by longitude would group geographically distant house records together simply based on the proximity of latitude. In such cases, sorting algorithms that are capable of handling N-dimensional records are needed. This is where Z-order and Hilbert optimization strategies can be applied.

The mathematical term *space-filling curve* describes a curve that traverses a space, intersecting with all possible points in that space and thereby filling it entirely. Once the curve is straightened, all the multi-dimensional points are mapped to a one-dimensional space and assigned a single-value coordinate. Among the various curve-drawing methods, Z-order and Hilbert, as shown in Figure 6-2, are two approaches that can effectively preserve spatial locality through this mapping, as the majority of nearby points on the curve are also close to each other in the original space.

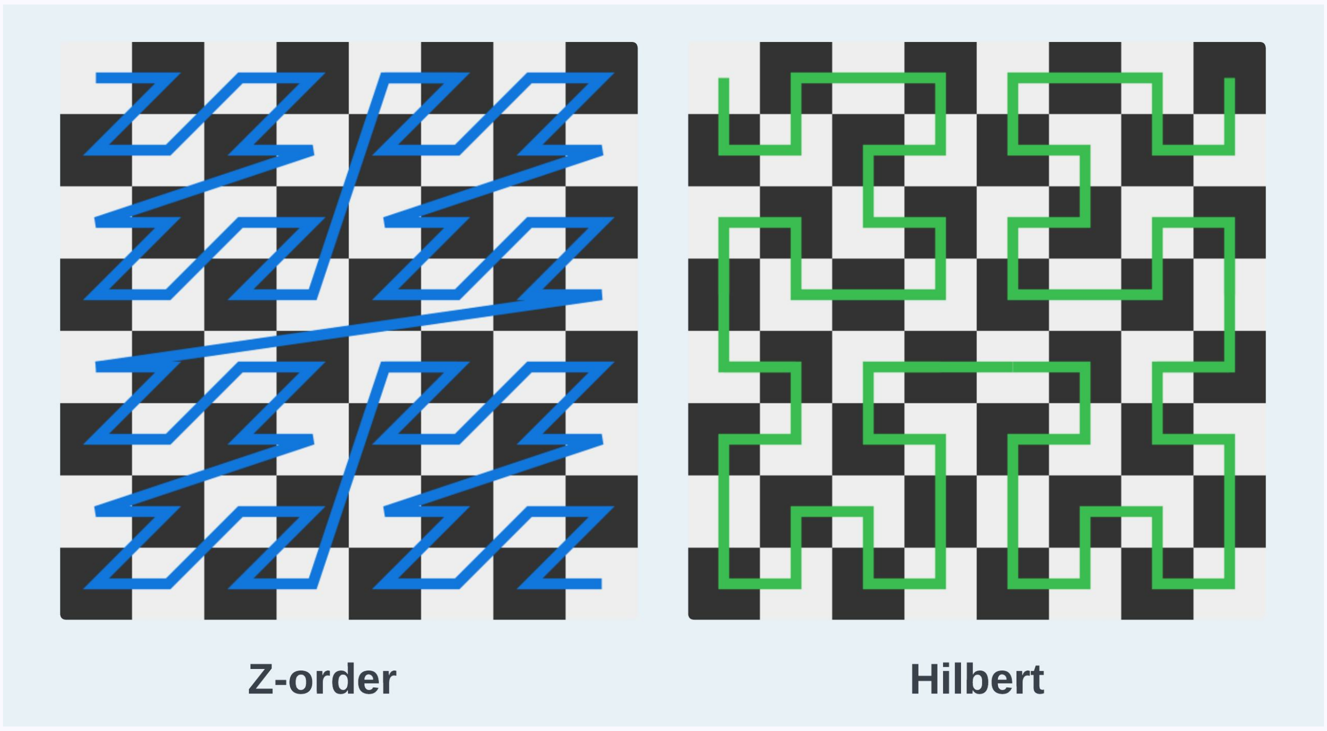


Figure 6-2: Z-order and Hilbert curves in a two-dimensional space (with thanks to the image's creator)

When we treat records as multi-dimensional points, drawing a Z-order or Hilbert curve defines an optimal way to sort them. Given that spatial locality is well preserved, nearby records are more likely to be stored in the same files. This fulfills the proximity condition and enhances read efficiency.

Recap

In this chapter, we described table service clustering techniques and their pivotal role in optimizing data storage for efficient reads. By exploring the concept of proximity and its significance within multidimensional datasets, we've provided valuable insights into the clustering workflow and the utilization of space-filling curves for enhanced storage layout. Through these discussions, readers are equipped with essential knowledge to leverage clustering methods and layout optimization strategies, and to thereby improve data retrieval efficiency and query performance.

Chapter 7:
Run Writers and Table Services Concurrently

With the knowledge gained in previous chapters, we'll now discuss concurrency control, focusing specifically on managing concurrency for writers and table services.

A Primer on Concurrency Control

Every commit to a Hudi table is a transaction, whether it involves adding new data or executing a table service job. *Concurrency control* is the practice of orchestrating concurrently executed transactions to ensure correctness and consistency while maintaining optimal performance. This primer aims to offer just enough information to provide the needed context for subsequent sections that delve into Hudi's implementation of concurrency control. To learn more about this complex topic, see this course and this paper.

In databases, atomicity, consistency, isolation, and durability (ACID), depicted in Figure 7-1, are the four essential properties required to maintain the integrity and reliability of transactions.

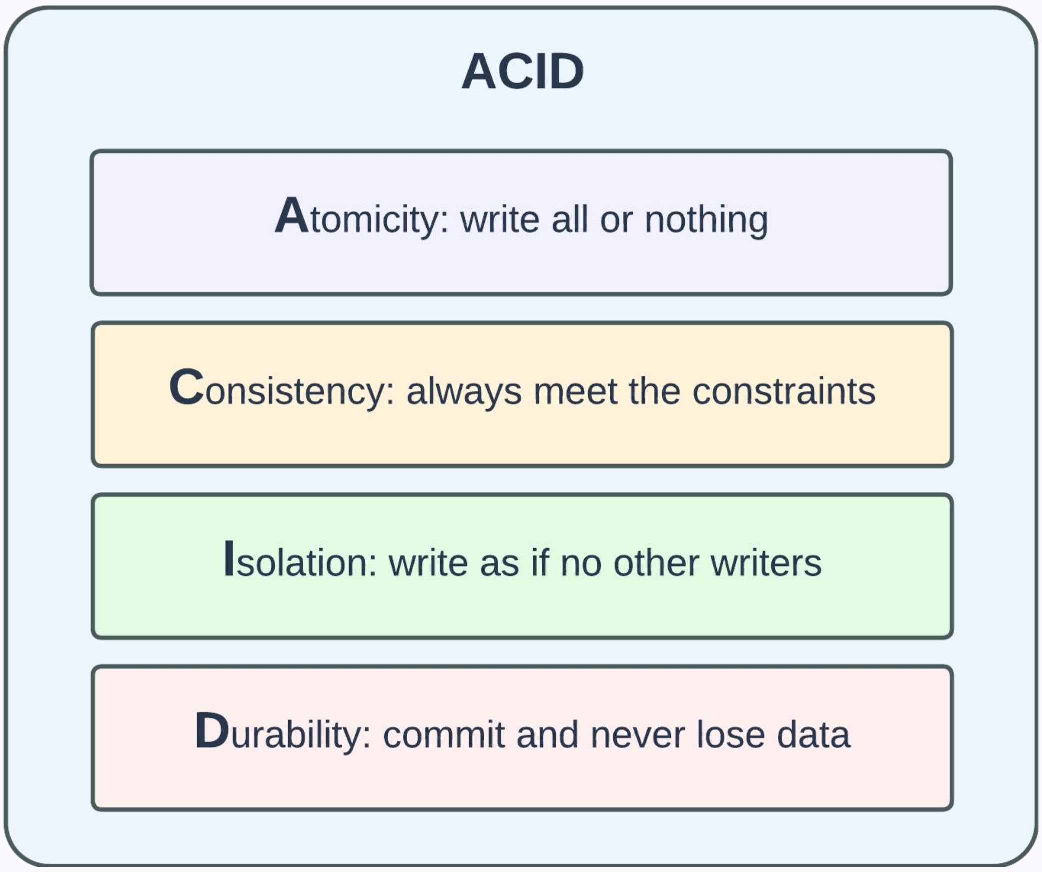

ACID
Atomicity: write all or nothing
Consistency: always meet the constraints
Isolation: write as if no other writers
Durability: commit and never lose data

Figure 7-1: The ACID principles of database design

Each property is focused on a particular aspect of transaction integrity and reliability:

- *Atomicity* requires that each transaction be treated as an indivisible unit of work; this implies that any changes made by the transaction should be rolled back in the event of a failure partway through an operation.
- *Consistency* is concerned with application-specific constraints. For example, a primary key field cannot have duplicates, or the product price column must be non-negative.
- *Isolation* ensures that concurrent transactions are isolated from each other, resulting in changes being made as if they are executed sequentially. If the Isolation property is not honored, concurrent transactions will incur read and write anomalies, such as dirty reads and writes, lost updates, and more. While enforcing a strictly serial execution of all transactions can eliminate the anomalies, this severely impacts performance, rendering the system practically unusable; hence, isolation is key.
- *Durability* mandates the preservation of committed data on storage, ensuring resilience against incidents such as hardware failures.

With these principles in mind, we should allow for concurrent execution for performance, and coordinate the results of execution so they are the same as would have resulted from performing the steps serially. In other words, what we need is a *serializable schedule*. Figure 7-2 depicts a serializable schedule for three transactions.

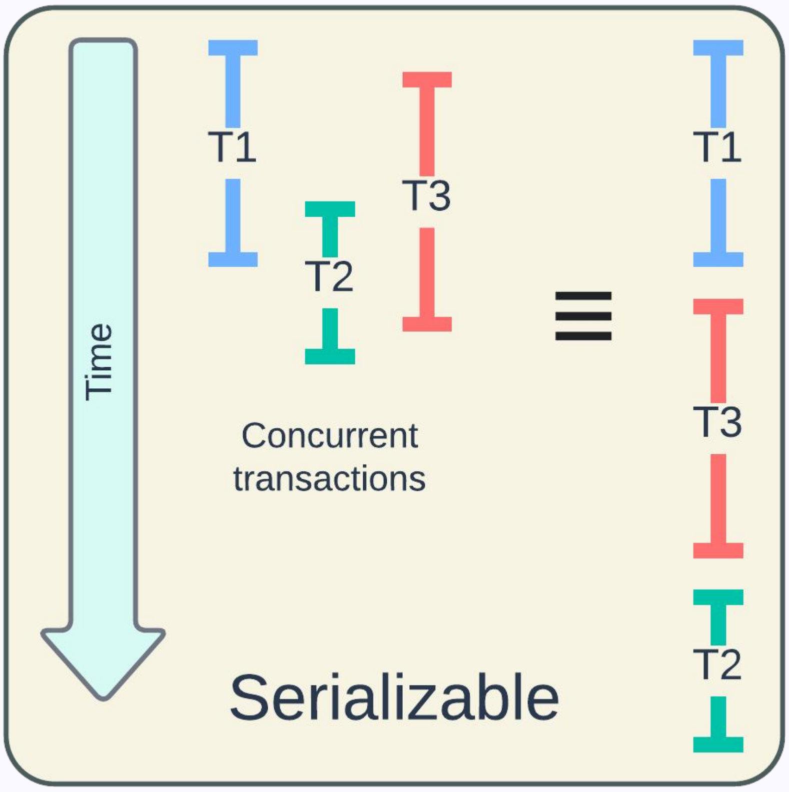


Figure 7-2: Execution of a serializable schedule for three transactions

Common strategies for enforcing serializable schedules in various database systems are *multi-version concurrency control (MVCC)* and *optimistic concurrency control (OCC)*. MVCC keeps multiple record versions on storage and associates them with monotonically increasing transaction IDs (i.e., timestamps). OCC "optimistically" allows concurrent transactions to proceed on their own and resolves any conflicts later. Initially, Hudi adopted MVCC for the handling of a single writer with concurrent table services without locking. In later releases, an OCC implementation was added to support multi-writer scenarios. In the upcoming sections, we'll explore how Hudi employs these strategies in dealing with concurrent writers and table services.

MVCC in Hudi

The timeline and file slices serve as the foundation to Hudi's MVCC implementation. The timeline uses monotonically increasing commit start times to keep track of transactions to the table. File slices handle record versioning and correspond to transaction timestamps. One layer above, Hudi constructs a view object, **TableFileSystemView**, which provides API calls to return the table's most recent storage states. Examples include the latest file slices under a partition path and file groups that undergo clustering.

Writers and readers always consult the table file system view to decide where to perform the actual I/O operations. This design, shown in Figure 7-3, provides read-write isolation, as the new data writing does not interfere with readers accessing past versions.

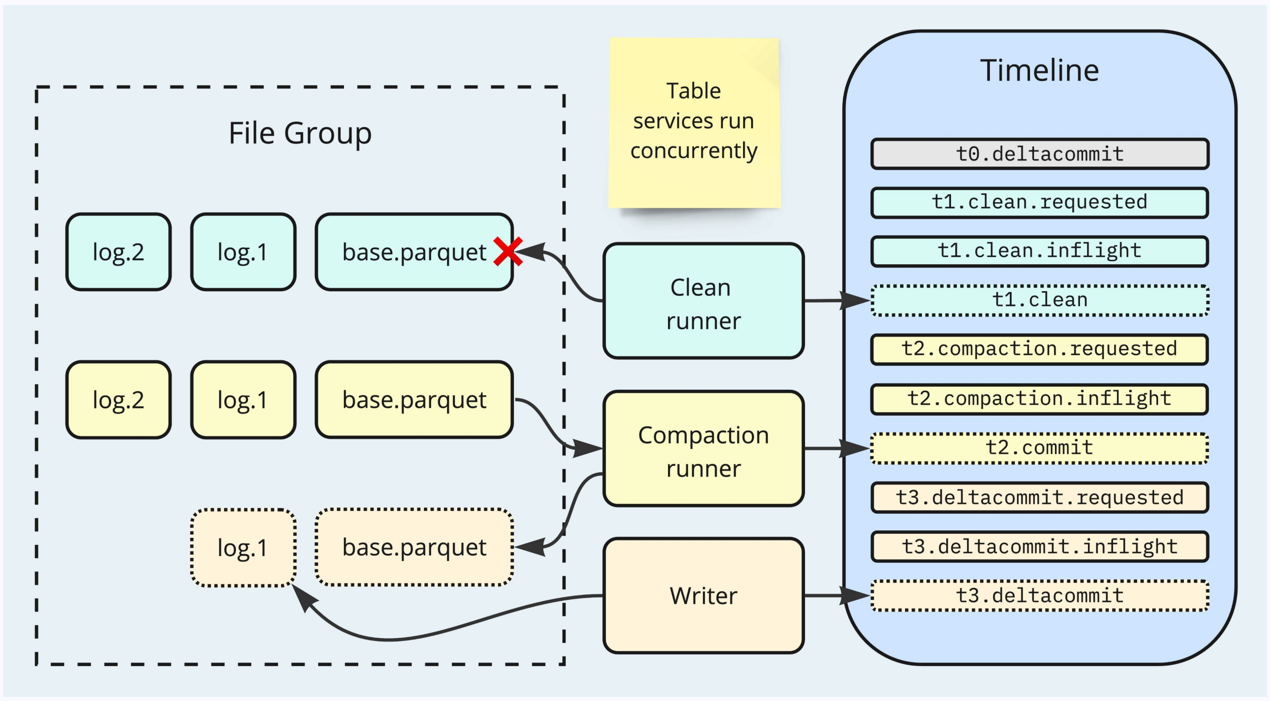


Figure 7-3: Using MVCC, Hudi table services run concurrently with a writer

When a write operation is in progress, a commit action indicating this write will be marked as "requested" or "inflight" in the timeline. This ensures that the table file system view is aware of the ongoing action, and that table service planners do not include the file slices currently being written for subsequent execution. This logic also holds true in the scenario of concurrent table service jobs. Hudi's table services are idempotent operations, because the plans containing information about which file slices to read are persisted. Therefore, retries in the event of failure won't impact the final result.

While a compaction operation is ongoing, any new write to the MoR table would either route new records to new file groups, or append updates and/or deletes to log files. As such, the base file that the compaction job is producing will be excluded by the view to prevent misuse. When clustering is pending, users can configure the writer's behavior to handle the updating of a file group that undergoes clustering. Possible behaviors include aborting the write, rolling back the clustering, deferring to later conflict resolution processes such as OCC, or performing dual-write operations to the source and target clustering file groups. Cleaning is always executed in a way that retains the latest file slices, keeping the deletes clear of new writes.

OCC in Hudi

An OCC protocol typically comprises three steps: read, validation, and write:

- In the read step, concurrent writers perform the necessary I/O operations to complete their work in isolation.
- The validation step collects the list of changes from each writer and determines if any conflicts exist.
- Lastly, during the write step, all changes are accepted if no conflicts are found. If conflicts arise, the changes from the writer with the later transaction time are rolled back.

This process is analogous to the GitHub workflow, where contributors can submit pull requests to the upstream repository. In GitHub, merging will be blocked for pull requests that have conflicts, similar to the validation phase in OCC.

As concurrent updates could lead to write anomalies, Hudi implements OCC with file-level granularity to handle multi-writer scenarios, as shown in Figure 7-4. To enable this feature, users need to set **hoodie.write.concurrency.mode** to **OPTIMISTIC_CONCURRENCY_CONTROL** and configure a lock provider accordingly. OCC is integrated into Hudi's write flow.

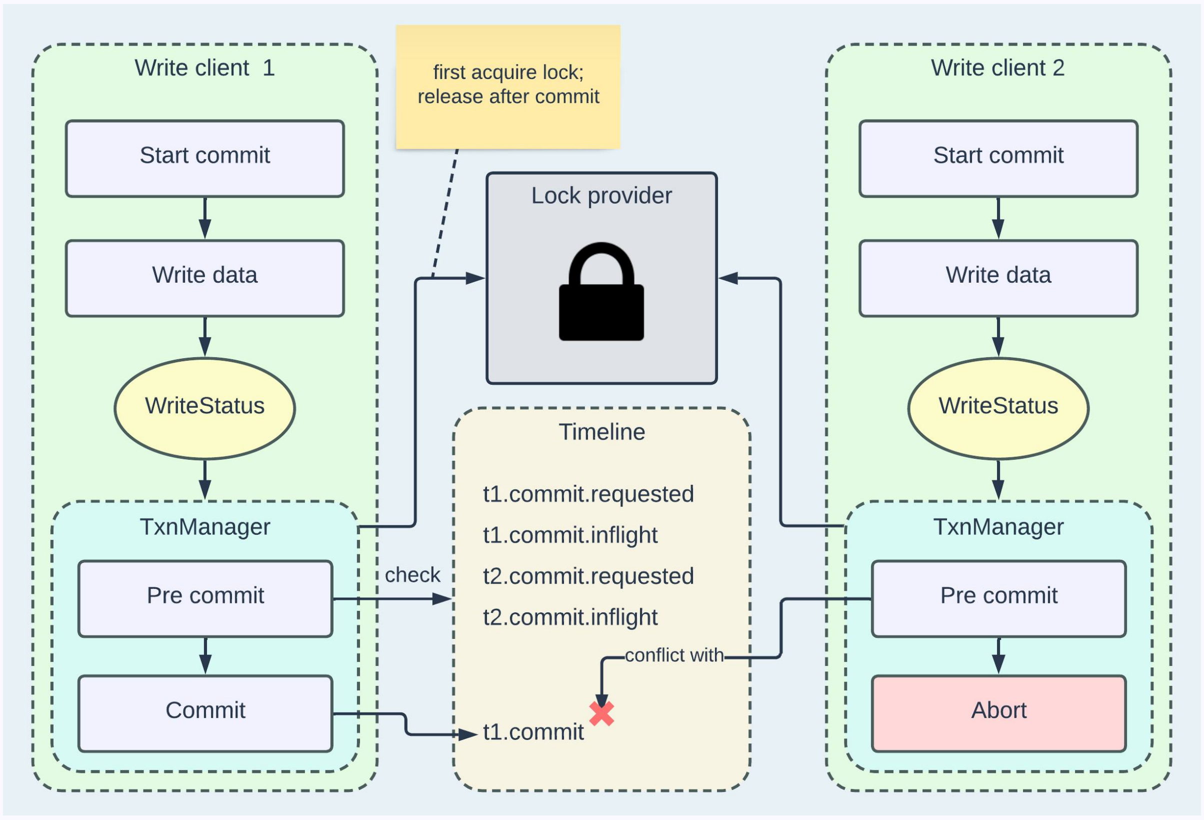


Figure 7-4: The use of OCC in a Hudi write operation with two write clients

The key steps are as follows:

- Write client 1 is writing **t1.commit** and is first to acquire a lock from the lock provider, which is usually implemented using an external server such as Zookeeper, Hive Metastore, or DynamoDB.
- While holding the lock, write client 1 checks the timeline to see if any concurrent commits have been completed before its own attempt. In this example, the **t2.commit** by write client 2 is the only candidate timeline to check against. As it's still in flight, write client 1 can proceed to commit and release the lock.
- Write client 2 writes **t2.commit** and acquires the lock after client 1 releases it. In the pre-commit phase, the files changed by client 2, obtained from **WriteStatus**, conflict with the files changed by client 1, derived from **t1.commit**. Consequently, client 2 will abort the write.

Aborted writes will be rolled back, implying the deletion of all the written files, both for data and metadata, as if the writes never occurred. While this process fulfills the atomicity requirement, it could also be wasteful, particularly when the conflict chances are high. As such, Hudi offers an early conflict detection mode for OCC. In this mode, before the actual files are written, lightweight marker files are created in a temporary folder. These markers serve as a preliminary step for conflict checking. For a detailed explanation of the design and implementation of early conflict detection, see this talk on YouTube.

Recap

In this chapter, we provided a brief overview of concurrency control and delved into the specifics of how Hudi implements concurrency control, specifically discussing MVCC and OCC.

Chapter 8:
Read and Process Incrementally

Building off of our overview of concurrency control mechanisms in Hudi, we now turn our attention to another crucial aspect of Hudi's functionality: incremental processing. We'll begin by providing a short overview of the incremental processing architecture before delving into two key features of incremental processing, the incremental query and change data capture (CDC). Through this exploration, we aim to provide a comprehensive understanding of how Hudi enables efficient and scalable incremental data processing workflows.

Overview

Incremental processing involves the extraction, loading, and transformation (ELT) of data subsets to maintain up-to-date results. This technique has become standard in constructing data pipelines for data lakehouses. Unlike traditional methods, which often require complete data snapshots or costly join operations, modern data lakehouses employ storage formats that support incremental processing, simplifying architecture.

Hudi adopts the medallion architecture, which consists of three layers: unchanged bronze tables for reprocessing needs, silver tables for data quality assurance, and gold tables for delivering business value (see Figure 8-1).

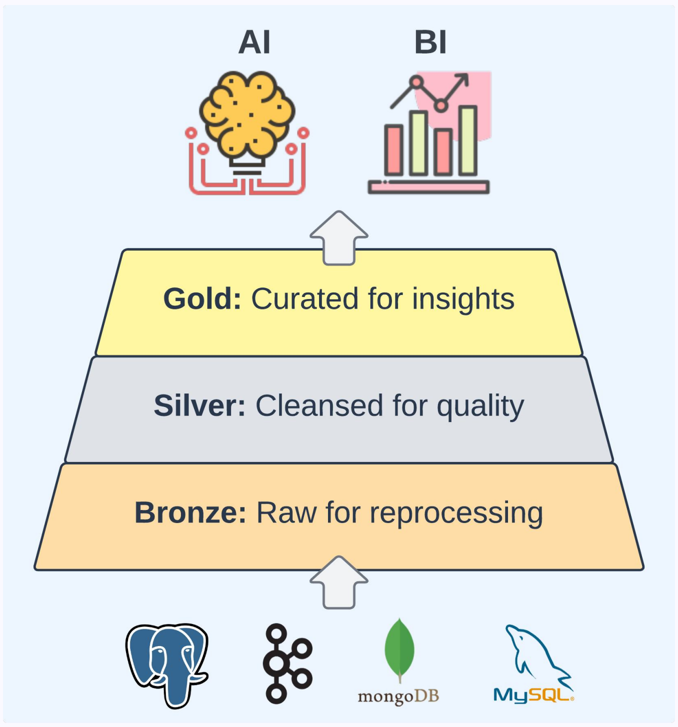

AI
BI
Gold: Curated for insights
Silver: Cleansed for quality
Bronze: Raw for reprocessing
mongoDB
MySQL

Figure 8-1: The medallion architecture that processes application data and empowers AI & BI

In the next sections, we'll discuss how Hudi achieves incremental processing, which is well-suited to supporting a robust implementation of the medallion architecture.

Incremental Query

Hudi effectively tracks changes in the form of transaction logs by persisting commit metadata within the timeline, thereby facilitating incremental processing - which, in most cases, relies on timestamp-based checkpointing. Hudi's incremental query feature is enabled through the following configurations:

```
hoodie.datasource.query.type=incremental
hoodie.datasource.read.begin.instanttime=202305150000
hoodie.datasource.read.end.instanttime=202305160000 # optional
```

These configurations allow for the retrieval of data that has changed within a defined time window. For more usage examples, see the docs. A few things to note about the behaviors:

- Setting **hoodie.datasource.read.begin.instanttime=0** effectively requests all changes made to the table from the very beginning of its history.
- Omitting **hoodie.datasource.read.end.instanttime** results in fetching the changes up to the most recent completed commit in the table.
- The data returned by incremental queries contains records that were updated during the specified time window. These records are matched to their versions corresponding to the latest completed commit in the table. If **hoodie.datasource.read.end.instanttime** is set, the records will align with the commit denoted by this specified end time.
- When the beginning time is set to zero and the end time is omitted, the incremental query effectively becomes equivalent to a snapshot query, retrieving all the most recent records in the table.

Now that we have an understanding of the behavior of incremental queries, we can delve into the details. Figure 8-2 shows the workflow involved in fetching incremental data from a Hudi MoR table.

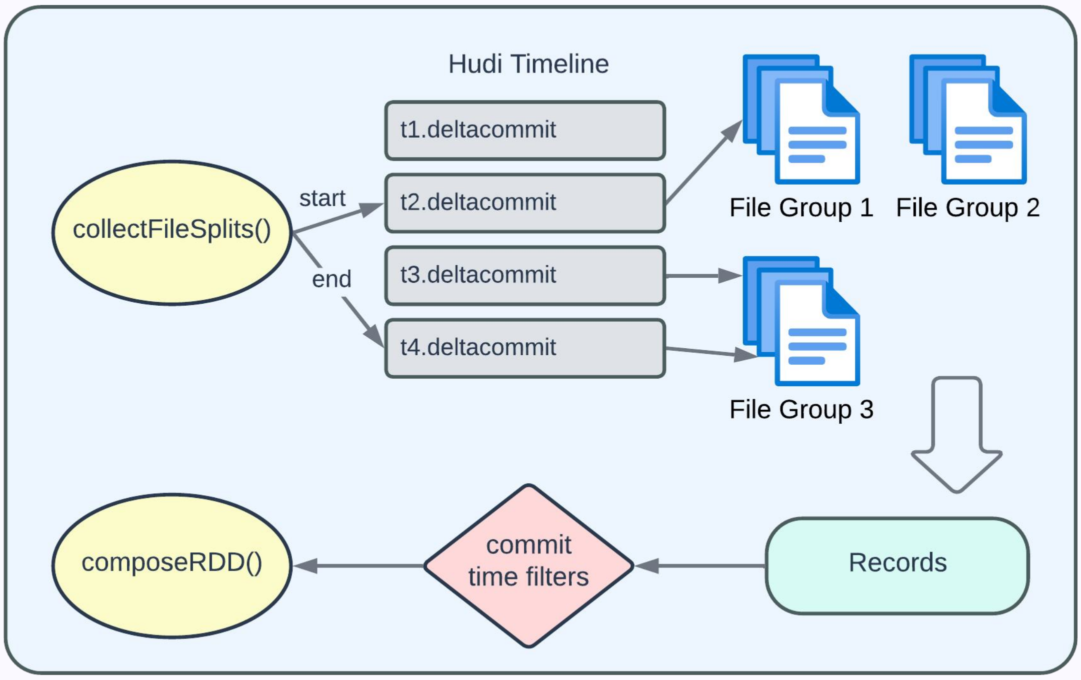


Figure 8-2: Fetching incremental data from a Hudi MoR table

Incremental queries follow the read flow as depicted in Chapter 2, implementing two internal APIs, **collectFileSplits()** and **composeRDD()**. The implementation consists of these steps:

- **collectFileSplits()** is responsible for identifying all files relevant to the query. This function derives start and end timestamps, based on user input, to define a specific time range. This time range is then used to filter commits in the timeline.
- Hudi's timeline, comprising a series of transaction logs, represents the changes made over time. With a specified time range, it becomes straightforward to filter down to the relevant files needed for the **composeRDD()** function to process.
- In a Hudi table, each record includes the field **_hoodie_commit_time**, which links the record to a specific commit in the timeline. During the process of loading target files for records, incremental queries construct a commit time filter to further minimize the amount of data read. This filter is pushed to the level of file reading, allowing **composeRDD()** to be optimized to load only those records that are intended to be returned.

Change Data Capture

Incremental queries effectively reveal which records have been changed, as well as their final states. However, they don't provide specific details about the nature of these changes. For instance, if record X is identified as having been modified, the incremental query doesn't clarify its column values prior to the update, or whether it was a newly inserted record. Additionally, it doesn't indicate if any records were hard-deleted. To address these limitations, Hudi 0.13.0 introduced *CDC*. This enhanced format of incremental processing provides a more comprehensive view of data modifications, including inserts, updates, and deletes, thereby enabling a clearer understanding of the changes within the dataset.

To enable the CDC functionality, set **hoodie.table.cdc.enabled=true**. Writers writing to the table will honor this setting and activate the process of creating CDC log files alongside base files. Thanks to Hudi's file grouping mechanism, these CDC log files are included in the same file groups that hold the changed data. This makes it easy to extend table services like cleaning, facilitate recovery operations like restore, and manage both CDC log files and data files together for easier file management.

To pull the CDC data, users simply need to set the incremental format to CDC when performing incremental queries. Time-range related behaviors still apply to the CDC query format. These are the necessary configurations:

```
hoodie.datasource.query.type=incremental
hoodie.datasource.query.incremental.format=cdc
hoodie.datasource.read.begin.instanttime=202305150000
hoodie.datasource.read.end.instanttime=202305160000 # optional
```

Figure 8-3 provides an overview of how writers and readers interact with CDC files and data.

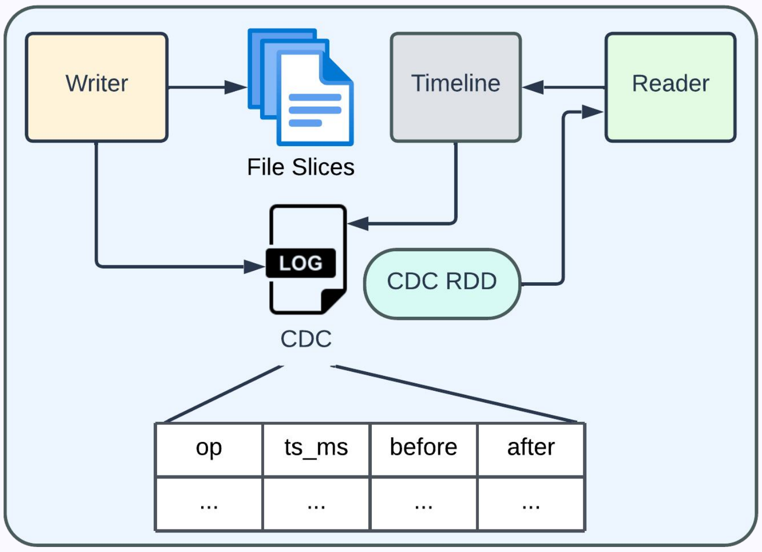


Figure 8-3: How Hudi writers and readers interact with CDC files and data

The general steps depicted in the diagram are described below:

1. On the writer side, Hudi's write handle holds the information about the intended operations for the writing records (**INSERT**, **UPDATE**, or **DELETE**).
2. This information is then encoded into a specific CDC log file format, containing four fields, as shown in the diagram. The nullable **before** and **after** fields store the complete record snapshot before and after the change. Users have the flexibility to reduce the volume of logged data by adjusting **hoodie.table.cdc.supplemental.logging.mode**. Specifically, use **DATA_BEFORE** to skip the **after** field, or set **OP_KEY_ONLY** to store record keys instead of **before** and **after** fields.
3. On the reader side, CDC log files are loaded to construct the results, following a process similar to that of normal incremental queries, whose incremental format is called **latest_state**. If both **before** and **after** fields are logged, the results will be directly extracted from the CDC log files. If a less verbose logging mode is used, the results will be computed on the fly by looking up existing records in the table. This is essentially a trade-off between saving storage space and the efficiency of running CDC queries.

Richer Insights

The introduction of CDC greatly enhances Hudi's capabilities, supporting a wider range of scenarios and offering valuable insights. For example, consider an account balance subject to frequent debit and credit transactions. Without CDC, periodic snapshot queries or the **latest_state** incremental queries might miss critical fluctuations. With CDC queries, all account changes are revealed, offering a comprehensive view of the account's activities. As such, this level of detail is essential to fraud detection algorithms.

Recap

In this chapter, we provided a concise introduction to incremental processing and to the medallion architecture, followed by an in-depth exploration of Hudi's approach to incremental queries and CDC, and their significance in deriving valuable business insights.

Chapter 9:
Hudi Streamer, a Swiss Army Knife for Ingestion

Over the course of the last eight chapters, we've explored a variety of Hudi topics related to internal mechanisms, including storage layout, read and write operations, indexing, table services, and concurrency control mechanisms. We'll now shift our focus to *Hudi Streamer*, a comprehensive data ingestion tool designed for deploying production-grade pipelines for Hudi tables. Given its versatility, a topic we'll delve into further within this chapter, Hudi Streamer is frequently described as a "Swiss Army Knife" for the importation of data into lakehouses.

Overview

Hudi Streamer is a Spark application designed to offer a wide range of customizable interfaces for managing the write process to Hudi tables. It enables users to configure source data, define schemas, schedule table services, keep data catalogs in sync, and so on. Figure 9-1 illustrates Hudi Streamer's workflow and components as part of an ingestion pipeline.

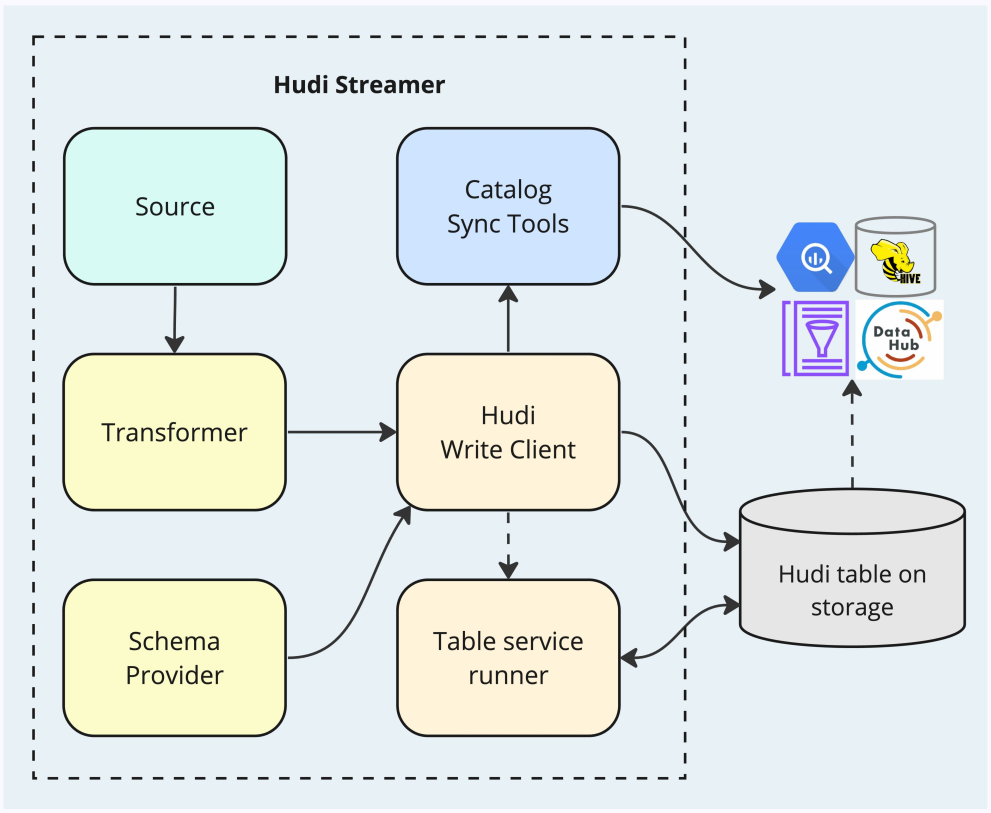


Figure 9-1: Hudi Streamer's workflow and components as part of an ingestion pipeline

Typically, ingestion pipelines are configured step-by-step by users. The setup process can be made much simpler by using the rich set of options that Hudi Streamer makes available. The key to mastering this tool is to understand what the options are meant for and how to configure each option properly. Here's an overview of various commonly-used options:

- The **--table-type** (CoW or MoR), **--table-name** (for identifying the table), and **--target-base-path** (physical location of the table) are three required properties for writing to a Hudi table.
- The **--continuous** flag indicates whether Hudi Streamer should operate in an ongoing manner or execute one step at a time. If the flag is present, the application will keep fetching source data and writing to storage in a loop ("continuous" mode), which is ideal when there is a steady stream of data from upstream sources. Without the flag, Hudi Streamer performs one-time data fetching and writing before terminating ("run-once" mode), which is ideal for batch or bootstrap (creating a new Hudi table using data from existing storage) use cases.
- The **--min-sync-interval-seconds** flag works with "continuous" mode, specifying the shortest allowable interval, in seconds, between ingestion cycles. For instance, if an ingestion operation requires 40 seconds to complete, and the min-sync-interval is configured to 60 seconds, Hudi Streamer will pause for 20 seconds before initiating the subsequent ingestion cycle. This pause ensures that the interval adheres to the minimum set duration. Conversely, if the ingestion duration extends to 70 seconds, surpassing the minimum interval, the application immediately proceeds to the next cycle without delay. This functionality is crucial for ensuring that sufficient data accumulates at the upstream source for processing, thereby preventing the inefficiency of handling numerous small-scale writes.
- The **--op** flag represents the type of operation to be executed by Hudi Streamer, which serves as another Hudi writer. This flag supports **UPSERT** (default), **INSERT**, and **BULK_INSERT**. For an overview of write operations, please revisit Chapter 3.
- The **--filter-dupes** flag corresponds to the write client configuration **hoodie.combine.before.insert=false|true**. This setting allows users to pre-combine records by keys within the incoming batch, effectively reducing the amount of data to process. The flag is applicable when the write operation is set to **INSERT** or **BULK_INSERT**. However, it should not be present when **--op** is set to **UPSERT**, to avoid losing potential updates before merging them with on-storage versions.

- The **--props** and **--hoodie-conf** flags offer flexible ways to ingest arbitrary Hudi properties. The former flag points to a file containing a collection of properties, and the latter accepts a single configuration in the format of **key=value**. Properties specified via **--hoodie-conf** take precedence over those provided via **--props**.

In the following sections, we'll delve into the major components depicted in the workflow diagram.

Source

Source is an abstraction for providing upstream source data for Hudi Streamer. Its primary responsibility is the fetching of data from the source system as an input batch for processing and writing. By extending the **Source** abstract class, Hudi Streamer can seamlessly integrate with a wide range of data systems. Designed with a platform vision from day one, Hudi currently offers more than a dozen off-the-shelf **Source** integrations, as shown in Figure 9-2.

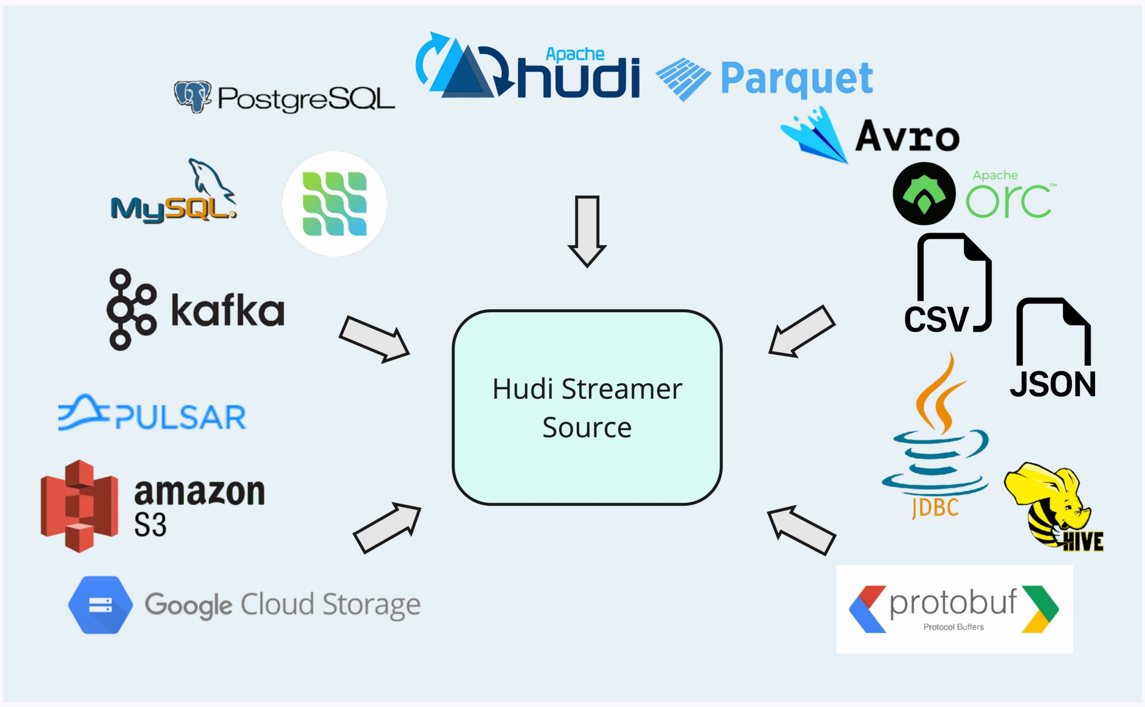


Figure 9-2: Hudi Streamer source integrations

To use a source for Hudi Streamer, set **--source-class** to the fully qualified class name of the source, and configure source-specific properties where applicable. For example, suppose you wish to use **KafkaSource** as the Hudi Streamer source. In this case, **hoodie.streamer.source.kafka.topic** is the fully qualified name. Additionally, setting a **--source-limit** for the **KafkaSource** sets an upper limit on the data amount to read during each fetch, enhancing control over the ingestion process. For more information, see the Hudi Streamer Source Configs docs.

Transformer

Upon retrieving incoming data from the source, the transformer interface performs lightweight transformations, such as adding or dropping specific columns or flattening the schema. The transformer processes a Spark dataset and outputs the transformed version of the dataset, enabling seamless data manipulation to meet the requirements of the ingestion pipeline.

The **--transformer-class** option takes in one or many class names of transformer implementations. When multiple transformers are given, they are applied sequentially. In other words, the output of one transformer serves as the input for the next. This chained approach provides flexibility and facilitates code maintenance.

Table Service Runner

Table services, as described in Chapter 5, can be managed by Hudi Streamer alongside data writing operations. When configured as **async**, compaction and clustering will be scheduled inline by the Hudi write client internal to Hudi Streamer and will be executed asynchronously by **HoodieAsyncTableService**, which uses a thread pool to submit and control table service jobs.

While async table service jobs are running, it might not always be desirable to write new data. For instance, the same cluster that is executing the table services may not have enough resources to perform ingestion. Furthermore, it's advisable to avoid running too many concurrent compaction or clustering jobs to prevent resource contention. Use **--max-pending-compactions** and **--max-pending-clustering** to limit the outstanding table service operations; when the limits are reached, no new ingestion job will be scheduled until at least one has been completed.

When running ingestion jobs and table service jobs concurrently within the same Spark application, it's crucial to appropriately allocate the cluster's resources to ensure optimal performance and efficiency. Hudi Streamer facilitates this by enabling users to input scheduling configurations through specific options. These configurations play a key role in managing how resources are distributed between the ingestion, compaction, and clustering processes.

For ingestion:

- **--delta-sync-scheduling-weight**
- **--delta-sync-scheduling-minshare**

For compaction:

- **--compact-scheduling-weight**
- **--compact-scheduling-minshare**

For clustering:

- **--cluster-scheduling-weight**
- **--cluster-scheduling-minshare**

The properties shown will be used to generate an XML file, which is then referenced by the Spark property **spark.scheduler.allocation.file**. To activate these settings, users should set **spark.scheduler.mode=FAIR** for the Spark application. For further explanation on the scheduling mechanism, please consult the Spark docs.

Catalog Sync Tools

Data catalogs play a crucial role in the data ecosystem, and Hudi supports multi-catalog sync via the ***SyncTool*** *classes*. Hudi Streamer can integrate with **SyncTools** through the **--sync-tool-classes** option, which takes in a list of **SyncTool** class names. Here are the various class names:

```
# for AWS Glue Data Catalog
org.apache.hudi.aws.sync.AwsGlueCatalogSyncTool

# for Google BigQuery
org.apache.hudi.gcp.bigquery.BigQuerySyncTool

# for Hive Metastore
org.apache.hudi.hive.HiveSyncTool

# for DataHub
org.apache.hudi.sync.datahub.DataHubSyncTool
```

After each write, if the catalog sync is enabled using the **--enable-sync** flag, each of the configured **SyncTools** will run synchronously in sequence to upload metadata to the target data catalog. For example, if the write operation created some new partitions and added a new column to the table, the **AwsGlueCatalogSyncTool** will update the partition list and the schema stored in the catalog table.

For **SyncTools** to function properly, users should supply additional properties through the **--props** or **--hoodie-conf** options. For configuration details, see the docs.

Schema Provider

The *schema provider*, specified through **--schemaprovider-class**, serves the schema for reading from the source and writing to the target table. A notable implementation of this is the **SchemaRegistryProvider**, which enables Hudi Streamer to access Kafka's schema registry, ensuring that data ingested from Kafka is accurately interpreted and processed.

Additional Features

In addition to the features described in previous sections, Hudi Streamer supports other useful features, such as the following:

- The **--checkpoint** and **--initial-checkpoint-provider** flags support pausing and resuming data fetching from the **Source**, avoiding data loss or duplication.
- The **--post-write-termination-strategy-class** flag allows for a graceful shutdown of Hudi Streamer running in "continuous" mode.
- The **--run-bootstrap** flag instructs the Hudi Streamer to perform a one-time bootstrap operation for a new Hudi table.

Recap

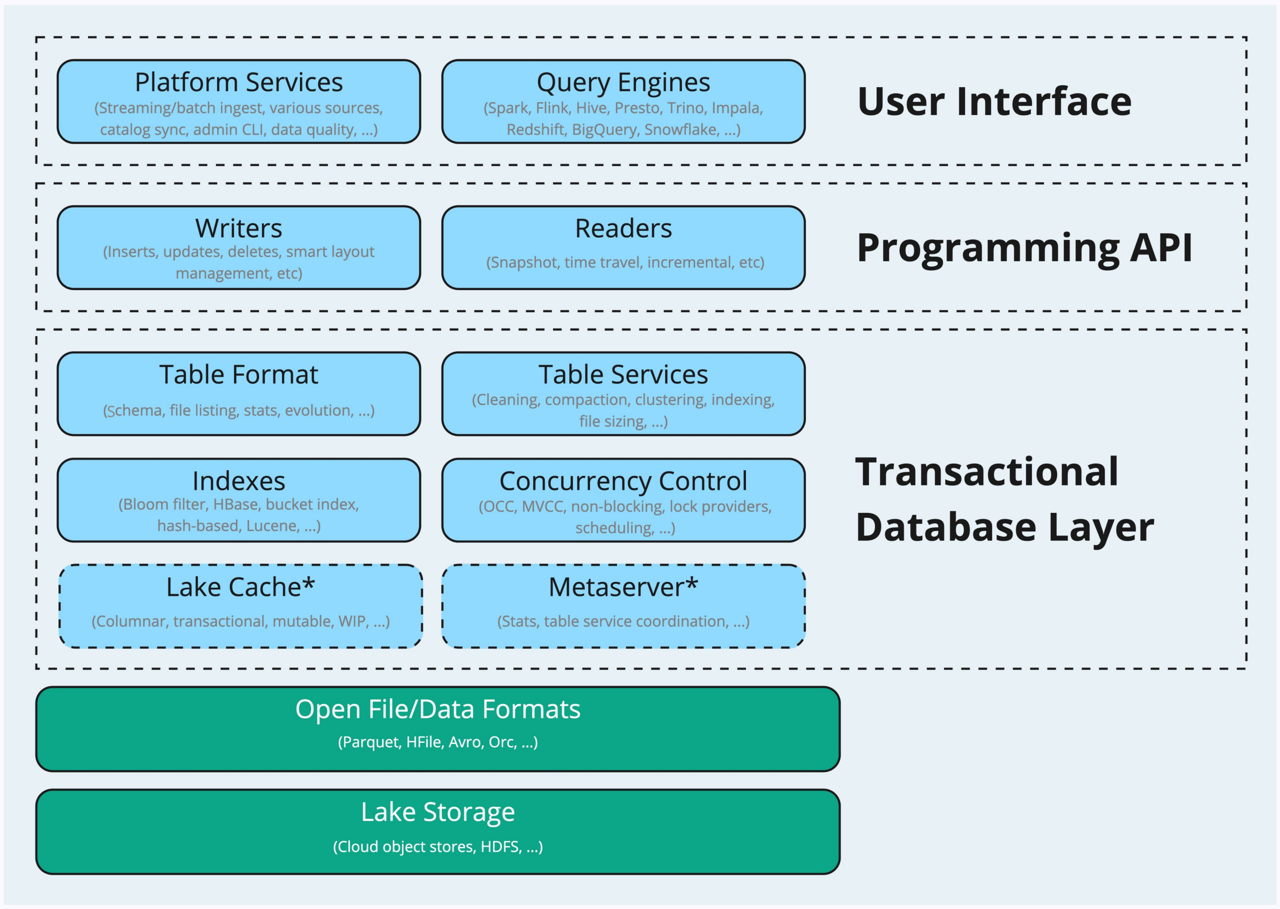
In this chapter, we explored the major components of the Hudi Streamer workflow and its diverse options.

Chapter 10:
Becoming "One" - The Upcoming 1.0 Highlights

Throughout the book, we've explored Hudi concepts relevant to 0.x versions. In this final chapter, we'll explore the future of Hudi and delve into the exciting new features in the upcoming 1.0 release, highlighting enhancements - such as the LSM tree timeline and the functional index - which significantly improve data lakehouse management and efficiency, providing a more database-like experience on the data lake.

Recap: The Hudi Stack

Before we discuss Hudi 1.0, we'll review the Hudi stack, as shown in Figure 10-1 - a framework that remains unchanged across 0.x and 1.x versions. Recall that the Hudi stack sits on top of storage systems, executing read and write operations against open file formats. It is structured into three layers: the transactional database, the programming API, and the user interface.



Platform Services
(Streaming/batch ingest, various sources, catalog sync, admin CLI, data quality, ...)
Query Engines
(Spark, Flink, Hive, Presto, Trino, Impala, Redshift, BigQuery, Snowflake, ...)
User Interface
Writers
(Inserts, updates, deletes, smart layout management, etc)
Readers
(Snapshot, time travel, incremental, etc)
Programming API
Table Format
(Schema, file listing, stats, evolution, ...)
Table Services
(Cleaning, compaction, clustering, indexing, file sizing, ...)
Indexes
(Bloom filter, HBase, bucket index, hash-based, Lucene, ...)
Concurrency Control
(OCC, MVCC, non-blocking, lock providers, scheduling, ...)
Lake Cache*
(Columnar, transactional, mutable, WIP, ...)
Metaserver*
(Stats, table service coordination, ...)
Transactional Database Layer
Open File/Data Formats
(Parquet, HFile, Avro, Orc, ...)
Lake Storage
(Cloud object stores, HDFS, ...)

Figure 10-1: A review of the Hudi stack

The transactional database layer is the core of Hudi. It has several key components, which work together to create a database-like experience with Hudi lakehouses:

- The table format, which defines the storage layout
- Table services, which keep the table optimized
- Indexes, which speed up reads and writes
- Concurrency control, which supports the isolation principle
- The lake cache, which elevates read efficiency
- The metaserver, which centralizes metadata access

The programming API layer introduces writer and reader interfaces, standardizing integration with other execution and query engines such as Spark. These APIs empower users to fully harness Hudi's advanced capabilities, such as efficient upserts and incremental processing.

The final layer, the user interface, provides robust tooling for adopting Hudi and building comprehensive lakehouse solutions. This tooling can be broken into two categories:

- Platform services, which include ingestion utilities, catalog sync tools, and admin CLI
- Query engines such as Spark, Flink, Presto, and Trino

With Hudi's architecture in mind, we can now begin our discussion of Hudi 1.0 and the new features it provides.

Release 1.0 Highlights

While the overall Hudi stack remains unchanged in version 1.0, the new release features redesigns and updates at the table-format level. These changes enhance overall efficiency and throughput, significantly upgrading Hudi's lakehouse capabilities.

LSM Tree Timeline

Recall that the Hudi timeline consists of a series of immutable transaction logs that record all changes made to a table. In 0.x versions, the volume of transaction logs in the timeline increases linearly with time, so older logs are archived to optimize storage use. However, this optimization comes with a tradeoff, which is increased compute cost when accessing archived logs. In Hudi 1.0, a key design goal is to support a nearly infinite timeline that balances optimized storage with efficient access. To achieve this goal, a *log-structured merge-tree (LSM tree)*, a multi-layered data structure designed for high write throughput, is used to define the timeline layout for tables in Hudi 1.0, as shown in Figure 10-2.

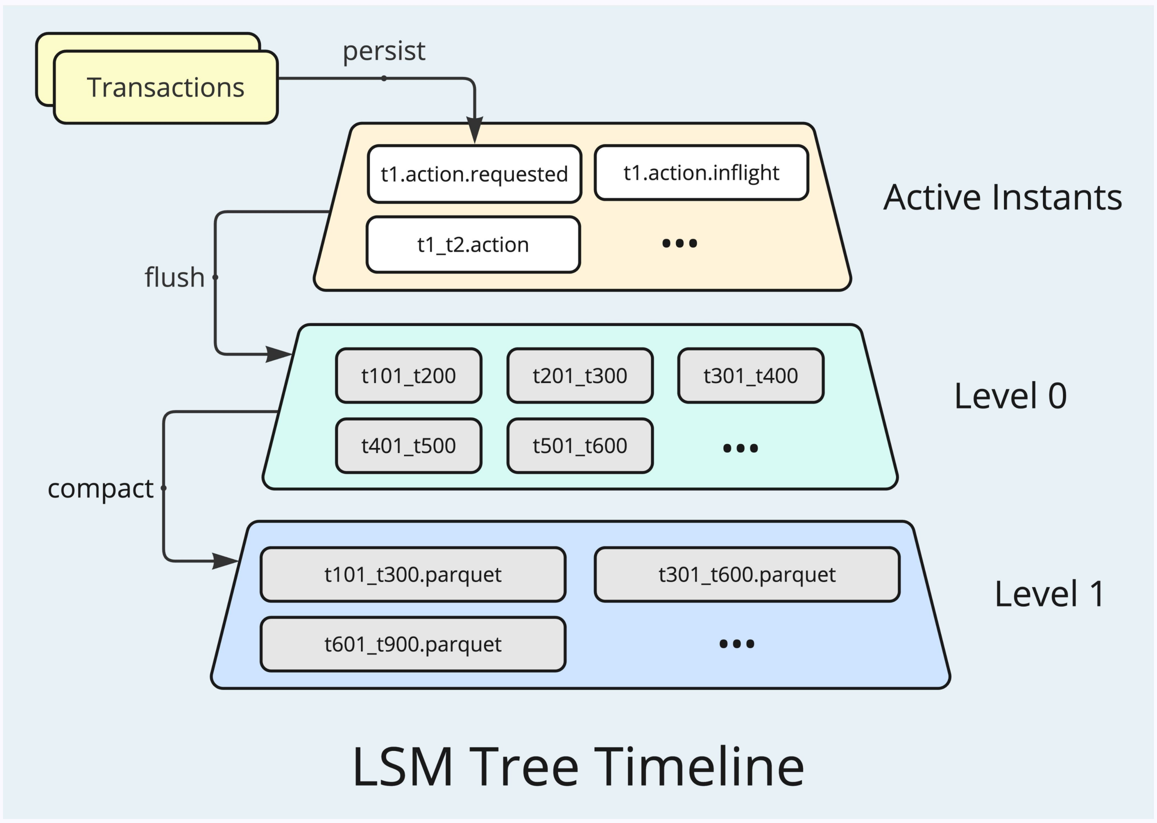


Figure 10-2: The LSM tree structure for Hudi timeline layout and its data flows

In the LSM tree, the top layer stores transactions as active instants in Avro files. These capture different states of each transaction, such as requested, in progress, and completed. Once the total number of Avro files reaches a set threshold, the files are combined and saved in Parquet format. These Parquet files are then sorted chronologically and given file names that include time-range data, making it easier to access the files quickly through manifest files. If the number of these Parquet files grows too large, they are further compressed into even larger Parquet files. This compression makes these files highly efficient for queries, especially when filtering by time range or specific columns, leading to overall quicker data access while still optimizing for storage.

Non-Blocking Concurrency Control

In 0.x versions, when a streaming writer is present in a concurrent writing scenario, contention can arise due to, for example, random updates incurred by running a separate GDPR delete job. To address this, Hudi 0.x versions support OCC and MVCC. In an example scenario, using OCC could lead to repeated retries, wasting precious compute resources. To avoid this problem, Hudi 0.x versions provide early conflict detection for OCC. MVCC is also available to prevent blocking and retry behaviors due to contention between streaming writers and table service runners. In 1.0, Hudi introduces *non-blocking concurrency control (NBCC)*, as shown in Figure 10-3, to maximize writer throughput for MoR tables.

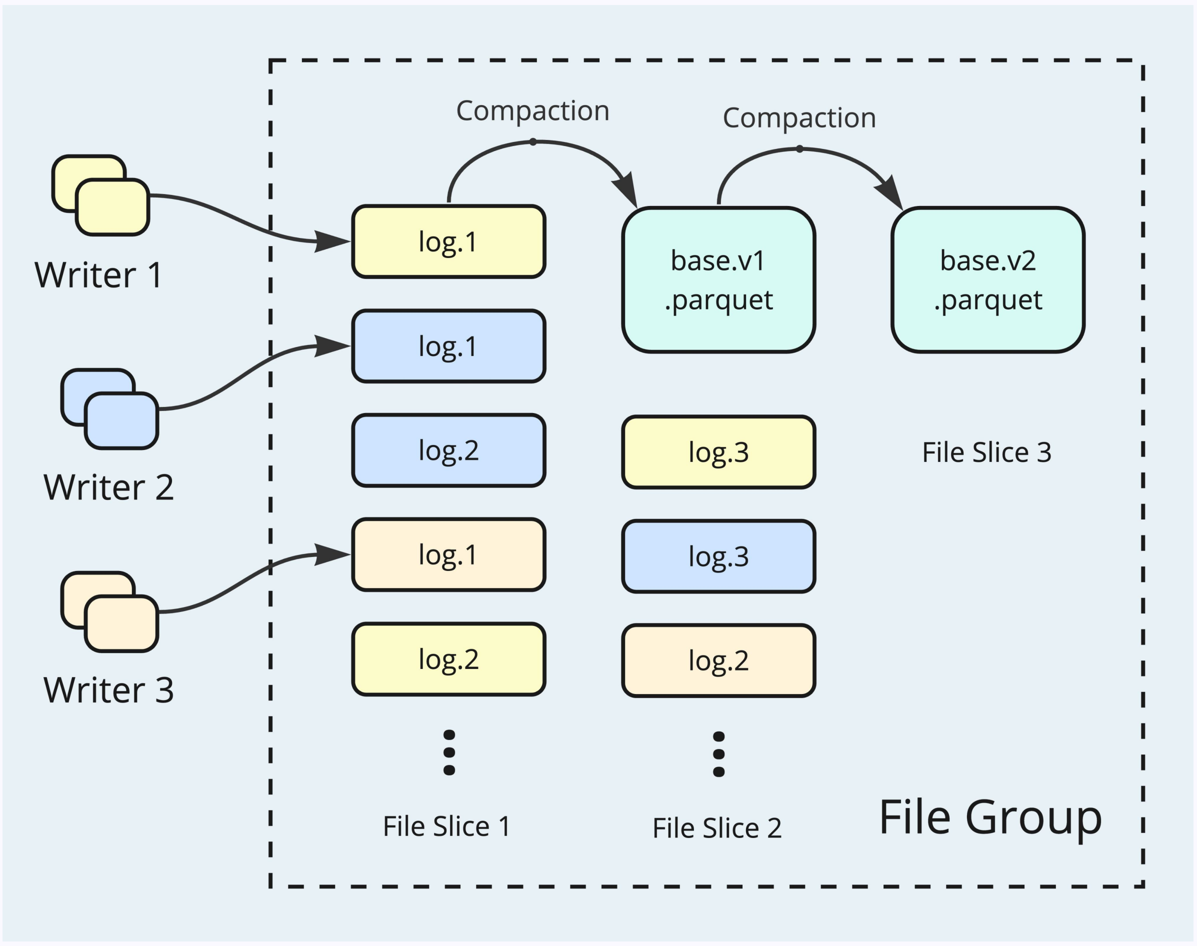


Figure 10-3: The NBCC workflow

NBCC allows multiple writers to update log files to the same file slice, and it defers the conflict resolution to the compaction stage. Unlike 0.x versions, log files in 1.0 record the completion time of each commit, as well as the starting time. This enables proper sorting for log files and helps to determine file slice boundaries. Merging semantics are applied to the updated records during compaction, based on a configurable ordering field. To resolve the clock skew issue, **TrueTimeGenerator** is implemented, ensuring all writer commits have monotonically increasing timestamps.

File Group Reader & Writer

Since the first version, Hudi has used record keys, a design choice that unlocks significant potential for operations at the record level. Paired with the file group model, this approach lays the foundation for efficient upserts and look-ups. Hudi 1.0 builds on the design advantages offered by record keys and file groups with the introduction of file group reader and writer APIs, shown in Figure 10-4.

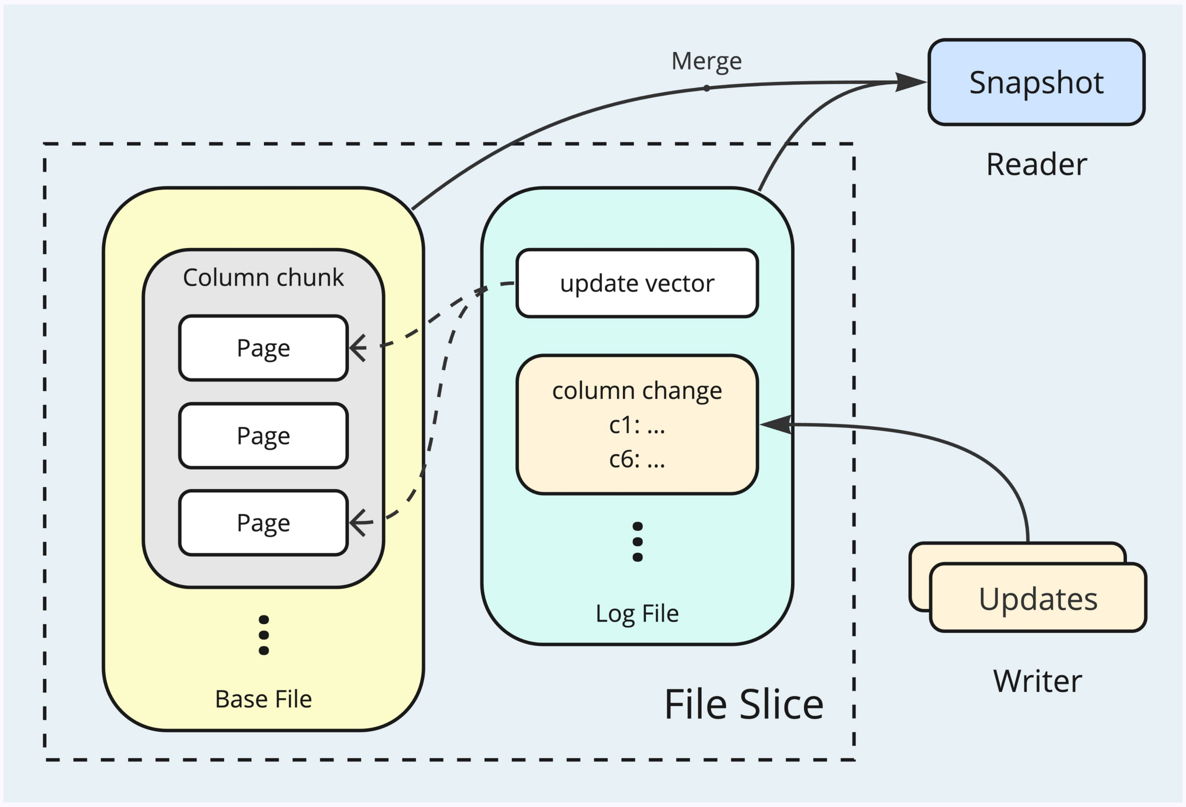


Figure 10-4: The file group reader and writer API workflow

Hudi 1.0 file group writer APIs use partial updates to reduce the log file sizes, involving only updated columns and values. By taking advantage of Hudi's indexing systems, targeted updates are efficiently located and positional information is encoded alongside the data log blocks. File group reader APIs have access to minimized log file data and positional information to pinpoint the updating rows and columns, allowing optimization of a snapshot query against an un-compacted file slice.

Functional Index

In the 0.x versions, Hudi supports a variety of indexing capabilities, including the bucket index and record-level index. To enhance flexibility and improve access speeds, Hudi 1.0 introduces the functional index, shown in Figure 10-5, enabling faster retrieval methods and incorporating partitioning schemes into the indexing system.

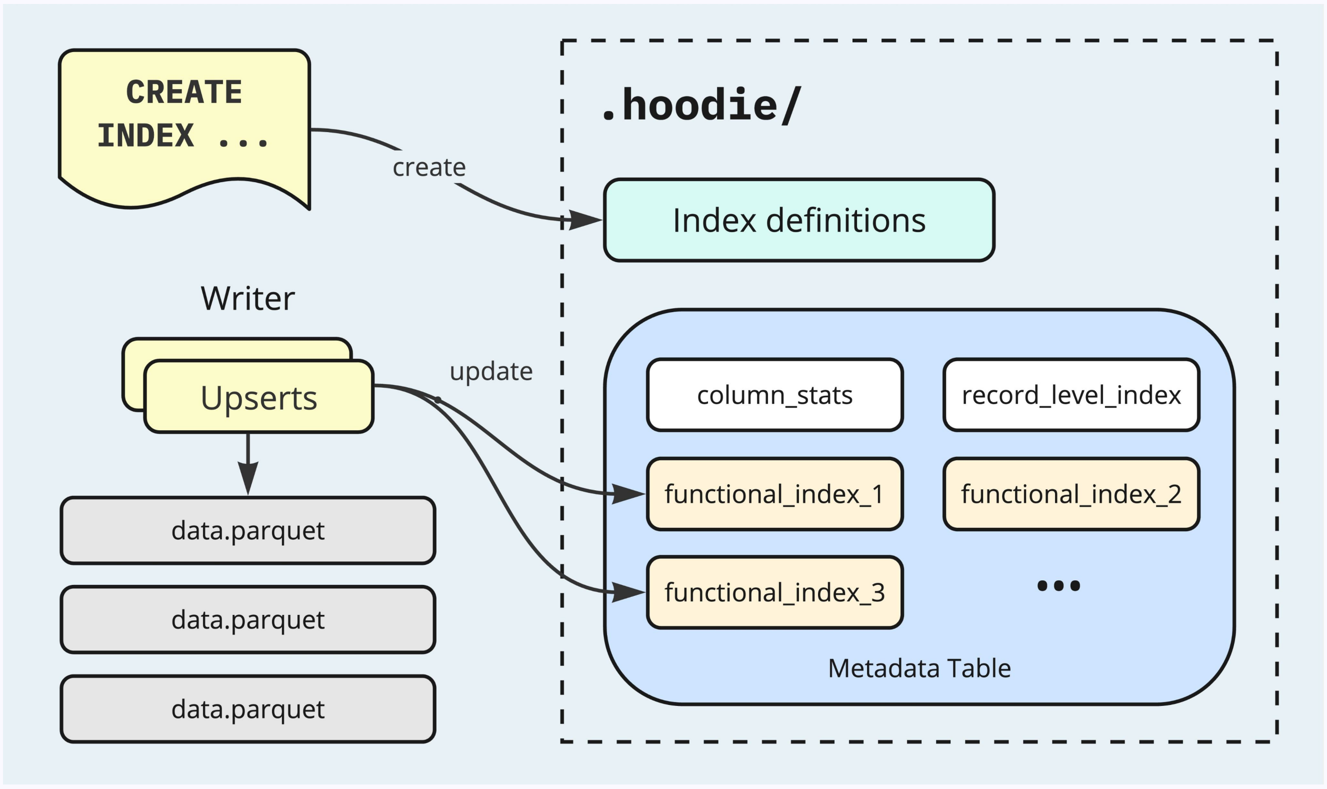


Figure 10-5: The flow for using Hudi functional index

We'll illustrate the benefits of the functional index with an example. Consider a column **ts** that holds epoch timestamps. Users may want to filter the data by different time precisions, such as by month or by hour. By building a functional index on the **ts** column using the following SQL, effective data-skipping is possible without the need to physically partition the table by hour, or to add a separate **hour** column.

```
CREATE INDEX ts_hour ON hudi_table USING column_stats(ts)
options(func='hour');
```

Hudi stores user-created index definitions in a dedicated directory under the **.hoodie/** metadata path. These definitions inform query engines of the available indexes, supporting more optimized query planning. The index entries are maintained under separate partitions within the metadata table, which serves as the indexing subsystem for the enclosing Hudi table. When writers commit changes, all available functional indexes are updated to reflect these changes, similar to other enabled indexing features in the metadata table. This ensures read and write efficiency and up-to-date indexes.

Recap

In this final chapter, we reviewed the Hudi stack and introduced noteworthy features set to debut in the upcoming Hudi 1.0 release: the LSM tree timeline, NBCC, file group read and write APIs, and the functional index. The 1.0 RFC describes Hudi 1.0 as "*...a reimagination of Hudi, as the transactional database for the lake, with polyglot persistence, raising the level of abstraction and platformization even higher for Hudi data lakes*," marking an exciting evolution in Hudi's capabilities.

Conclusion

Over the course of this book, we discussed the core functionalities and features of Apache Hudi, from storage format to the complexities of handling concurrent operations and incremental processing. We then discussed the advanced features of Hudi, such as Hudi Streamer, demonstrating Hudi's versatility in managing data ingestion pipelines and showcasing its role as an integral tool for modern data architectures. The final chapter explored the upcoming Hudi version 1.0, describing its groundbreaking features, which promise to revolutionize data lakehouse architecture. Through practical insights and detailed examples, the book illustrated Apache Hudi's significance as an indispensable resource for data professionals aiming to take full advantage of the potential of data lakehouse solutions.