
ICEBERG
Apache
hudi

Challenges with traditional data lake storages?

Grow Data Skills

Data lakes, while revolutionary, brought several challenges in practice:

- **Lack of Schema Enforcement**
  - Traditional data lakes often store raw data in formats like CSV, JSON, or Parquet without enforcing a schema.
  - This leads to **data quality issues**, schema inconsistencies, and difficulties in integrating data.
- **Data Update Challenges**
  - Data lakes are inherently **append-only** systems. Updating or deleting data (e.g., GDPR compliance) is cumbersome and requires complex workflows.
  - Managing **Slowly Changing Dimensions (SCDs)** in traditional lakes is inefficient.
- **Lack of ACID Transactions**
  - Traditional data lakes lack **atomicity**, **consistency**, **isolation**, and **durability** (ACID properties).
  - Concurrent reads/writes lead to **dirty reads**, **partial updates**, or **data corruption**.
- **Query Performance Issues**
  - Large datasets require **full scans** for most queries, leading to slow performance.
  - Lack of **data indexing** further deteriorates query speed.
- **Data Management Complexity**
  - Managing metadata and tracking file locations manually or through third-party tools increases operational overhead.
  - Fragmented files **(small file problem)** lead to inefficient storage and degraded performance.

What is open table format?

Grow Data Skills

An **open table format** is a set of standards and technologies designed to manage large-scale datasets in data lakes by organizing and tracking files and their metadata as a logical “table.” These formats build on open file formats like **Parquet**, **ORC**, or **Avro**, adding **transactional capabilities**, **metadata management**, and **data versioning**.

Examples of open table formats include:

- **Apache Iceberg**
- **Apache Hudi**
- **Delta Lake**

These formats transform data lakes into **transactional data platforms**, enabling features like **schema evolution**, **time travel**, and **incremental updates**.

Challenges solved by open table formats?

Grow Data Skills

- Implements **ACID transactions**, ensuring reliable concurrent operations without data corruption.
- Enables **consistent reads and writes**, even in multi-user or distributed environments.
- Enables **row-level updates** and **deletes** without rewriting the entire dataset.
- Uses **merge-on-read** or **copy-on-write** mechanisms for efficient data modifications.
- Maintain **rich metadata layers** that track data files, versions, and partitions directly within the system.
- Improve query performance by minimizing file scans through metadata indexing.
- Support **schema evolution**, allowing changes like adding/removing fields without breaking queries.
- Enforce **schema validation** to ensure data quality and compatibility.
- Use **partition pruning**, **file pruning**, and **metadata caching** to avoid unnecessary scans.
- Optimize storage by compacting small files and leveraging **indexing**.
- Enable **time travel**, allowing users to query data at specific points in time.
- Maintain **data versioning**, making it easy to recover from errors or audit historical changes.
- Support **incremental processing**, so only changed or new data is read and processed.
- Provide **upsert** and **merge** operations for efficient data workflows.

What is small file problem?

Grow Data Skills

The **small file problem** arises when a data lake contains a large number of small files instead of fewer large, well-organized files. This issue is common in systems that ingest data incrementally or in real-time, leading to fragmented and inefficient storage.

- **Inefficient Query Performance**:
  - Query engines like Spark, Hive, or Presto must scan a large number of small files, increasing **I/O operations** and query runtime.
  - Metadata overhead increases as each file has associated metadata (e.g., file paths, schema).
- **High Storage Overhead**:
  - Each file, no matter how small, incurs additional storage metadata, leading to inefficient storage utilization.
- **Resource Wastage**:
  - Distributed processing frameworks like Spark schedule tasks per file. Too many small files result in numerous tasks, leading to **executor inefficiencies** and **increased resource usage**.
- **Data Fragmentation**:
  - Managing fragmented files across partitions or directories becomes cumbersome, especially in systems relying on consistent partition structures.

How open table formats solved small file problem?

Grow Data Skills

- **File Compaction**
  - Automatically merge small files into larger files during write operations or scheduled compaction jobs.
  - Types of Compaction:
    - **Delta Lake**: Background optimize commands for merging files.
    - **Hudi**: Offers merge-on-read for combining files during reads.
    - **Iceberg**: Supports rewrite data files to compact small files.
- **Metadata Management**
  - Maintain metadata layers that track file locations and sizes, reducing the need for full directory scans.
  - Efficient metadata querying enables file pruning, so only relevant files are accessed during queries.
- **Partition Optimization**
  - Dynamically reorganize partitions to ensure balanced file sizes across partitions.
  - **Iceberg**: Supports partition evolution, enabling adjustments to partitioning schemes without rewriting the entire table.
  - **Hudi**: Allows partition pruning for better management of file sizes in directories.
- **Data Skipping**
  - Use statistics-based filtering (e.g., min/max values, bloom filters) to avoid scanning unnecessary files.
  - Reduces the performance impact of small files by ensuring only relevant files are processed.

How open table formats solved small file problem?

Grow Data Skills

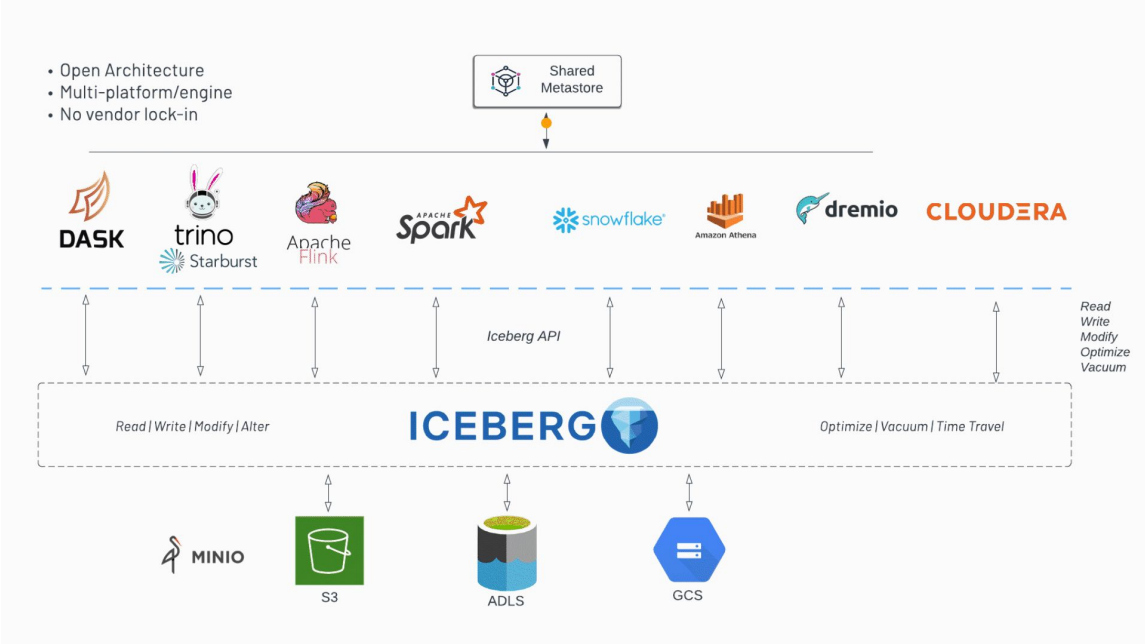
- **Automatic File Management**
  - Automatically detect and consolidate fragmented files during ingestion or query execution.
  - Periodic maintenance jobs (e.g., optimize in Delta Lake or compaction jobs in Hudi) keep file sizes under control.
- **Query Optimization**
  - Enable file pruning and lazy loading to process only required files instead of scanning all small files.
  - Improves query performance by bypassing irrelevant files.
- **Incremental Processing**
  - Write only incremental changes, reducing the creation of unnecessary files.
  - **Hudi**: Supports incremental writes using merge-on-read, minimizing small file creation.
- **ACID Transactions**
  - Ensure atomic writes to avoid creating partial files during concurrent operations.
  - Allows retries or rollbacks without leaving behind orphaned small files.
- **Efficient Data Writing**
  - Use advanced algorithms to determine optimal file sizes during write operations:
  - **Iceberg**: Writes data in large, contiguous files with support for customizable file size thresholds.
  - **Delta Lake**: Allows configurations for the desired file size during compaction.

Apache Iceberg

Grow Data Skills

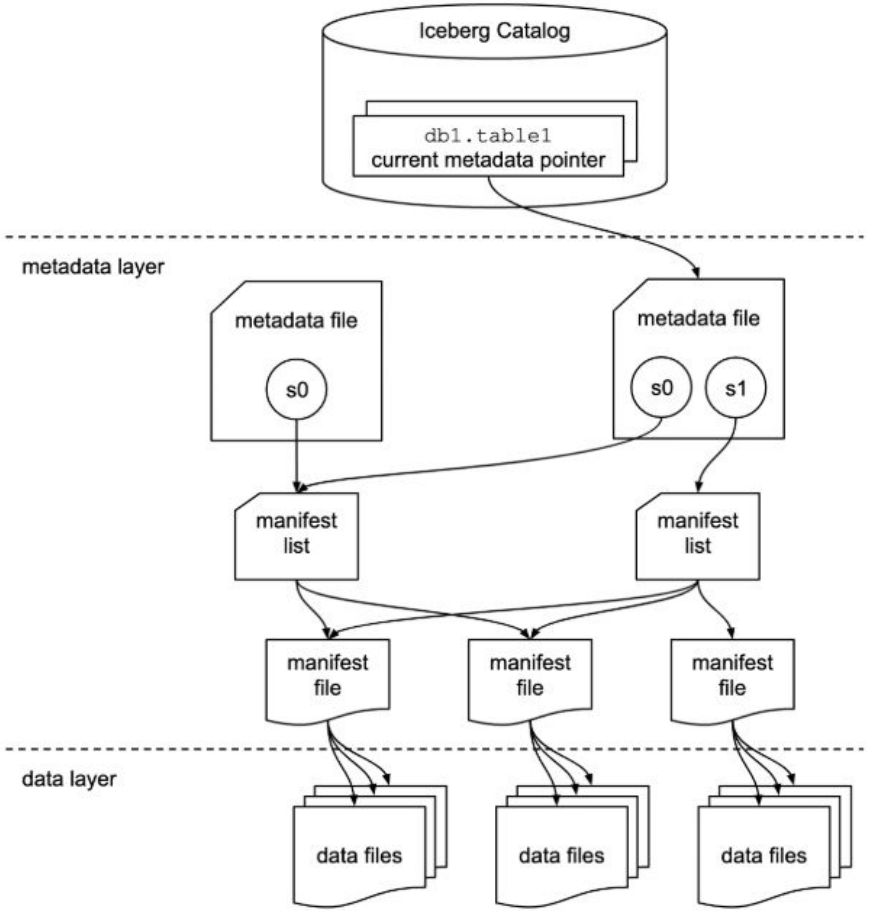
Apache Iceberg is an **open-source table** format designed to bring reliability, performance, and scalability to large-scale datasets in data lakes. It was developed by **Netflix** to address challenges in traditional data lake systems, such as lack of ACID transactions, inefficient metadata handling, and schema evolution issues.

Iceberg organizes data into a logical table abstraction while maintaining underlying files in open formats like **Parquet**, **ORC**, or Avro. It is optimized for analytics workloads and integrates seamlessly with modern data processing engines like Apache Spark, Trino, Hive, Flink, and more.



Apache Iceberg - Architecture

Grow Data Skills

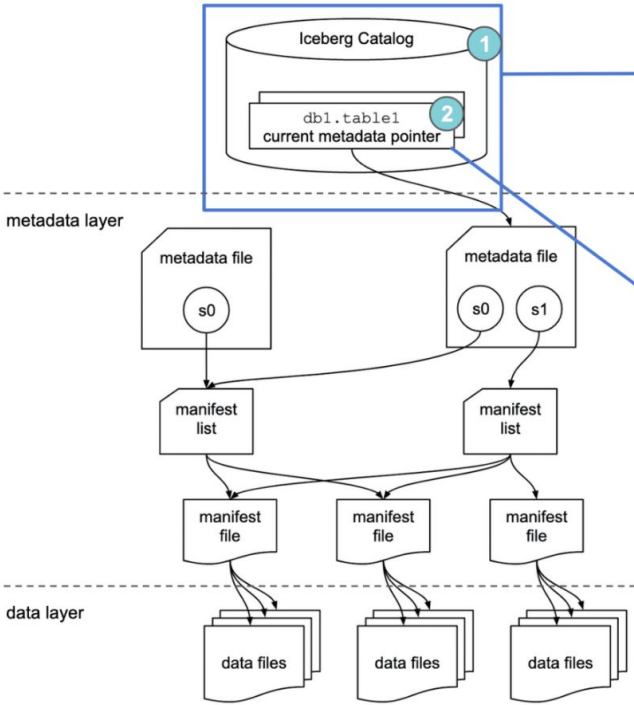


Apache Iceberg - Architecture

Grow Data Skills

There are 3 layers in the architecture of an Iceberg table:

- **The Iceberg catalog**
- **The metadata layer - *which contains metadata files, manifest lists, and manifest files***
- **The data layer**



Iceberg Catalog

- A store that houses the current metadata pointer for Iceberg tables
- Must support atomic operations for updating the current metadata pointer (e.g. HDFS, HMS, Nessie)

`table1`'s current metadata pointer

- Mapping of table name to the location of current metadata file

Apache Iceberg - Architecture

Grow Data Skills

- **Iceberg catalog** - Anyone reading from a table (let alone 10s, 100s, or 1,000s) needs to know where to go first — somewhere they can go to find out where to read/write data for a given table. The first step for anyone looking to read the table is to find the **location** of the **current metadata pointer** (note the term “current metadata pointer” is not an official term, but rather a descriptive term because there is no official term at this point and there hasn’t been push-back in the community on it).

  This central place where you go to find the current location of the current metadata pointer is the **Iceberg catalog**.

  The primary requirement for an Iceberg catalog is that it must support atomic operations for updating the current metadata pointer (e.g., HDFS, Hive Metastore, Nessie). This is what allows transactions on Iceberg tables to be atomic and provide correctness guarantees.

  Within the catalog, there is a reference or pointer for each table to that table’s current metadata file. For example, in the diagram shown, there are **2 metadata files**. The value for the table’s current metadata pointer in the catalog is the location of the metadata file on the right.

  What this data looks like is dependent on what Iceberg catalog is being used. Here is the example:

  - With **Hive metastore** as the catalog, the table entry in the metastore has a table property which stores the location of the current metadata file.

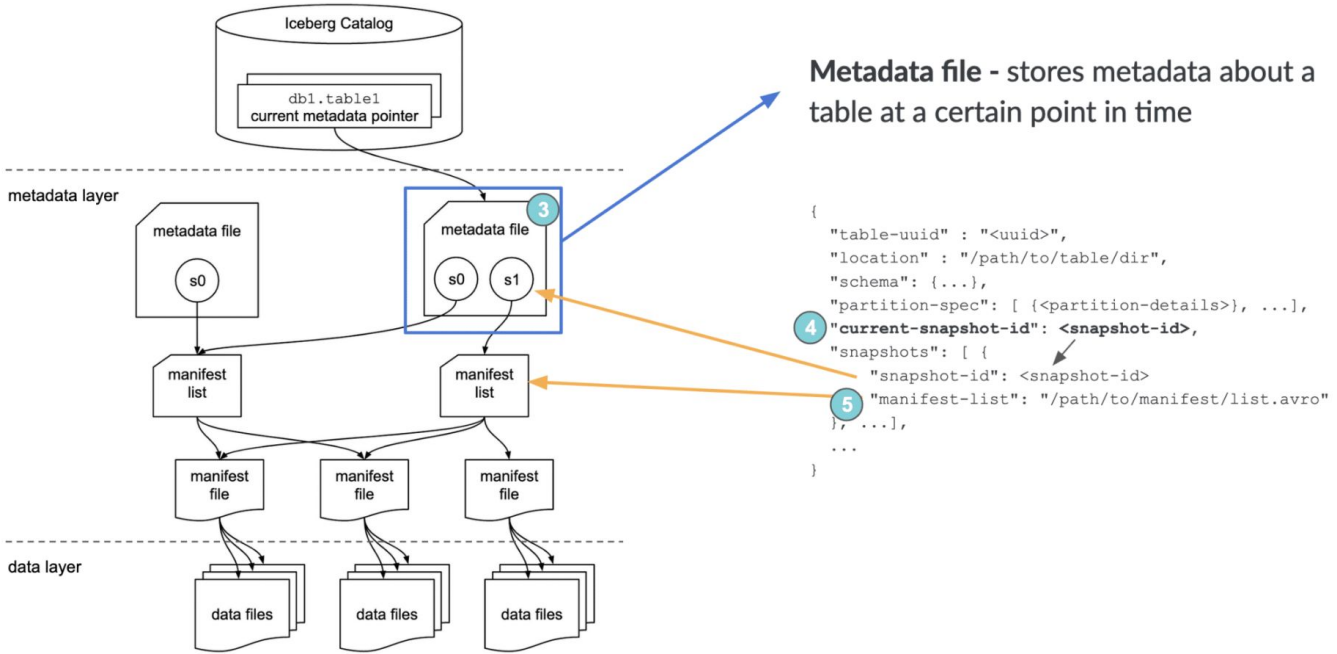
So, when a SELECT query is reading an Iceberg table, the query engine first goes to the Iceberg catalog, then retrieves the entry of the location of the current metadata file for the table it’s looking to read, then opens that file.

Apache Iceberg - Architecture

Grow Data Skills

- **Metadata File** - As the name implies, metadata files store metadata about a table. This includes information about the table's schema, partition information, snapshots, and which snapshot is the current one.

When a SELECT query is reading an Iceberg table and has its current metadata file open after getting its location from the table's entry in the catalog, the query engine then reads the value of current-snapshot-id. It then uses this value to find that snapshot's entry in the snapshots array, then retrieves the value of that snapshot's manifest-list entry, and opens the manifest list that location points to.

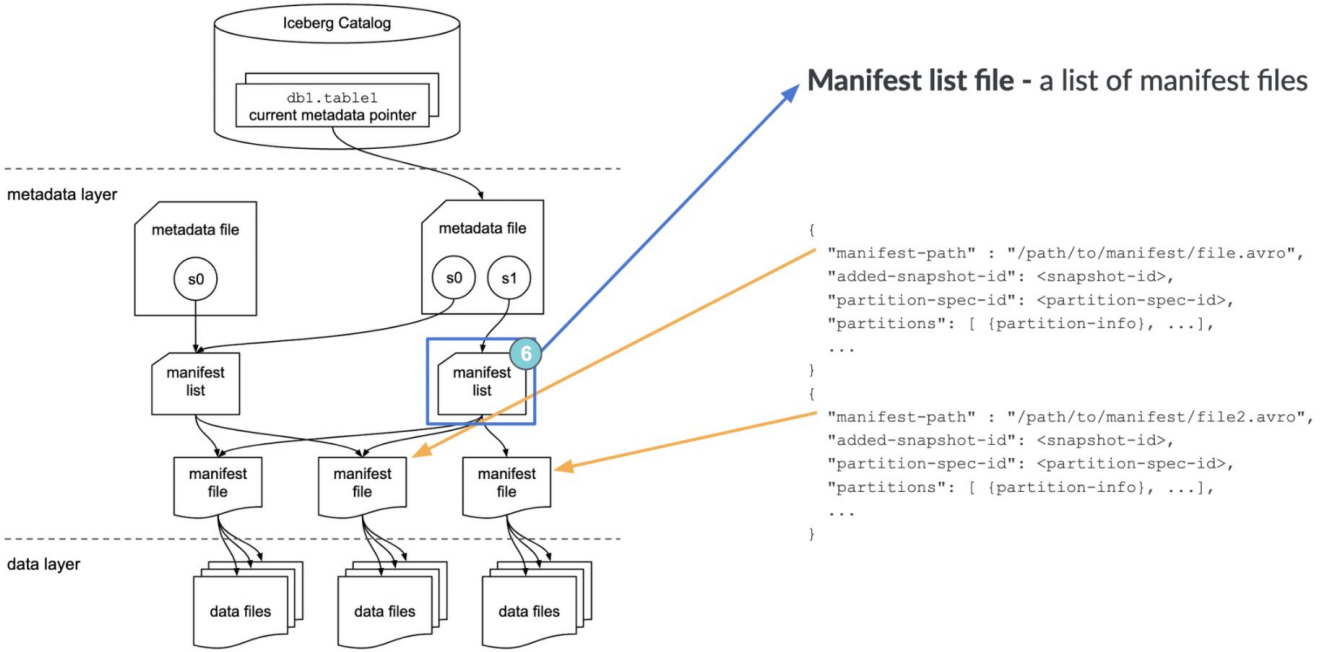


Apache Iceberg - Architecture

Grow Data Skills

- **Manifest List** - Another aptly named file, the manifest list is a list of manifest files. The manifest list has information about each manifest file that makes up that snapshot, such as the location of the manifest file, what snapshot it was added as part of, and information about the partitions it belongs to and the lower and upper bounds for partition columns for the data files it tracks.

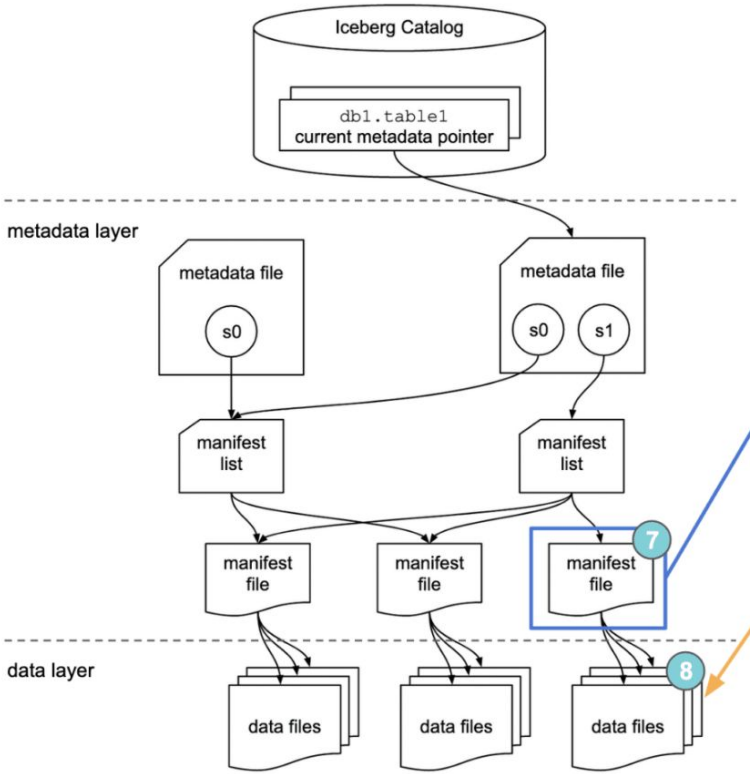
When a SELECT query is reading an Iceberg table and has the manifest list open for the snapshot after getting its location from the metadata file, the query engine then reads the value of the manifest-path entries, and opens the manifest files. It could also do some optimizations at this stage like using row counts or filtering of data using the partition information.



Apache Iceberg - Architecture

Grow Data Skills

- **Manifest File**



Manifest file - a list of data files, along with details and stats about each data file

```
{
  "data-file": {
    "file-path": "/path/to/data/file.parquet",
    "file-format": "PARQUET",
    "partition": {"<part-field>":{"<data-type>":<value>}},
    "record-count": <num-records>,
    "null-value-counts": [{
      "column-index": "1", "value": 4
    }, ...],
    "lower-bounds": [{
      "column-index": "1", "value": "aaa"
    }, ...],
    "upper-bounds": [{
      "column-index": "1", "value": "eee"
    }, ...],
  }
  ...
}
{
 ...
}
```

Apache Iceberg - Architecture

- **Manifest File** - Manifest files track data files as well as additional details and statistics about each file. As mentioned earlier, the primary difference that allows Iceberg to address the problems of the Hive table format is tracking data at the file level — manifest files are the boots on the ground that do that.

  Each manifest file keeps track of a subset of the data files for parallelism and reuse efficiency at scale. They contain a lot of useful information that is used to improve efficiency and performance while reading the data from these data files, such as details about partition membership, record count, and lower and upper bounds of columns. These statistics are written for each manifest's subset of data files during write operation, and are therefore more likely to exist, be accurate, and be up to date than statistics in Hive.

  As to not throw the baby out with the bathwater, Iceberg is file-format agnostic, so the manifest files also specify the file format of the data file, such as Parquet, ORC, or Avro.

When a SELECT query is reading an Iceberg table and has a manifest file open after getting its location from the manifest list, the query engine then reads the value of the file-path entries for each data-file object, and opens the data files. It could also do some optimizations at this stage like using row counts or filtering of data using the partition or column statistic information.

Apache Iceberg - Architecture

- **Data Layer** - Each file in the data layer represents a leaf node in the Iceberg table's tree structure, providing scalable and low-cost storage by utilizing distributed filesystems like Hadoop Distributed File System (HDFS) or cloud object storage solutions such as Amazon S3, Azure Data Lake Storage (ADLS), and Google Cloud Storage (GCS).

Datafiles in Iceberg can use various file formats, such as Apache Parquet, Apache ORC, and Apache Avro. Iceberg's format-agnostic nature provides flexibility, allowing organizations to choose the format that best fits their specific workloads. Parquet, with its columnar structure, is commonly used for its performance benefits in analytical workloads. Parquet files allow engines to read columns and even row groups independently, improving parallelism, compression, and query efficiency by leveraging statistics like minimum and maximum values for each column.

A Look Under the Covers When CRUDing

Grow Data Skills

- **CREATE TABLE** - First, let's create a table in our environment.

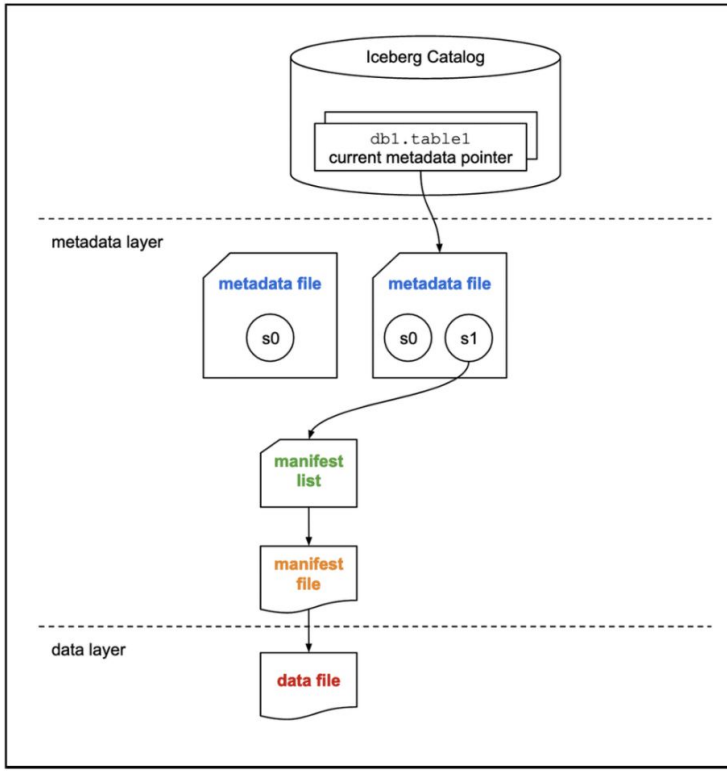
```
CREATE TABLE table1 (
    order_id BIGINT,
    customer_id BIGINT,
    order_amount DECIMAL(10, 2),
    order_ts TIMESTAMP
)
USING iceberg
PARTITIONED BY ( HOUR(order_ts) );
```

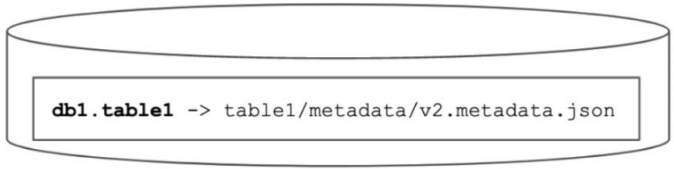
A Look Under the Covers When CRUDing

Grow Data Skills

- **INSERT** - Now, let's add some data to the table (albeit, literal values).

```
INSERT INTO table1 VALUES
    123,
    456,
    36.17,
    '2021-01-26 08:10:23'
);
```






```
table1/
 |- metadata/
 |    |- v1.metadata.json
 |    |- v2.metadata.json
 |    |- snap-2938-1-4103.avro
 |    |- d8f9-ad19-4e.avro
 |- data/
 |    |- order_ts_hour=2021-01-26-08/
 |    |    |- 00000-5-cae2d.parquet
```

A Look Under the Covers When CRUDing

Grow Data Skills

- **MERGE INTO / UPSERT** - Now, let's step through a MERGE INTO / UPSERT operation.

  Let's assume we've landed some data into a staging table we created in the background. In this simple example, information is logged each time there's a change to the order, and we want to keep this table showing the most recent details of each order, so we update the order amount if the order ID is already in the table. If we don't have a record of that order yet, we want to insert a record for this new order.

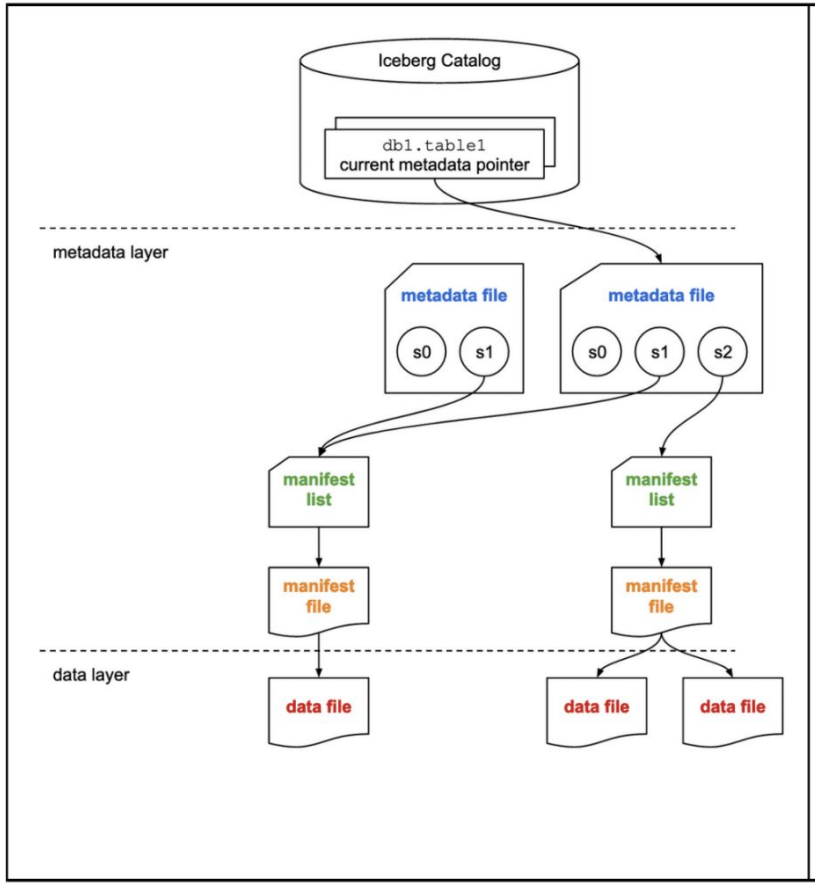
  In this example, the stage table includes an update for the order that's already in the table (order_id=123) and a new order that isn't in the table yet, which occurred on January 27, 2021 at 10:21:46.

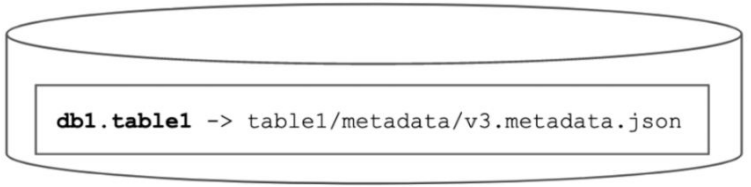
```
MERGE INTO table1
USING ( SELECT * FROM table1_stage ) s
    ON table1.order_id = s.order_id
WHEN MATCHED THEN
    UPDATE table1.order_amount = s.order_amount
WHEN NOT MATCHED THEN
    INSERT *
```

A Look Under the Covers When CRUDing

Grow Data Skills

- **MERGE INTO / UPSERT**





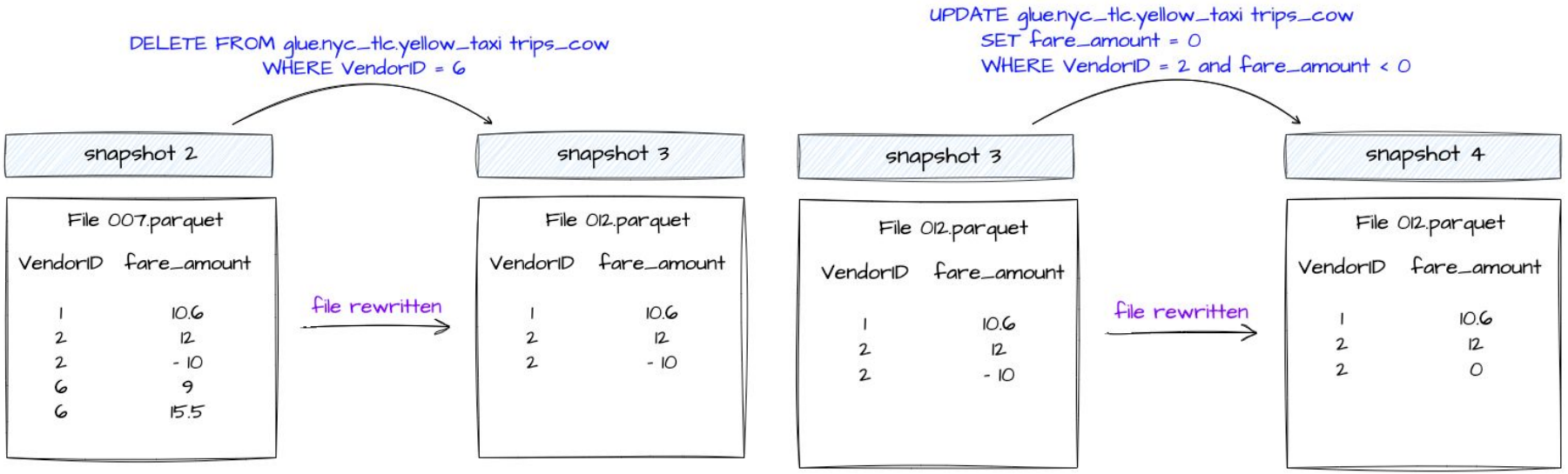

```
table1/
 |- metadata/
 |    |- v1.metadata.json
 |    |- v2.metadata.json
 |    |- v3.metadata.json
 |    |- snap-29c8-1-b103.avro
 |    |- snap-9fa1-3-16c3.avro
 |    |- d8f9-ad19-4e.avro
 |    |- 0d9a-98fa-77.avro
 |- data/
 |    |- order_ts_hour=2021-01-26-08/
 |    |    |- 00000-5-cae2d.parquet
 |    |    |- 00000-1-aef71.parquet
 |    |- order_ts_hour=2021-01-27-10/
 |    |    |- 00000-3-0fa3a.parquet
```

Copy-On-Write (CoW) and Merge-On-Read (MoR)

Grow Data Skills

In **Copy-On-Write** approach, if even a single row in a data file is updated or deleted, the associated data file is rewritten with the updated or deleted records.

The new snapshot of the table created because of these operations will point to this newer version of the data file. This is the default approach.



Copy-On-Write (CoW) and Merge-On-Read (MoR)

Before we dive into MOR, it's important to understand the Delete Files and what information these files have.

Delete Files

Delete files track which records in the dataset have been logically deleted and need to be ignored when a query engine tries to read the data from an Iceberg Table.

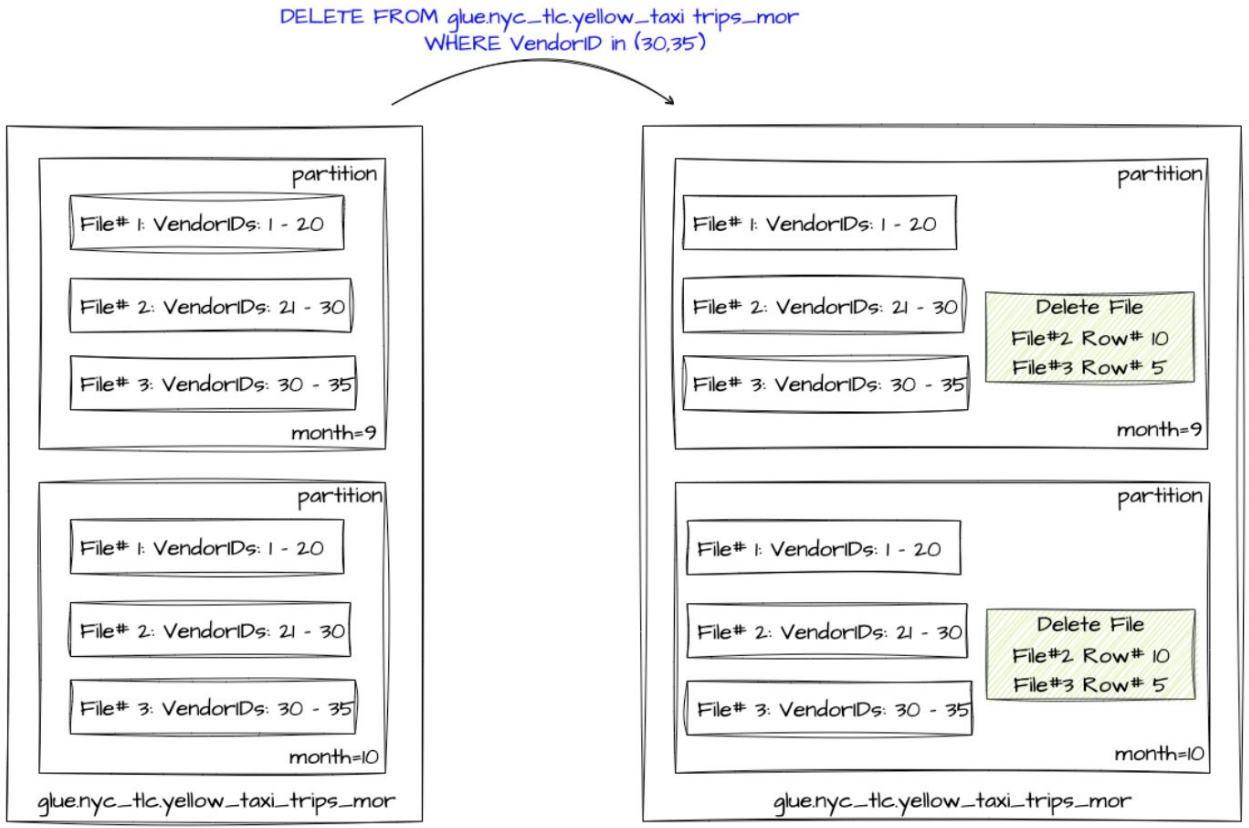
Delete files are created within each partition depending on the data file from where the record is logically deleted or updated.

There are 2 types of delete files based on how these delete files store delete records information.

Copy-On-Write (CoW) and Merge-On-Read (MoR)

Grow Data Skills

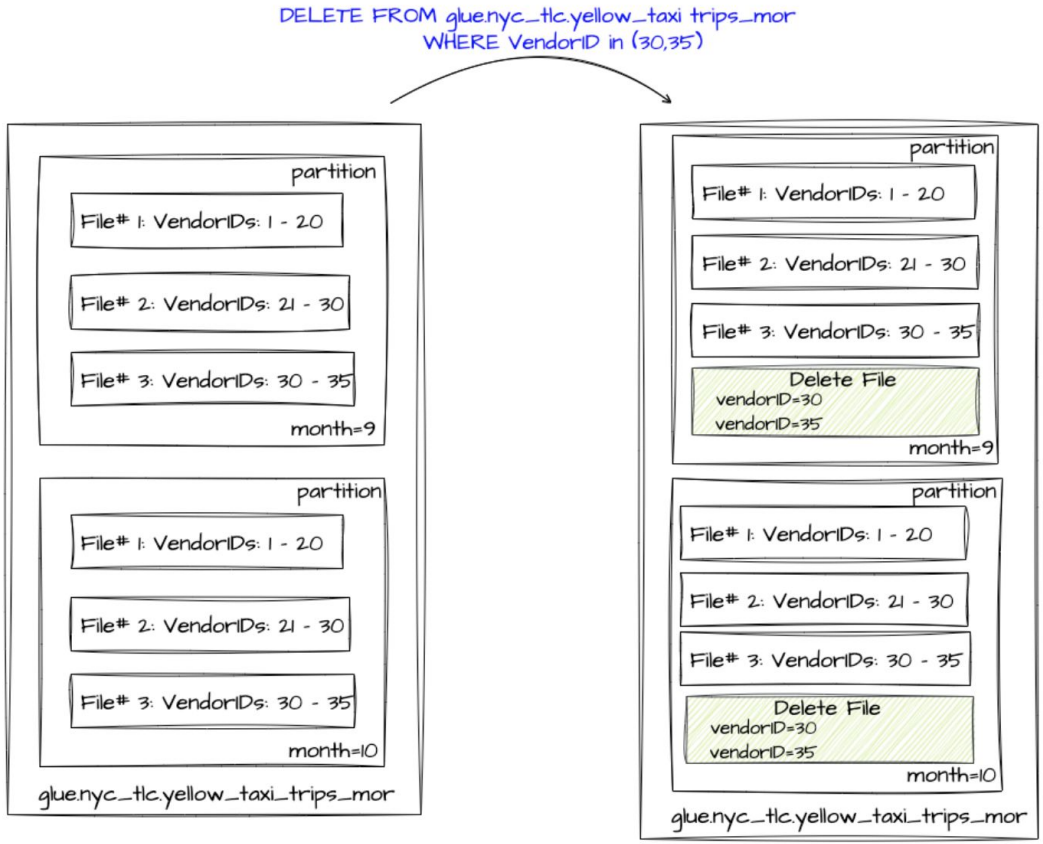
Positional Delete Files - Positional Delete files store the exact position of the deleted records in the dataset. It keeps track of the file path of the data file along with the position of the deleted records in that file.



Copy-On-Write (CoW) and Merge-On-Read (MoR)

Grow Data Skills

Equality Delete Files - Equality Delete Files stores the value of one or more columns of the deleted records. These column values are stored based on the condition used while deleting these records.

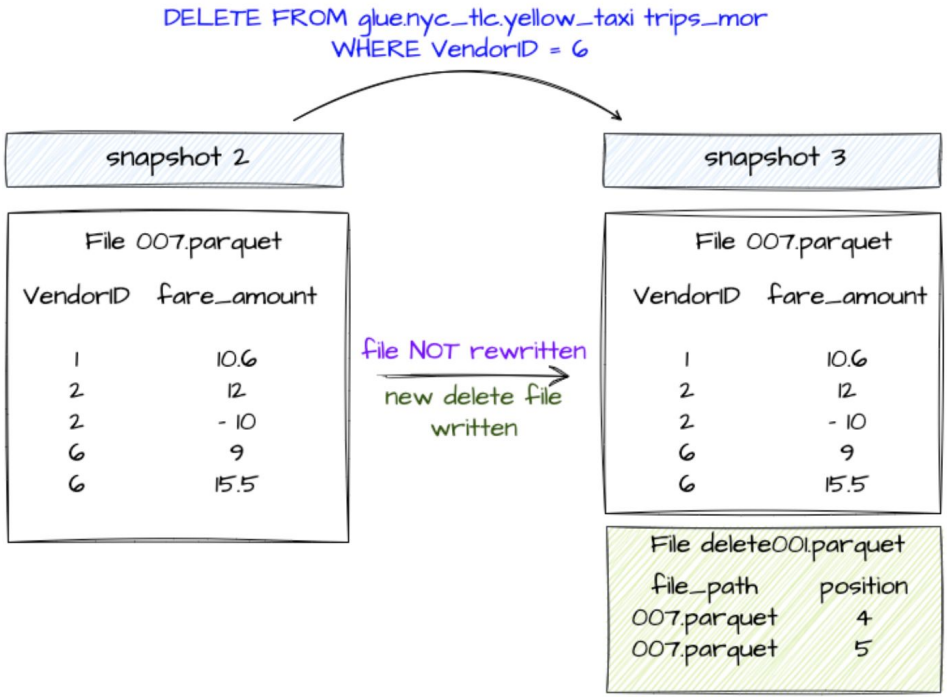


Copy-On-Write (CoW) and Merge-On-Read (MoR)

Grow Data Skills

In Merge-on-Read approach, update or delete operations on the Iceberg Table, the existing data files are not rewritten. Instead, a delete file is generated that keeps track of which records need to be ignored.

In case of deleting records, the record entries are listed in a Delete File.



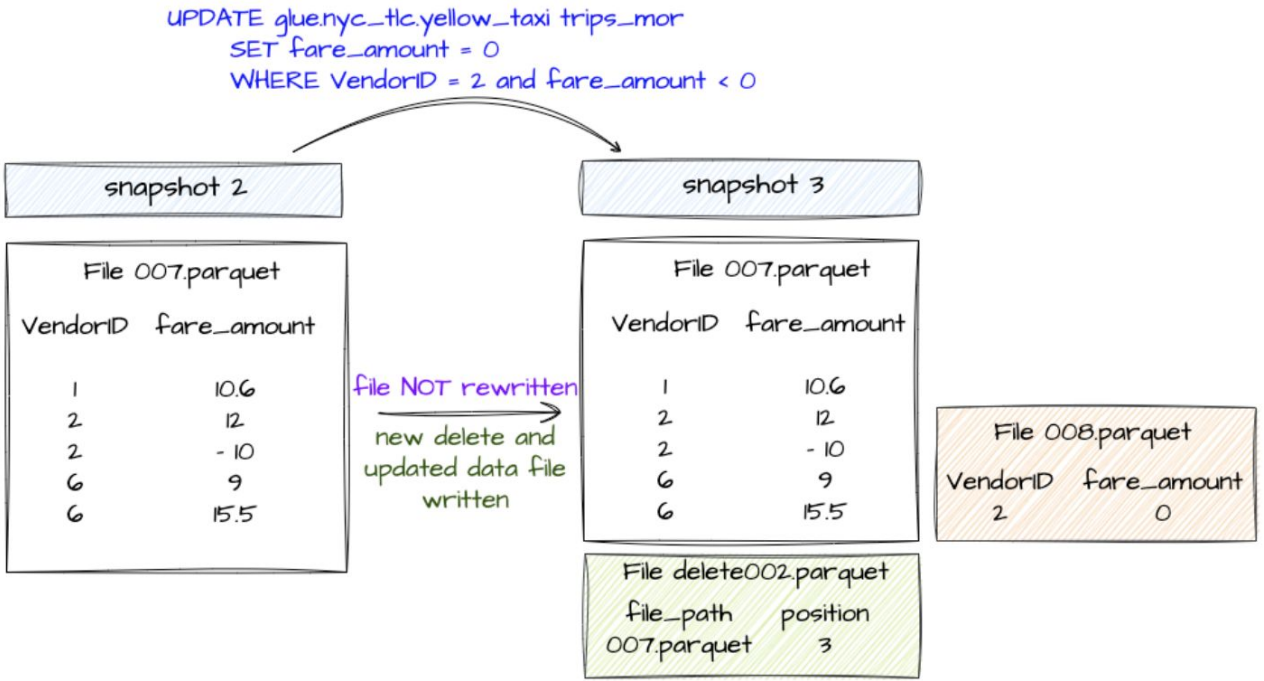
Merge-on-Read: New Delete File created after DELETE operation

Copy-On-Write (CoW) and Merge-On-Read (MoR)

Grow Data Skills

In case of updating records:

- The records to be updated are listed in a delete file.
- A new data file is created that contains only the updated records.



Merge-on-Read: New Delete and New updated data file written after UPDATE operation

How to choose between COW and MOR?

Grow Data Skills

- **COW:**
  - In case of row-level deletes/updates, it rewrites the entire file even if there is a single record is impacted.
  - More data needs to be written that causes slower row-level Updates/Deletes.
  - Data is read without reconciling. It writes any deleted or updated files, resulting in faster reads.
- **MOR:**
  - In case of row-level deletes/updates, it avoids rewriting the entire data file.
  - It writes only the Delete File along with the updated data file in case of Updates i.e. basically writing less data and hence faster writes.
  - Data is read along with reconciling any deleted or updated files, resulting in slower reads.

Summarized COW vs MOR

APPROACH	READ SPEED	WRITE SPEED	BEST PRACTICES (to minimize read cost)	BEST SUITED USE CASE
Copy-on-Write	Fastest	Slowest row level updates/deletes		Infrequent updates/deletes
Merge-on-Read (Positional Delete File)	Fast	Fast row level updates/deletes	Use Regular Compaction	Frequent updates/deletes
Merge-on-Read (Equality Delete File)	Slowest	Fastest row level updates/deletes	Use Frequent Compaction	Frequent updates/deletes (Read time from table is not an issue, and MOR with Positional Delete writes are not fast enough)

A Look Under the Covers When CRUDing

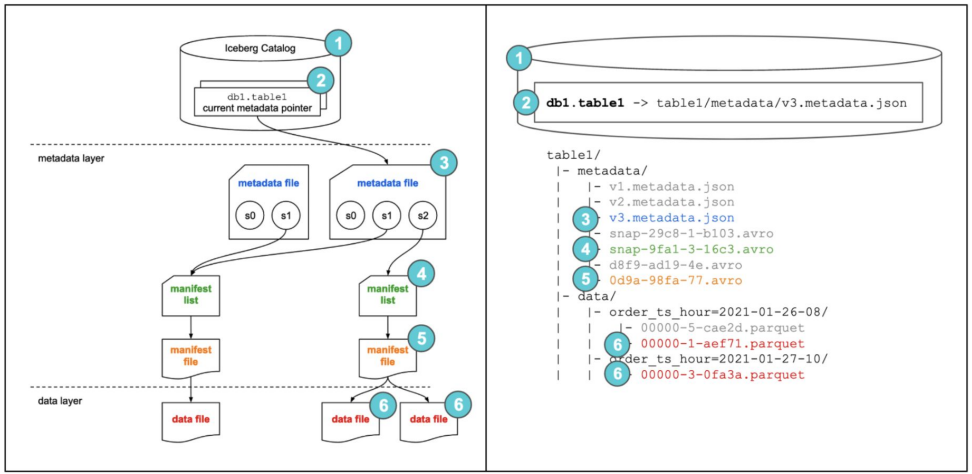
Grow Data Skills

SELECT - Let's review the SELECT path again, but this time on the Iceberg table we've been working on.

```
SELECT * FROM db1.table1
```

When this SELECT statement is executed, the following process happens:

- The query engine goes to the Iceberg catalog
- It then retrieves the current metadata file location entry for db1.table1
- It then opens this metadata file and retrieves the entry for the manifest list location for the current snapshot, s2
- It then opens this manifest list, retrieving the location of the only manifest file
- It then opens this manifest file, retrieving the location of the two data files
- It then reads these data files, and since it's a SELECT *, returns the data back to the client



Create Table Command in Iceberg

Grow Data Skills

```
CREATE TABLE ecommerce.transactions (
    transaction_id BIGINT COMMENT 'Unique identifier for the transaction',
    customer_id BIGINT COMMENT 'Unique identifier for the customer',
    product_id BIGINT COMMENT 'Unique identifier for the product',
    quantity INT COMMENT 'Quantity of the product purchased',
    total_amount DECIMAL(10, 2) COMMENT 'Total amount for the transaction',
    transaction_date TIMESTAMP COMMENT 'Timestamp of the transaction',
)
USING iceberg PARTITIONED BY (country, transaction_date)
WITH TBLPROPERTIES (
    -- General Table Properties
    'format-version' = '2',                     -- Use Iceberg format version 2
    'write.format.default' = 'parquet',         -- Default file format is Parquet
    'write.target-file-size-bytes' = '536870912',  -- Target file size of 512 MB
    'write.upsert.enabled' = 'true',            -- Enable upserts for the table
    'write.update.mode' = 'merge-on-read',      -- Use merge-on-read for delete operations
    'write.delete.mode'='copy-on-write',
    'write.merge.mode'='merge-on-read',
    'write.distribution-mode' = 'hash',         -- Use hash-based distribution for writes

    -- Snapshot and Metadata Management
    'commit.manifest.min-count-to-merge' = '5',    -- Minimum number of manifests to trigger a merge
    'commit.manifest.target-size-bytes' = '104857600', -- Target size for each manifest (100 MB)
    'snapshot-id-inheritance.enabled' = 'true',   -- Optimize snapshot management
    'metadata.delete-after-commit.enabled' = 'true', -- Clean up old metadata files after commits

    -- Data Compaction and Optimization
    'write.merge.small-files' = 'true',         -- Merge small files automatically
    'write.merge.file-size-threshold' = '134217728', -- Threshold for merging small files (128 MB)
    'write.compaction.file-count-threshold' = '50', -- Trigger compaction when there are > 50 small files

    -- Query Optimization
    'read.split.target-size' = '268435456',     -- Split size for parallel reads (256 MB)

    -- Garbage Collection
    'gc.enabled' = 'true',                      -- Enable garbage collection for orphaned files
    'gc.file-threshold' = '10485760',           -- Minimum size of orphan files for GC (10 MB)

    -- Performance Settings
    'parquet.row-group-size-bytes' = '134217728'   -- Parquet row group size (128 MB)
);
```

Insert Commands in Iceberg

Grow Data Skills

– Normal insert where data will be placed automatically inside partitions

```
INSERT INTO ecommerce.transactions
VALUES
    (1001, 12345, 56789, 2, 49.99, 99.98, '2025-01-13 12:30:45', 'United States', 'Credit Card',
'Completed'),
    (1002, 12346, 56780, 1, 19.99, 19.99, '2025-01-13 13:15:20', 'Canada', 'PayPal', 'Completed');
```

– Static overwrite of partitions

```
INSERT OVERWRITE ecommerce.transactions
PARTITION (country = 'United States', transaction_date = '2025-01-13')
VALUES
    (1003, 12347, 56781, 3, 29.99, 89.97, '2025-01-13 14:00:00', 'United States', 'Debit Card', 'Pending'),
    (1004, 12348, 56782, 5, 9.99, 49.95, '2025-01-13 15:20:10', 'United States', 'Credit Card', 'Completed');
```

– Dynamic overwrite of partitions

```
INSERT OVERWRITE ecommerce.transactions
VALUES
    (1003, 12347, 56781, 3, 29.99, 89.97, '2025-01-13 14:00:00', 'United States', 'Debit Card', 'Pending'),
    (1004, 12348, 56782, 5, 9.99, 49.95, '2025-01-13 15:20:10', 'United States', 'Credit Card', 'Completed'),
    (1005, 12349, 56783, 1, 99.99, 99.99, '2025-01-14 09:00:00', 'Canada', 'Credit Card', 'Completed');
```

Delete & Update Commands in Iceberg

Grow Data Skills

```
DELETE FROM ecommerce.transactions
WHERE transaction_date = '2025-01-13';
```

```
UPDATE ecommerce.transactions
SET status = 'Completed'
WHERE transaction_id = 1003;
```

Alter Command in Iceberg

Grow Data Skills

```
ALTER TABLE catalog.db.sample_table SET TBLPROPERTIES (
        'write.delete.mode'='merge-on-read',
        'write.update.mode'='copy-on-write',
        'write.merge.mode'='copy-on-write'
);
```

Merge Query in Iceberg

Grow Data Skills

```
MERGE INTO ecommerce.transactions AS target
USING incoming_updates AS source
ON target.transaction_id = source.transaction_id
WHEN MATCHED AND source.status = 'Cancelled' THEN DELETE
WHEN MATCHED THEN UPDATE SET
    target.quantity = source.quantity,
    target.total_amount = source.total_amount,
    target.status = source.status
WHEN NOT MATCHED THEN INSERT (
    transaction_id, customer_id, product_id, quantity, price, total_amount, transaction_date, country,
payment_method, status
) VALUES (
    source.transaction_id, source.customer_id, source.product_id, source.quantity, source.price,
source.total_amount, source.transaction_date, source.country, source.payment_method, source.status
);
```

Time travel (Read) in Iceberg

Grow Data Skills

```
-- To list down all snapshots of a Iceberg table
SELECT *
FROM ecommerce.transactions.snapshots;


-- Read from timestamp
SELECT *
FROM ecommerce.transactions
FOR TIMESTAMP AS OF TIMESTAMP '2025-01-01 00:00:00';


-- Read from snapshot
SELECT *
FROM ecommerce.transactions
FOR SNAPSHOT 1234567890123456789;
```

Compaction in Iceberg

Grow Data Skills

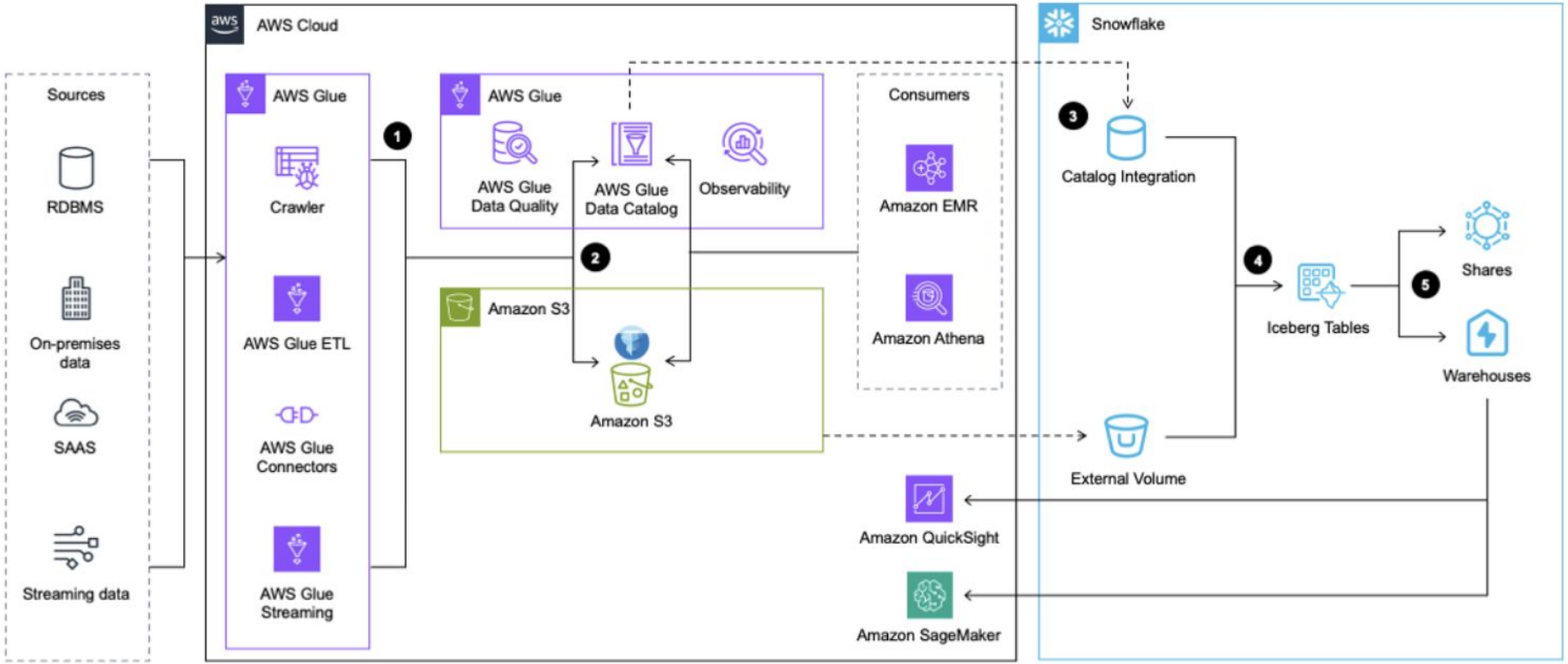
Compaction is a technique and a recommended (yet, mandatory) maintenance that needs to happen on Iceberg table periodically. It will help in combining smaller files into fewer larger files.

```
CALL iceberg.system.rewrite_data_files(
  table => 'database_name.table_name',
   options => map(
    'target-file-size-bytes', '524288000',  -- Target size of each compacted file (500MB)
    'split-size', '268435456',              -- Size of each data split (256MB) for parallel processing
    'max-file-size', '1073741824'           -- Maximum allowable size of a single file (1GB)
  )
);
```

```
CALL iceberg.system.rewrite_manifests(
  table => 'database_name.table_name',
  options => map(
    'max-manifest-file-size', '104857600',  -- Maximum size of each manifest file (100MB)
    'min-manifest-file-size', '5242880',    -- Minimum size to consider for compaction (5MB)
    'manifest-target-count', '10'           -- Target number of compacted manifests
  )
);
```

Case Study - 1

Grow Data Skills



Case Study - 1

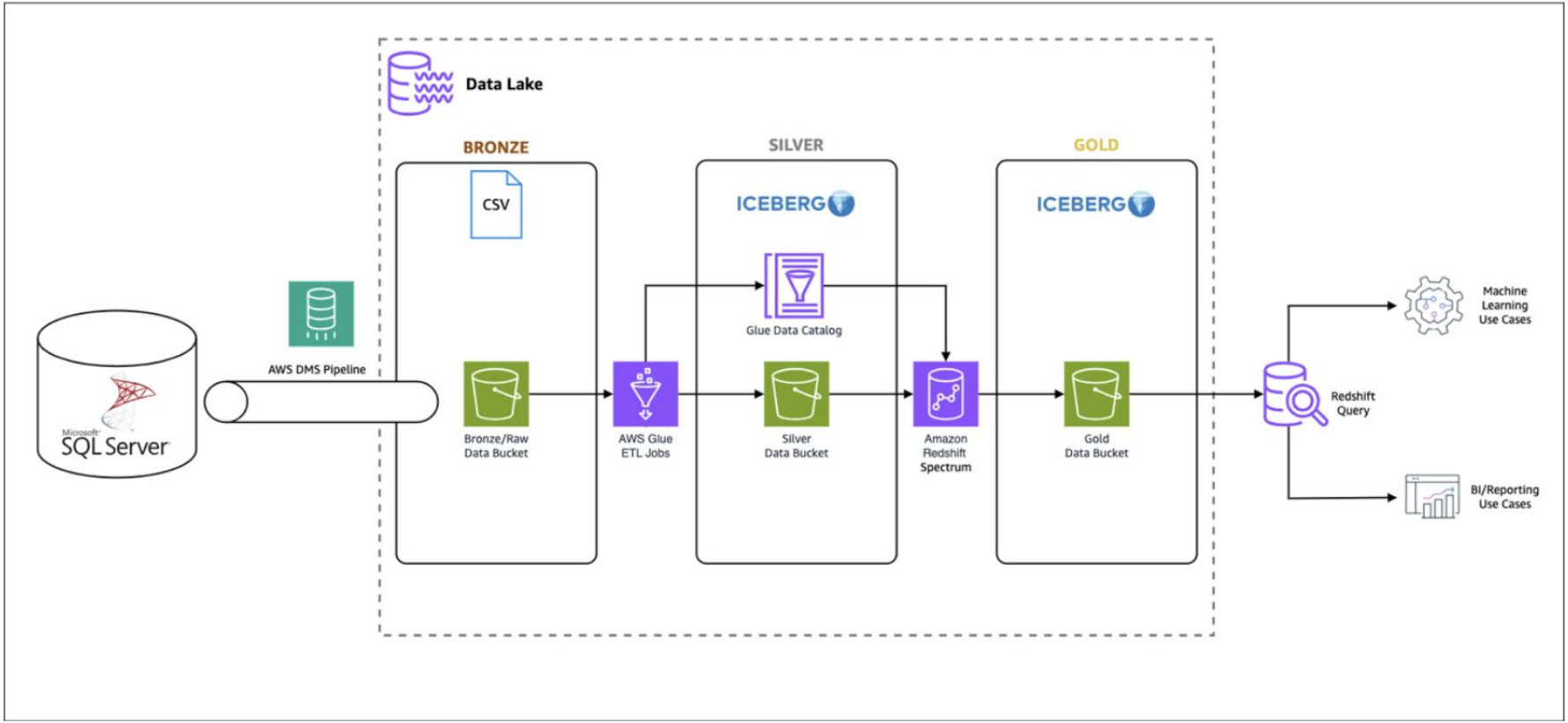
Grow Data Skills

The workflow includes the following steps:

- AWS Glue extracts data from applications, databases, and streaming sources. AWS Glue then transforms it and loads it into the data lake in Amazon S3 in Iceberg table format, while inserting and updating the metadata about the Iceberg table in AWS Glue Data Catalog.
- The AWS Glue crawler generates and updates Iceberg table metadata and stores it in AWS Glue Data Catalog for existing Iceberg tables on an S3 data lake.
- Snowflake integrates with AWS Glue Data Catalog to retrieve the snapshot location.
- In the event of a query, Snowflake uses the snapshot location from AWS Glue Data Catalog to read Iceberg table data in Amazon S3.
- Snowflake can query across Iceberg and Snowflake table formats. You can share data for collaboration with one or more accounts in the same Snowflake region. You can also use data in Snowflake for visualization using Amazon QuickSight, or use it for machine learning (ML) and artificial intelligence (AI) purposes with Amazon SageMaker.

Case Study 2 - Medallion Architecture Using Iceberg

Grow Data Skills



Apache Hudi

Grow Data Skills

Apache Hudi **(Hadoop Upserts and Incremental)** is an open-source transactional data lake platform (open table format) designed to manage large-scale datasets on distributed storage like HDFS, Amazon S3, or Azure Data Lake. Hudi enables data versioning, efficient data updates, deletes, and incremental processing, making it a powerful choice for modern data lakes and real-time analytics.

Key Features of Apache Hudi

- **ACID Transactions:**
  - Provides atomicity, consistency, isolation, and durability (ACID) for writes, updates, and deletes in data lakes.
  - Ensures consistency even with concurrent writes.
- **Data Upserts and Deletes:**
  - Unlike traditional data lakes that are append-only, Hudi allows row-level updates and deletes.
- **Real-Time and Incremental Processing:**
  - Supports real-time ingestion and incremental query capabilities for low-latency analytics.
- **Two Storage Modes:**
  - Copy-on-Write (CoW): Entire files are rewritten during updates.
  - Merge-on-Read (MoR): Log files track changes, and merging happens at query time, improving write efficiency.
- **Time Travel:**
  - Query historical snapshots of data for auditing, debugging, or recovering deleted data.
- **Efficient Storage Management:**
  - Automatically compacts small files and manages metadata for optimal performance.
- **Integration with Big Data Ecosystems:**
  - Seamlessly integrates with Apache Spark, Presto, Hive, Trino, Flink, and DeltaStreamer.

Apache Hudi

Grow Data Skills

Apache Hudi

Grow Data Skills

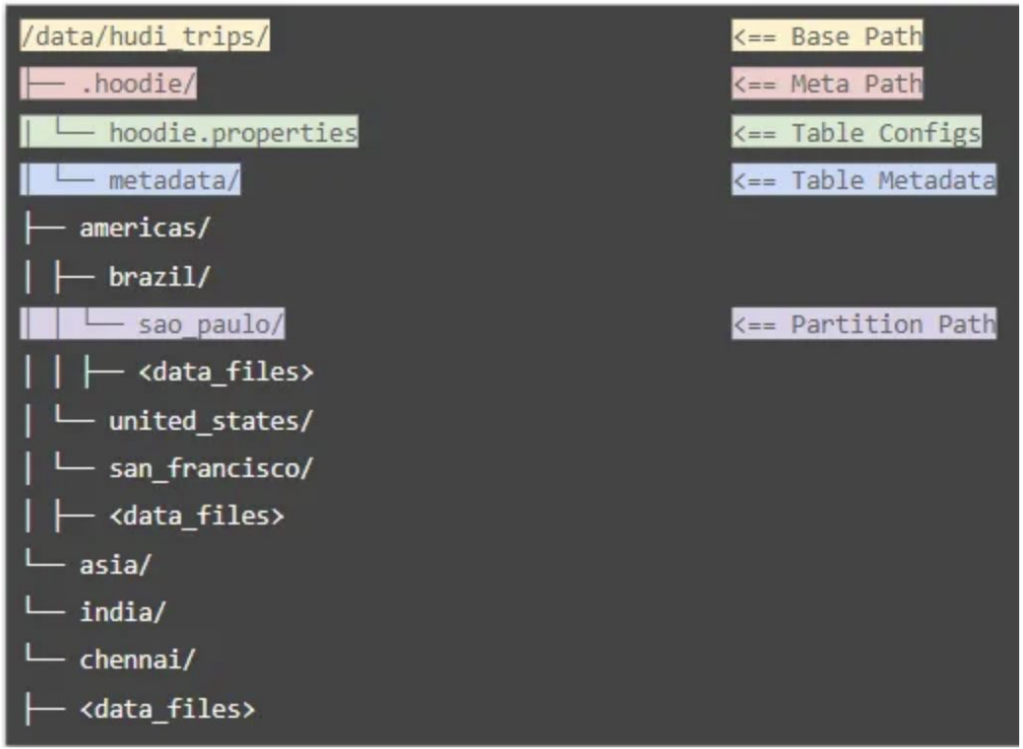
Apache Hudi's architecture is designed to provide a comprehensive solution for managing data in a modern data lake environment. It consists of several key components that seamlessly work together to enable efficient data operations and analytics.

- **Data Sources:** Apache Hudi can ingest data from various sources, including data streams, databases, and cloud storage. This versatility ensures compatibility with a wide range of data inputs.
- **Hudi Core**: The heart of Apache Hudi comprises several essential components:
  - ***ACID Guarantees***: Just like databases, Hudi offers ACID (Atomicity, Consistency, Isolation, Durability) guarantees, ensuring data integrity and reliability.
  - ***Incremental Pipelines***: Hudi introduces efficient incremental data processing, enabling low-latency analytics.
  - ***Multimodal Indexes***: Multimodal indexing supports various optimized query types and patterns.
  - ***Managed Tables***: Hudi manages data in tables, providing structure and organization within the lake.
- **Lakehouse Platform**: Apache Hudi seamlessly eases into a "lakehouse," combining data lake and data warehouse capabilities offering the best of both worlds with efficient data storage and analytics.
- **Metadata**: Metadata management is crucial for tracking and organizing data. Hudi has an efficient metadata management system to keep track of changes to schemas and table structures.
- **Data Sinks**: Data processed through Apache Hudi can be directed to various data sinks, including BI analytics tools, interactive analytics platforms, batch processing systems, and stream analytics solutions. This flexibility ensures that data is available for a wide range of analytics and reporting purposes.
- **Simplified Orchestration**: Orchestrating data pipelines gets simplified with Apache Hudi, as it integrates well with orchestration tools to automate data workflows.

Apache Hudi Storage Layout

Grow Data Skills

Apache Hudi organizes data in a structured directory layout on distributed storage systems like HDFS, Amazon S3, or Azure Data Lake Storage. This layout is designed to efficiently manage metadata, partitions, and data files, supporting features like incremental updates, time travel.



Apache Hudi Storage Layout

Grow Data Skills

- **Base Path** - The root directory where all the table data and metadata are stored.
  - ***Example***: /data/hudi_trips/
- **Meta Path** - Located at ***/.hoodie/*** within the base path.
  - Contains the metadata required to manage the table and its operations.
  - Key Components in Meta Path:
    - ***Hoodie.properties*** - Contains table-level configurations like storage type (Copy-on-Write or Merge-on-Read) and schema details.
    - ***metadata/*** - Stores detailed metadata files, such as commit files (.commit), savepoint files, and auxiliary information required for incremental processing and time travel.
- **Partition Paths**
  - Hudi organizes data into partition directories based on the partitioning logic defined at table creation.
  - Each partition contains data files specific to that partition.
  - Example of Partition Paths:
    - /data/hudi_trips/americas/brazil/sao_paulo/
    - /data/hudi_trips/asia/india/chennai/
- **Data Files** - Data files are stored within the partition paths and managed by Hudi based on the storage mode
  - ***Copy-on-Write (CoW)*** - Contains updated data files where entire files are rewritten after modifications.
  - ***Merge-on-Read (MoR)*** - Contains base files (like Parquet) and log files (delta logs) that record incremental changes.

Apache Hudi Query Types

Grow Data Skills

Hudi supports the following query types.

- **Snapshot Queries :** Queries see the latest snapshot of the table as of the latest completed action. These are the regular SQL queries everyone is used to running on a table. Hudi storage engine accelerates these snapshot queries with indexes whenever possible, on supported query engines.

  SELECT transaction_id, customer_id, total_amount, transaction_date FROM hudi_table;

- **Time Travel Queries :** Queries a snapshot of a table as of a given instant in the past. Time-Travel queries help access multiple versions of a table (for e.g. Machine learning feature stores, to score algorithms/models on exact data used to train them) on instants in the active timeline or savepoints in the past.

  SELECT transaction_id, customer_id, total_amount, transaction_date FROM hudi_table AS OF '20250101000000';

- **Read Optimized Queries (Only MoR tables)** : Read-optimized queries provides excellent snapshot query performance via purely columnar files (e.g. Parquet base files). Users typically use a compaction strategy that aligns with a transaction boundary, to provide older consistent views of table/partitions. This is useful to integrate Hudi tables from data warehouses that typically only query columnar base files as external tables or latency insensitive ML/AI training jobs that favor efficiency over data freshness.

  SELECT transaction_id, customer_id, total_amount, transaction_date FROM hudi_table_ro;

Apache Hudi Query Types

Grow Data Skills

- **Incremental Queries (Latest State) :** Incremental queries only return new data written to the table since an instant on the timeline. Provides latest value of records inserted/updated (i.e 1 record output by query for each record key), since a given point in time of the table. Can be used to "diff" table states between two points in time.

  SELECT transaction_id, customer_id, total_amount, transaction_date FROM hudi_table WHERE _hoodie_commit_time > '20250101000000';

- **Incremental Queries(CDC) :** These are another type of incremental queries, that provides database like change data capture streams out of Hudi tables. Output of a CDC query contain records inserted or updated or deleted since a point in time or between two points in time with both before and after images for each change record, along with operations that caused the change.

  SELECT * FROM hudi_table WHERE _hoodie_commit_time BETWEEN '20250101000000' AND '20250102000000';

Create Table Command in Hudi

Grow Data Skills

```
CREATE TABLE ecommerce.hudi_transactions (
    transaction_id BIGINT COMMENT 'Unique identifier for the transaction',
    customer_id BIGINT COMMENT 'Unique identifier for the customer',
    product_id BIGINT COMMENT 'Unique identifier for the product',
    quantity INT COMMENT 'Number of products purchased',
    total_amount DECIMAL(10, 2) COMMENT 'Total transaction amount',
    transaction_date DATE COMMENT 'Date of the transaction',
    country STRING COMMENT 'Country of the transaction'
)
  USING hudi
  OPTIONS (
    -- **Hudi Table Type**
    'hoodie.table.type' = 'MERGE_ON_READ', -- Options: COPY_ON_WRITE or MERGE_ON_READ

    -- **Compaction and Log Management**
    'hoodie.compact.inline.max.delta.commits' = '5', -- Trigger compaction after 5 commits

    -- **Cleaning Policy**
    'hoodie.cleaner.policy' = 'KEEP_LATEST_COMMITS', -- Retain latest commits
    'hoodie.cleaner.commits.retained' = '10',       -- Retain 10 commits during cleaning

    -- **Indexing**
    'hoodie.index.type' = 'BLOOM', -- Default: BLOOM (can also use SIMPLE or GLOBAL_BLOOM)
    'hoodie.bloom.index.filter.type' = 'DYNAMIC_V0', -- Optimized Bloom filter for updates
    'hoodie.bloom.index.update.partition.path' = 'true', -- Update partition paths for record key changes

    -- **Time Travel and Incremental Queries**
    'hoodie.keep.min.commits' = '10', -- Minimum commits to retain for time travel
    'hoodie.keep.max.commits' = '20', -- Maximum commits to retain before cleanup
    'hoodie.clean.automatic' = 'true', -- Automatically clean old commits

    -- **Schema Evolution**
    'hoodie.datasource.write.schema.allow.auto.evolution' = 'true', -- Enable schema evolution
    'hoodie.datasource.write.reconcile.schema' = 'true' -- Reconcile schema during writes
)
PARTITIONED BY (country);
```

Alter Command in Hudi

Grow Data Skills

```
ALTER TABLE ecommerce.hudi_transactions
SET TBLPROPERTIES (
    'hoodie.compact.inline' = 'true',
    'hoodie.compact.inline.max.delta.commits' = '10'
);

ALTER TABLE ecommerce.hudi_transactions
DROP PARTITION (country='USA');

ALTER TABLE ecommerce.hudi_transactions
ADD PARTITION (region='North America');
```

Insert, Update & Delete Command in Hudi

Grow Data Skills

```
INSERT INTO ecommerce.hudi_transactions
VALUES
    (1001, 12345, 56789, 2, 49.99, 99.98, '2025-01-13', 'Credit Card', 'Completed', 'USA'),
    (1002, 12346, 56790, 1, 19.99, 19.99, '2025-01-13', 'PayPal', 'Pending', 'Canada');

UPDATE ecommerce.hudi_transactions
SET total_amount = 79.99, status = 'Refunded'
WHERE transaction_id = 1001;

DELETE FROM ecommerce.hudi_transactions
WHERE transaction_id = 1002;
```

Merge Query in Hudi

Grow Data Skills

```
MERGE INTO ecommerce.hudi_transactions AS target
USING incoming_updates AS source
ON target.transaction_id = source.transaction_id
WHEN MATCHED THEN UPDATE SET
    target.customer_id = source.customer_id,
    target.product_id = source.product_id,
    target.quantity = source.quantity,
    target.price = source.price,
    target.total_amount = source.total_amount,
    target.transaction_date = source.transaction_date,
    target.payment_method = source.payment_method,
    target.status = source.status,
    target.country = source.country
WHEN NOT MATCHED THEN INSERT
    (transaction_id, customer_id, product_id, quantity, price, total_amount, transaction_date, payment_method, status, country)
    VALUES
    (source.transaction_id, source.customer_id, source.product_id, source.quantity, source.price, source.total_amount, source.transaction_date, source.payment_method, source.status, source.country);
```

Time Travel QUERIES in Hudi

Grow Data Skills

Query to get commit details - ***SELECT * FROM hudi_table.snapshots;***

- To fetch the table’s state as of a specific commit time:

  SELECT * FROM hudi_table AS OF '20250101000000';

- To fetch data that was updated or inserted after a specific commit time:

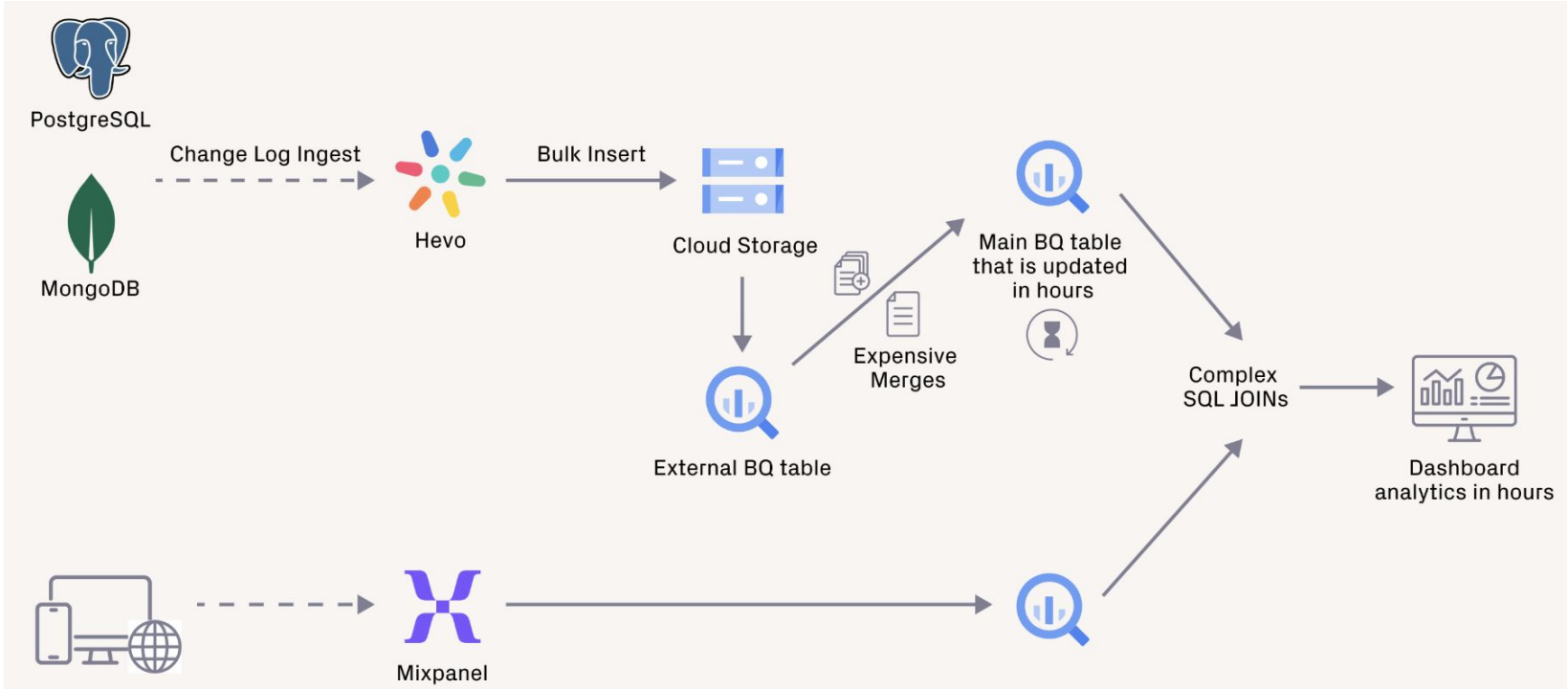
  SELECT * FROM hudi_table WHERE _hoodie_commit_time > '20250101000000';

- To retrieve data changes between two points in time:

  SELECT * FROM hudi_table WHERE _hoodie_commit_time BETWEEN '20250101000000' AND '20250102000000';

Case Study - Apna (Previous Architecture)

Grow Data Skills



Case Study - Apna (Previous Architecture)

Grow Data Skills

This architecture came with numerous issues related to cost, efficiency, and maintenance. Examples include:

- The monolithic software backend meant that software updates were infrequent, difficult to develop, and difficult to deliver.
- High costs prevented the use of real-time sync for CDC data via Hevo, forcing reliance on much slower batch processing. This limited data freshness and consistency.
- Batch processing also generated a large, temporary external BigQuery table, which was costly, time-consuming to maintain, and failure-prone.
- Time travel was only supported for seven days. Apna needed nearly unlimited time travel for flexible querying and comparisons to support feature development.
- Interactions across many different systems created multiple potential points of failure, resulting in significant operational challenges for the production system.

These challenges caused latency throughout the system, resulting in delayed data delivery, reduced developer productivity, and the inability to scale to meet user growth for Apna's existing applications. New functionality, such as the use of AI/ML for job matching, was simply out of reach with this legacy architecture.

Case Study - Apna (New Architecture)

Grow Data Skills