CLOUDERA

WHITEPAPER

Apache Iceberg: Performance and Cost Optimization Techniques

A Technical Reference Guide

By Dipankar Mazumdar, Director of Developer Relations, Cloudera



Table of Contents

Chapter 1. Introduction: Why Iceberg Tables Slow Down 5

Chapter 2. Partitioning and Partition Pruning 6

- Designing an Effective Partitioning Strategy 9
  - Start with Query Patterns 9
  - Select the Transform and Granularity 9
  - Account for Combined Fields and Data Skew 9
  - Validate the Layout in Production 10
  - Revisit the Strategy as the Table Evolves 10

Chapter 3. Compaction and the Small File Problem 10

- Rewriting Data Files 11
  - Maximum File-Group Size 13
  - Maximum Concurrent File-Group Rewrites 13
  - Target File Size 14
  - Partial Progress 14
  - Maximum Partial-Progress Commits 14
- Tuning Compaction for the Workload 15
  - Establish the Target File Size 15
  - Size the Rewrite Groups 15
  - Add Concurrency After Groups Exist 15
  - Use Partial Progress for Large Rewrites 15
  - Limit the Rewrite Scope 15
  - Validate the Result 16

Chapter 4. Managing Delete Files and Read Amplification 16

- Copy-on-Write and Merge-on-Read 16
- Position Deletes and Equality Deletes 17
  - Position Deletes 18
  - Equality Deletes 18
- Determining Which Deletes Apply 18

How Delete Files Increase Read Cost 19
Inspecting Delete Activity 19
Rewriting Position Delete Files 20
Applying Deletes Through Data-File Rewrites 21
Deletion Vectors in Iceberg v3 22
Tuning Delete Maintenance for the Workload 22
Consolidating Fragmented Position Delete Files 23
Rewriting Data with a High Deleted-Row Ratio 23
Managing Equality Deletes 23
Aligning Maintenance with the Write Strategy 23
Scoping Maintenance to Active Data 23
Validating Maintenance Thresholds 23

Chapter 5. Clustering Data 23

How Clustering Improves File Pruning 24
Partitioning and Clustering Solve Different Problems 25
Sort Orders in Apache Iceberg 25
Write Distribution and Maintaining the Sort Order 26
Lexicographic Sorting 26
Z-Ordering 27
Clustering Existing Data 28
Combining Clustering and Compaction 29
Sort Rewrite Configuration 29
Clustering Cost and Layout Degradation 30
Measuring Clustering Effectiveness 30
Designing an Effective Clustering Strategy 31
Start with Query Predicates 31
Choose Between Lexicographic Sorting and Z-Ordering 31
Account for Correlated Columns 31
Match the Rewrite Scope to Data Volatility 31
Coordinate Clustering with Compaction 31
Balance File Size and Pruning Granularity 31
Preserve the Layout During Writes 32
Validate the Cost-Benefit Tradeoff 32

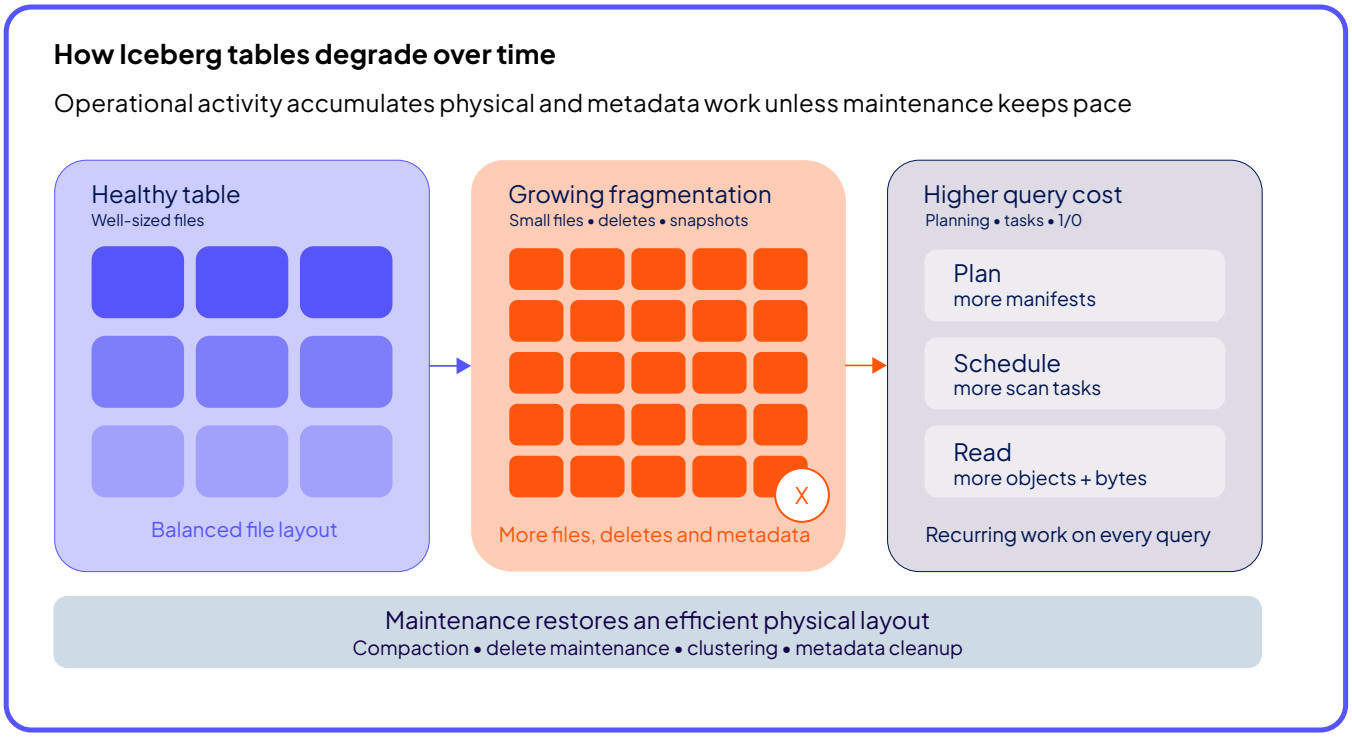
Chapter 6. Cleaning – Data and Metadata Maintenance 32
Expiring Snapshots 33
Account for Branches and Tags 34
Removing Orphan Files 34
Cleaning Table Metadata JSON Files 35
Rewriting Manifests 35
Designing a Safe Cleanup Strategy 36
Separate Storage Cleanup from Planning Optimization 36
Run File Rewrites Before Metadata Cleanup 36
Expire Snapshots Before Removing Orphans 36
Monitor the Outcome 36
Match Frequency to Table Activity 36
Chapter 7. Automating Iceberg Maintenance with Cloudera Lakehouse Optimizer 37
Policy-Based Maintenance 37
CLO Architecture and Execution Model 38
Policy Structure 38
Triggering Policy Evaluation 38
Event-Based Evaluation 38
Schedule-Based Evaluation 39
Manual Execution and Dry Runs 39
Associating Policies with Tables 39
Automating Compaction 39
Automating Clustering 40
Coordinating Data and Metadata Maintenance 41
Designing Policies Around Workload Classes 42
Path to Continuous Maintenance 42

Chapter 1

Introduction: Why Iceberg Tables Slow Down

Apache Iceberg provides the transactional and metadata foundation required to manage large analytical datasets in object storage. It tracks table state through snapshots, maintains file-level statistics, supports hidden partitioning and allows readers to plan queries without recursively listing storage directories. These capabilities address many of the limitations associated with traditional **data lakes**. They do not, however, guarantee that a table will remain performant as its data and workload evolve.

An Iceberg table that performs well when first deployed can gradually become slower and more expensive to query. Streaming ingestion and frequent micro-batch writes may produce thousands of small data files. Updates, deletes and merge operations can introduce delete files that must be evaluated during reads. Data written in ingestion order may no longer align with the columns used by analytical queries. Frequent commits can also increase the number of snapshots, manifests and metadata files associated with the table.



These conditions affect different parts of query execution. At a high level, the cost of reading an Iceberg table can be divided into five areas:

- **Planning work:** The engine must identify the current snapshot, inspect manifest metadata and determine which files may contain matching records.
- **File-processing overhead:** Every selected file creates work related to task scheduling, object-store requests, file opens and metadata reads.
- **Data scanned:** Files that cannot be eliminated using partition values or column statistics must be read, even when they contain few relevant records.
- **Delete processing:** Readers may need to load and apply position or equality deletes before returning the current state of the data.
- **Maintenance cost:** Compaction, clustering and metadata maintenance improve subsequent reads, but consume compute and rewrite data or metadata.

The contribution of each area depends on the table and its workload. A streaming table may suffer primarily from file-processing overhead, while a large append-only table with poorly localized values may read too much data despite having appropriately sized files. A table receiving frequent row-level changes may have acceptable data-file sizes but still incur substantial read amplification from delete files. As a result, no single optimization technique is sufficient for every Iceberg table.

The optimization methods examined in this paper operate at different levels of the Iceberg read path. Partitioning reduces the initial search space. Compaction controls file count and size. Clustering improves value locality so file-level statistics can eliminate more data. Snapshot, manifest and orphan-file maintenance control metadata growth and reclaim storage. Delete-file maintenance addresses the additional read work introduced by row-level mutations.

These operations also involve tradeoffs. Larger files reduce file-opening overhead but may reduce parallelism or increase the amount of data read for selective queries. Fine-grained partitioning can improve pruning while creating fragmented partitions and more small files. Similarly, clustering can reduce scan volume, but requires an expensive rewrite and must be aligned with actual query patterns. Aggressive snapshot expiration can reduce storage costs while shortening the available history for time travel and rollback.

Keeping an Iceberg lakehouse fast is therefore not a one-time configuration exercise. It requires observing how each table changes, identifying the dominant source of overhead and applying the appropriate optimization only where its expected benefit exceeds its execution cost. The following sections examine these techniques individually, beginning with partitioning and partition pruning.

Chapter 2

Partitioning and Partition Pruning

Partitioning is the first level at which an Iceberg table can eliminate data during query planning. Instead of evaluating every data file in a table, the query engine uses the partition specification and metadata stored in manifests to identify groups of files that cannot satisfy the query predicate. When the partitioning strategy matches common access patterns, this can substantially reduce both planning work and the amount of data read.

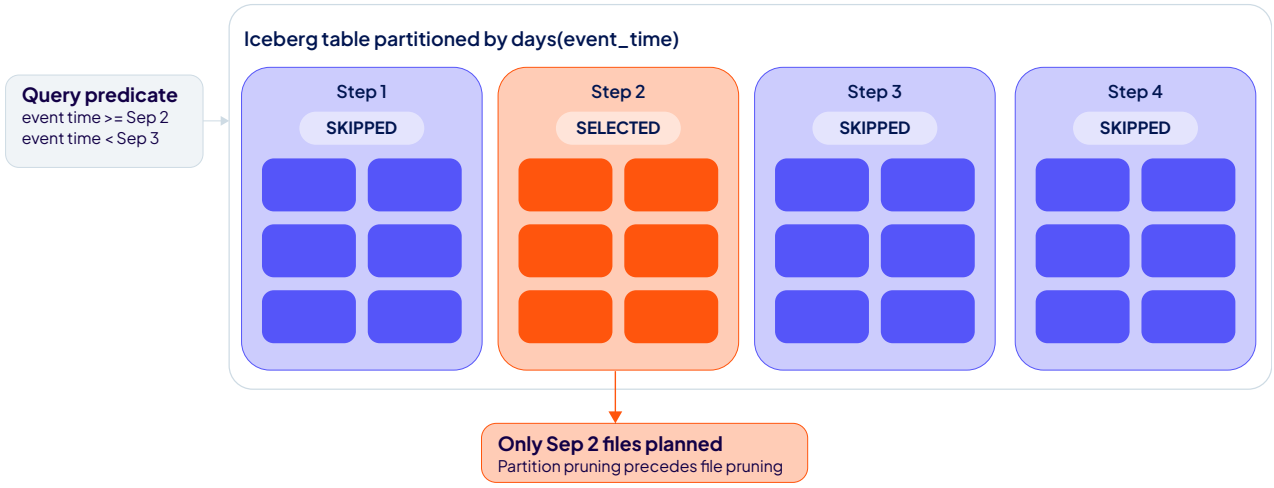
Traditional Hive-style tables encode partition values in directory paths such as:

```
/events/year=2026/month=09/day=16/
```

This approach exposes the physical layout to users. Queries may need explicit predicates on the partition columns, even when those columns are derived from another field. A table partitioned by `event_date`, for example, may not prune correctly when a query filters only on the original `event_time` timestamp. Partition changes may also require changes to directory layouts and query logic.

Partition pruning in Apache Iceberg

A compatible predicate eliminates complete partition groups before file-level scanning



Iceberg instead uses hidden partitioning. The table's partition specification defines how source-column values are transformed, and Iceberg stores the resulting partition values in table metadata. Users continue to query the source columns without needing to know how the data is physically partitioned.

Consider an events table partitioned by month:

```
CREATE TABLE catalog.analytics.
events (
    event_id BIGINT,
    event_time TIMESTAMP,
    event_type STRING,
    payload STRING
)
USING iceberg
PARTITIONED BY (month(event_time));
```

An application can query the original timestamp column:

```
SELECT event_id, event_type
FROM catalog.analytics.events
WHERE event_time >= TIMESTAMP '2026-
09-01 00:00:00'
  AND event_time <  TIMESTAMP '2026-
10-01 00:00:00';
```

During planning, Iceberg projects the predicate on `event_time` onto the corresponding month partition values. Files belonging only to other months can be eliminated before their contents are read. The application does not need to add a separate predicate such as `event_month = '2026-09'`.

Iceberg supports several partition transforms for different data distributions and query patterns:

- `identity` preserves the source value and is useful for suitable low-cardinality columns.
- `year`, `month`, `day` and `hour` derive temporal partition values.
- `bucket` hashes values into a fixed number of buckets and can distribute high-cardinality values such as customer or device identifiers.
- `truncate` groups values by a specified width, such as ranges of integers or string prefixes.

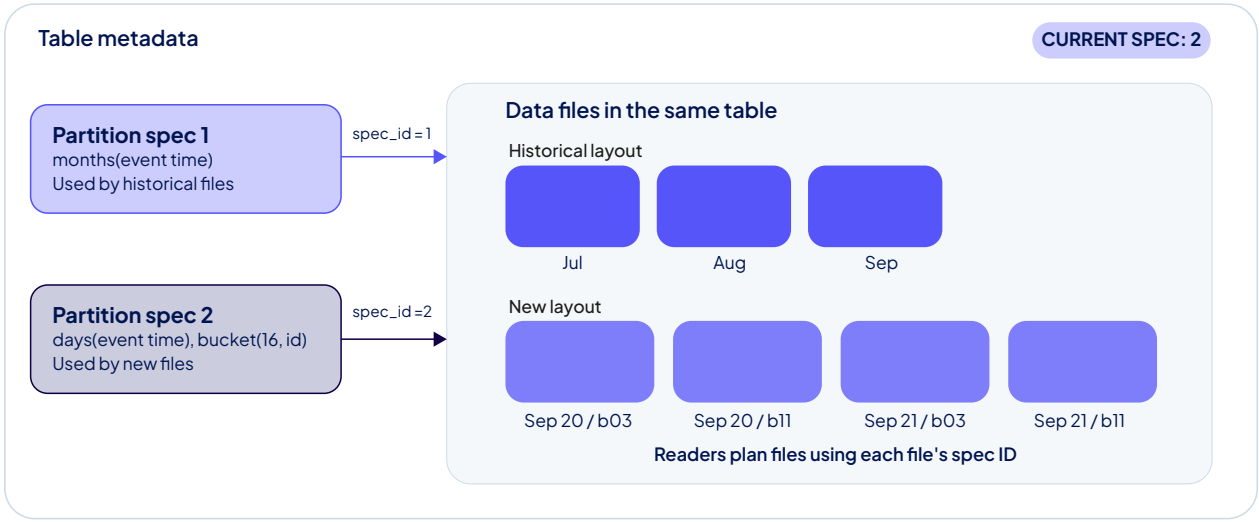
Choosing the transform is a balance between pruning precision and data fragmentation. Partitioning an event table by `month` may work well when queries typically read several days or weeks of data. If most queries access only a few hours, monthly partitions may be too coarse and leave too many candidate files. Hourly partitioning may improve pruning, but it can create many low-volume partitions and increase the likelihood of small files. Similarly, applying identity partitioning to a highly cardinal column can produce a large number of sparsely populated partitions. Partitioning should therefore reflect both query predicates and the volume of data written into each resulting partition. A useful partition field is one that regularly eliminates meaningful amounts of data while still receiving enough records to produce efficiently sized files.

Iceberg also supports partition evolution. A table partitioned by month can later move to daily partitioning, add a bucket transform, or remove an existing partition field without rewriting existing data files. Each file remains associated with the partition specification under which it was written. During query planning, Iceberg evaluates each partition layout separately using the filter derived for that specification. New writes use the current specification, while existing files retain their original layout. Iceberg's partition evolution model makes this a metadata-only change and it does not physically reorganize historical data.

This distinction matters operationally. Changing the partition specification improves the partition layout of future writes, but it does not retroactively change the partition-pruning granularity of existing files. Historical data must be rewritten if it also needs to adopt the new layout. Partition evolution and data rewriting are therefore related, but separate, operations.

Partition evolution without rewriting existing files

Files written under different partition specifications coexist in one Iceberg table



Designing an Effective Partitioning Strategy

In this section, we provide practical guidance for designing an effective partitioning strategy based on query patterns, partition granularity, column cardinality, data skew and how the table is expected to evolve over time.

Start with Query Patterns

A good partitioning strategy starts with the queries that matter most, particularly the predicates used by frequent or expensive workloads. A column is worth including in the partition specification only when filtering on it regularly eliminates a meaningful portion of the table. The fact that a column appears frequently in queries is not sufficient. If its predicates are not selective, partitioning by it may add complexity without reducing the number of files that Iceberg needs to consider.

Select the Transform and Granularity

The next decision is the transform and its granularity. For time-based data, this usually means choosing between hourly, daily, monthly or yearly partitions. The answer depends on the relationship between the typical query window and the volume of data written during that interval. If most queries read a few days of data, monthly partitioning may be too coarse because each query still selects a large amount of unrelated data. Daily partitioning would provide more precise pruning. Hourly partitioning may narrow the search further, but only makes sense if each hour receives enough data to produce efficiently sized files. Otherwise, the table gains more partitions while also creating a small-file problem.

The following specifications illustrate how the same timestamp column can be partitioned at different levels of granularity:

```
-- Monthly partitions
PARTITIONED BY (month(event_time));
-- Daily partitions
PARTITIONED BY (day(event_time));
-- Hourly partitions
PARTITIONED BY (hour(event_time));
```

Moving from monthly to hourly partitioning increases pruning precision, but also increases the number of partitions that receive data. The finer specification is useful only when the write volume is sufficient to produce well-sized files within each interval.

Cardinality needs similar consideration. Identity partitioning is suitable for columns with a limited and reasonably stable set of values, such as region or business unit. It is less suitable for fields such as `customer_id`, `device_id` or `transaction_id`, where the number of distinct values may be very high. A bucket transform is often a better option in this case because it distributes the values across a fixed number of partitions rather than creating one partition for every value. This works well for equality and IN predicates, but offers little value for range filters because hashing does not preserve the original ordering of the values.

For example, identity partitioning on `customer_id` creates a distinct partition for every customer:

```
PARTITIONED BY (customer_id);
```

Bucketing limits the number of partition values by hashing each customer identifier into a fixed number of buckets:

```
PARTITIONED BY (bucket(32, customer_id));
```

The number of buckets should be selected using the table's data volume and expected write distribution. Too few buckets can produce large partitions, while too many can recreate the fragmentation that bucketing is intended to avoid.

Account for Combined Fields and Data Skew

Multiple partition fields should be evaluated as a combined layout. A specification such as `day(event_time)` and `region` may be effective when queries frequently filter on both fields and every region receives enough daily data. The same design performs poorly when some regions receive very little data, since each day is divided into several sparsely populated partitions. Looking at the cardinality of each field independently can therefore be misleading. What matters is the volume and file distribution produced by each combination of partition values.

The combined specification described above can be defined as follows:

```
PARTITIONED BY (
    day(event_time),
    region
);
```

This creates partitions for each observed combination of day and region. Before using such a specification, the expected data volume should be evaluated at that combined level. Ten daily partitions and five region values do not necessarily produce fifteen partitions; they can produce as many as fifty day-region combinations.

Skew can make this harder to identify during initial design. A field may have a manageable number of values while one of those values accounts for most of the table. This creates a small number of very large partitions alongside many low-volume partitions. The larger partitions produce uneven scan work, while the smaller ones are more likely to accumulate undersized files. In that situation, changing the transform, introducing bucketing or removing the skewed field may produce a healthier layout.

Validate the Layout in Production

The final test is production behavior. Query plans should show whether predicates are being projected onto partition fields and how many partitions and files remain after pruning. These results should be considered together with the bytes scanned, file-size distribution within each partition, planning time and the difference between the largest and smallest active partitions. No single metric is enough: strong partition pruning is of limited value if the resulting partitions each contain hundreds of small files.

Iceberg's partitions metadata table can be used to inspect this distribution for the current table snapshot:

```
SELECT
    partition,
    spec_id,
    record_count,
    file_count,
    total_data_file_size_in_bytes,
    ROUND(
        total_data_file_size_in_bytes /
file_count / 1024 / 1024,
        2
    ) AS avg_file_size_mb
FROM catalog.analytics.events.partitions
ORDER BY file_count DESC;
```

The result will show how records and data files are distributed across partitions. Partitions with unusually high file counts may indicate fragmentation, while partitions with low record counts and small average file sizes may indicate that the specification is too granular. The `spec_id` is also useful after partition evolution because it identifies the specification associated with each partition.

Revisit the Strategy as the Table Evolves

Partitioning should also be revisited as the table grows. An initial monthly specification may become too coarse after ingestion volume increases, while new query patterns may justify adding another partition field. Iceberg makes these changes possible through partition evolution, but the benefits initially apply to newly written files. Historical files retain their earlier layout unless they are rewritten. The old and new partition specifications should therefore be measured separately before deciding whether the historical data also needs to be reorganized.

Chapter 3

Compaction and the Small File Problem

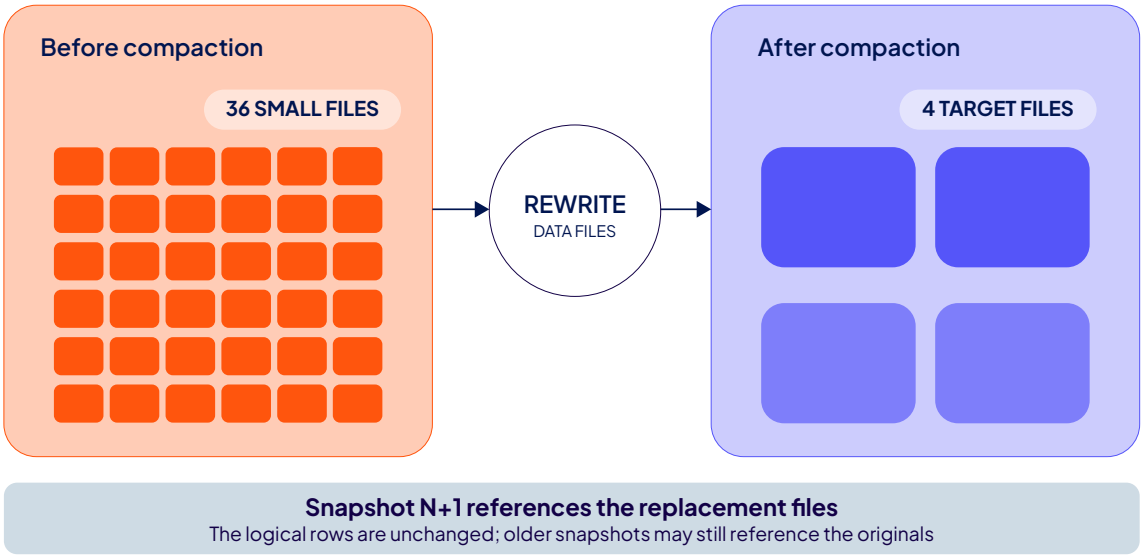
Partitioning can reduce the number of partitions considered by a query, but it does not control the number or size of files within those partitions. A single partition may still contain hundreds or thousands of small data files, particularly when data is written through streaming or frequent micro-batch pipelines.

Each file adds work to the query. Iceberg stores an entry for the file in a manifest, the query engine creates scan tasks for selected files, and the storage layer must process requests to open and read them. When a table contains many small files, a larger share of query time is spent planning, scheduling and performing file-level I/O rather than processing the data itself. Small files may also provide less opportunity for effective compression and sequential reads.

Small files commonly accumulate when the amount of data written during each commit is insufficient to produce well-sized files. They can also result from concurrent writers, uneven partition volumes, or highly granular partitioning. A streaming job that commits every few minutes, for example, may write a small file into each active partition during every checkpoint. An individual commit may add only a few files, but the file count accumulates across hundreds or thousands of commits.

Compaction rewrites files, not logical records

A new snapshot replaces many small data files with fewer target-sized files



Compaction addresses this problem by rewriting a collection of small data files into a smaller number of larger files. In Apache Iceberg, the operation creates replacement files and commits a new snapshot that removes the original files from the table's current state. The logical records remain unchanged, but the physical layout contains fewer files for query engines to plan, schedule, open and read. This reduces file-level overhead while producing files that are better aligned with the table's target size.

Iceberg exposes the `write.target-file-size-bytes` table property to specify the approximate file size that writers should target. The default is 512 MB, although the appropriate value depends on the query engine, storage system and workload.

```
ALTER TABLE catalog.analytics.events
SET TBLPROPERTIES (
    'write.target-file-size-bytes' =
'536870912'
);
```

This property defines a target rather than a guaranteed output size. A writer cannot produce a file larger than the amount of data available to its task, and the final compressed size may differ from the in-memory data used to plan the write. Writer configuration can reduce the rate at which small files are created, but it cannot always prevent them.

Rewriting Data Files

Existing files can be compacted using Iceberg's `rewrite_data_files` Spark procedure. With the default `binpack` strategy, Iceberg selects files that do not meet the configured size criteria, groups them and rewrites their records into fewer, better-sized files.
Here is an example.

```
CALL catalog.system.rewrite_data_files(
    table => 'analytics.events',
    strategy => 'binpack'
);
```

The procedure does not concatenate the existing objects. Iceberg reads the selected files and writes new data files containing the same live records. A successful commit adds the replacement files and removes the original files from the new table snapshot. The original files remain available to older snapshots until those snapshots are expired.

Compaction can also be limited to files that may contain data matching a predicate:

```
CALL catalog.system.rewrite_data_files(
    table => 'analytics.events',
    strategy => 'binpack',
    where => 'event_time >= TIMESTAMP
''2026-09-01 00:00:00'''
);
```

This is useful for continuously ingested tables where recent partitions accumulate small files while older partitions remain stable. Restricting the rewrite to active partitions reduces compute usage and avoids moving data that is already well sized.

The behavior of the rewrite is controlled by several configuration parameters. Five of these are particularly important for managing file-group size, concurrency, output sizing and commit behavior:

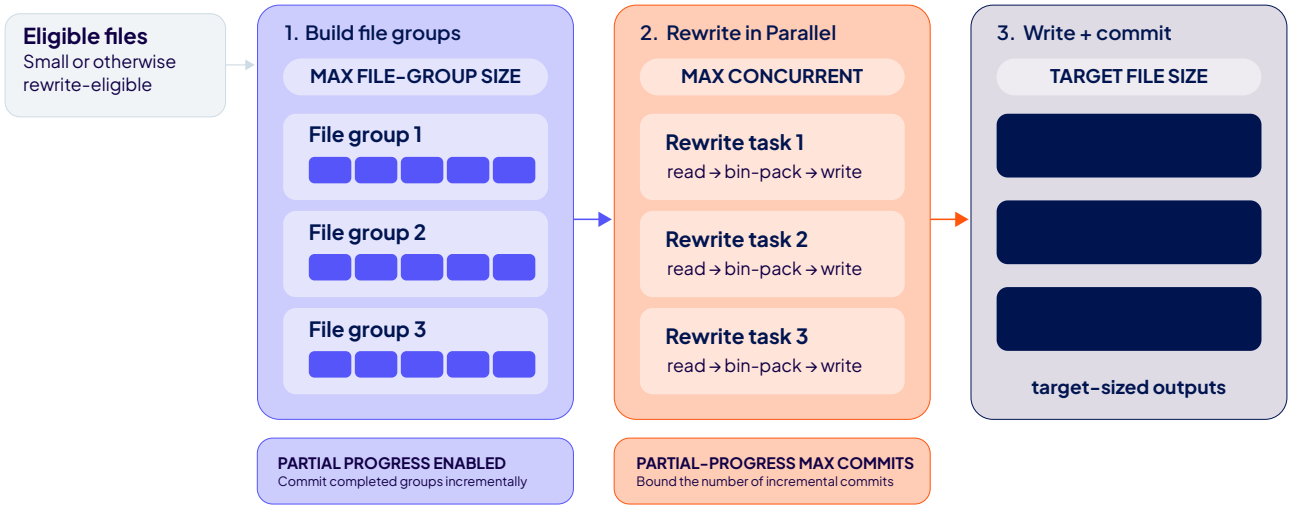
```
CALL catalog.system.rewrite_data_files(
    table => 'analytics.events',
    strategy => 'binpack',
    options => map(
        'max-file-group-size-bytes',
'21474836480',
        'max-concurrent-file-group-
rewrites', '5',
        'target-file-size-bytes',
'536870912',
        'partial-progress.enabled', 'true',
        'partial-progress.max-commits', '10'
    )
);
```

In this example, Iceberg limits each file group to approximately 20 GB of input data, allows up to five groups to be rewritten concurrently, targets 512 MB output files and can publish completed work across a maximum of ten commits.

Let's understand these in detail.

How compaction configuration controls execution

The five settings shape grouping, parallelism, output sizing, and commit behavior



Maximum File-Group Size

`max-file-group-size-bytes` sets the largest amount of input data that Iceberg can assign to one rewrite group. The default is 100 GB.

```
'max-file-group-size-bytes',
'21474836480'   -- 20 GB
```

Iceberg first organizes candidate files by partition. If the eligible files within a partition exceed the configured limit, they are divided into multiple file groups. A file group does not span partitions.

The file-group limit is different from the target output file size. A 20 GB file group does not produce one 20 GB file. It defines the amount of input handled by one independently processed rewrite unit. If the target file size is 512 MB, a 20 GB group would produce approximately 40 output files when the input and output compression ratios are comparable.

File-group size also affects the amount of independent work available to the compaction job. Consider a partition containing 100 GB of files eligible for rewriting. With the default 100 GB limit, the partition may be planned as a single group. Reducing the limit to 20 GB can produce approximately five groups, allowing several groups to be processed at the same time.

The approximate number of groups in a partition can be estimated by dividing the eligible input bytes by the maximum file-group size and rounding up. The actual number may differ because Iceberg must keep individual files intact when assigning them to groups.

A larger group size produces fewer rewrite groups, but each group takes longer to process and creates a larger failure boundary. A smaller value exposes more group-level parallelism and limits the amount of data handled by each rewrite unit. It also increases coordination overhead and may produce more commits when partial progress is enabled.

This setting is not an executor-memory limit. Spark can divide the work within a file group into multiple tasks, shuffle partitions and output files. The parameter controls the logical amount of input assigned to a rewrite group.

Maximum Concurrent File-Group Rewrites

`max-concurrent-file-group-rewrites` limits the number of file groups that Iceberg can rewrite at the same time. The default is five.

```
'max-concurrent-file-group-rewrites',
'5'
```

Actual concurrency is limited by whichever is smaller, i.e., the number of planned file groups or the configured concurrency. If the operation produces only two groups, setting concurrency to ten will not create additional work. If the operation produces twenty groups and concurrency is five, no more than five groups can be active at once.

This is why concurrency should not be tuned in isolation. When a large partition is planned as a single group, increasing `max-concurrent-file-group-rewrites` has no effect. The partition must first be divided into smaller groups using `max-file-group-size-bytes`.

Higher concurrency can reduce the total duration of a rewrite when the cluster has available capacity. It also increases simultaneous reads, shuffle activity, output writes and driver-side coordination. On a shared cluster, an aggressive value can interfere with ingestion or analytical queries. Concurrency should be increased only after file-group sizing has created enough independent work and cluster metrics show that additional groups can run safely.

Target File Size

`target-file-size-bytes` controls the approximate size of the data files produced by the rewrite. If the option is not provided, it inherits the value of the table property `write.target-file-size-bytes`, which defaults to 512 MB.

```
'target-file-size-bytes', '536870912'
-- 512 MB
```

The final files will not always match the target exactly. Spark plans output work using estimated data sizes, while the actual result depends on compression ratio, record width, data distribution and task boundaries.

The target also influences which files are selected for compaction. By default, files smaller than 75 percent of the target are treated as undersized, and files larger than 180 percent are treated as oversized. With a 512 MB target, those thresholds are approximately 384 MB and 922 MB. They can be changed using `min-file-size-bytes` and `max-file-size-bytes`.

A larger target reduces file count and the associated planning and object-store overhead. It may also reduce scan parallelism or cause a selective query to read more data from each chosen file. A smaller target creates more files and more scan tasks, but eventually reintroduces the overhead that compaction is intended to remove.

The target should therefore be selected using the table's partition sizes and query behavior. A partition that contains only 200 MB of data cannot produce a 512 MB file. Similarly, choosing a 1 GB target for partitions that receive small incremental writes will not prevent small files from appearing unless those files are periodically rewritten.

Partial Progress

By default, `partial-progress.enabled` is set to `false`.

```
'partial-progress.enabled', 'false'
```

Iceberg waits for all planned file groups to finish and then commits the rewrite as a single table update. This produces one new snapshot for the operation. If the rewrite fails before the commit succeeds, none of the completed groups are published to the table.

Partial progress changes this behavior:

```
'partial-progress.enabled', 'true'
```

When enabled, Iceberg can commit successfully rewritten groups before the entire operation has finished. If a later group fails, the groups that were already committed remain part of the table. This is useful for large compaction jobs where restarting the entire operation would discard hours of completed work.

Each partial-progress commit is atomic. The table does not expose a group with only some of its replacement files committed. However, the complete compaction job is no longer published through a single all-or-nothing commit. Readers may observe several valid table states as additional groups are completed and committed.

Partial progress also creates more snapshots and performs more catalog commits. This adds metadata and increases the number of opportunities for commit contention with concurrent writers. It is therefore most useful for long-running jobs with many rewrite groups, rather than small compaction operations that can complete reliably in one commit.

Maximum Partial-Progress Commits

`partial-progress.max-commits` sets the maximum number of commits that the rewrite may produce when partial progress is enabled The default is ten.

```
'partial-progress.max-commits', '10'
```

This value does not control the number of file groups. The groups are determined by the eligible files, table partitions and `max-file-group-size-bytes`. The commit limit controls how those groups can be published to the table.

If a rewrite produces 50 file groups and allows a maximum of ten commits, Iceberg must include multiple completed groups in some commits rather than creating one commit per group. The exact batching depends on how the rewrite completes, but the operation cannot exceed the configured commit limit.

A low value reduces snapshot creation and catalog activity, but more completed groups are associated with each commit. A high value publishes progress more frequently and creates smaller recovery boundaries, but increases metadata, snapshot and commit overhead. Very high values can also increase commit contention when ingestion or other maintenance operations are writing to the table. The option has no effect when `partial-progress.enabled` is `false`.

Tuning Compaction for the Workload

Compaction tuning should begin with the files that are eligible for rewriting rather than the size of the available cluster. The important inputs are the number of candidate files, their size distribution, the amount of eligible data in each partition and the rate at which new small files are created. A table-wide average can hide the problem: one active partition may contain thousands of small files while older partitions are already well sized.

Establish the Target File Size

Start with `target-file-size-bytes` The value should be large enough to reduce planning, scheduling and object-store overhead without removing useful scan parallelism.

Tables dominated by broad analytical scans may benefit from larger files. Highly selective queries, small partitions or engines with limited split handling may perform better with a smaller target. The default 512 MB value is a reasonable starting point, but it should not be treated as a standard for every table.

File-size distribution is more useful than average file size when evaluating the result. The median, lower percentiles and number of files below the minimum threshold show whether the table is continuing to produce small files after compaction.

Size the Rewrite Groups

After selecting the target, configure `max-file-group-size-bytes` so that large partitions are divided into manageable units. Dividing the eligible bytes in each partition by the group-size limit gives an estimate of how many groups the operation can produce.

If a large rewrite produces only one group, increasing concurrency or adding executors will not expose more group-level work. Reducing the group-size limit is the relevant adjustment. The value should still be large enough for each group to contain enough input files to produce efficiently sized output files.

Add Concurrency After Groups Exist

Once the operation produces enough groups, increase `max-concurrent-file-group-rewrites` until the maintenance cluster reaches an acceptable level of CPU, memory, shuffle and storage utilization.

A useful starting point is to keep the default concurrency of five, observe resource usage and then increase it gradually. The value should account for ingestion jobs, queries and other maintenance operations sharing the same compute or object store. Faster compaction is not useful if it creates instability elsewhere in the platform.

Use Partial Progress for Large Rewrites

Small and predictable compaction jobs can remain single-commit operations. Partial progress is more useful when the rewrite contains many groups, covers a large amount of data or runs long enough that a late failure would make restarting expensive.

`partial-progress.max-commits` should then be selected based on how much completed work should be protected by each commit. Too few commits create larger batches and delay the publication of completed groups. Too many commits create unnecessary snapshots and place more pressure on the catalog and concurrent writers.

Limit the Rewrite Scope

Compaction should normally target the parts of the table where small files are accumulating. For a continuously ingested table, this often means recent or active partitions:

```
CALL catalog.system.rewrite_data_files(
    table => 'analytics.events',
    strategy => 'binpack',
    where => 'event_time >= TIMESTAMP ''2026-09-01 00:00:00''',
    options => map(
        'max-file-group-size-bytes', '21474836480',
        'max-concurrent-file-group-rewrites', '5',
        'target-file-size-bytes', '536870912',
        'partial-progress.enabled', 'true',
        'partial-progress.max-commits', '10'
    )
);
```

Limiting the rewrite prevents stable historical partitions from being evaluated and rewritten repeatedly. It also makes the resource requirements and duration of each maintenance run easier to predict.

Validate the Result

The `rewrite_data_files` procedure returns the number of rewritten data files, added data files, rewritten bytes and failed data files. These values show whether the operation reduced file count and whether any groups failed when partial progress was enabled.

The compaction should also be evaluated using query metrics. Relevant measurements include planning time, files selected, scan tasks created, bytes read and end-to-end query latency. On the maintenance side, track job duration, shuffle volume, peak resource usage, commit failures and the rate at which small files reappear.

A rewrite that finishes successfully has not necessarily produced a useful outcome. The recurring query savings must justify the compute, storage I/O and write amplification required to create the new file layout.

Chapter 4

Managing Delete Files and Read Amplification

Compaction addresses the cost of reading too many data files, but data-file layout is only one source of read amplification in an Iceberg table. Tables that receive frequent deletes, updates, and `MERGE` operations can accumulate delete files that must be considered when queries are planned and executed. A table may therefore have well-sized data files and still perform poorly because readers must open, evaluate, and apply a growing set of deletes.

This behavior is most common when row-level changes are written using merge-on-read. Rather than rewriting an affected data file immediately, merge-on-read records the change separately and defers its application until query time. This reduces the amount of data rewritten by each mutation, but shifts part of the work to readers.

Delete-file maintenance is therefore a separate concern from ordinary data-file compaction. Rewriting position delete files can reduce the number of delete files that must be opened, while rewriting the affected data files can remove the need to apply those deletes during future reads. Choosing between the two requires understanding how the table represents deletes and where the workload is paying the cost.

Copy-on-Write and Merge-on-Read

Iceberg supports two general approaches for row-level modifications: **copy-on-write** and **merge-on-read.** These modes control when the cost of applying a delete, update, or merge is paid.

With copy-on-write, the engine rewrites the affected data files as part of the mutation. Rows that remain valid are written to replacement files, while the original files are removed from the current snapshot. Reads see data files that already reflect the change and do not need to combine those files with newly written delete files. Copy-on-write increases write amplification because changing a small number of rows can require rewriting much larger data files. However, it generally provides a simpler read path, making it appropriate for tables where mutations are relatively infrequent and read performance has a higher priority than mutation latency.

With merge-on-read, the engine can record row-level changes without immediately rewriting the original data files. The existing data files remain in the table, and the deleted rows are described separately. Readers must determine which delete files apply to each data file and suppress the matching rows while scanning.

Merge-on-read reduces the amount of data rewritten by an individual mutation and can improve the latency of frequent `DELETE`, `UPDATE` and `MERGE` operations. The trade-off is additional read-time processing and the need for regular delete maintenance.

Iceberg exposes independent table properties for the three row-level operations:

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

Each property accepts `copy-on-write` or `merge-on-read`. In current Iceberg releases, the default for these properties is `copy-on-write`. Support for a particular mode also depends on the compute engine and connector version, so the table configuration must be evaluated together with the capabilities of the engines that read and write the table.

The choice does not need to be uniform across every operation. For example, a workload may use merge-on-read for a frequent `MERGE` pipeline while leaving infrequent explicit deletes in copy-on-write mode. The appropriate configuration depends on mutation frequency, mutation selectivity, query frequency, and the maintenance capacity available to the platform.

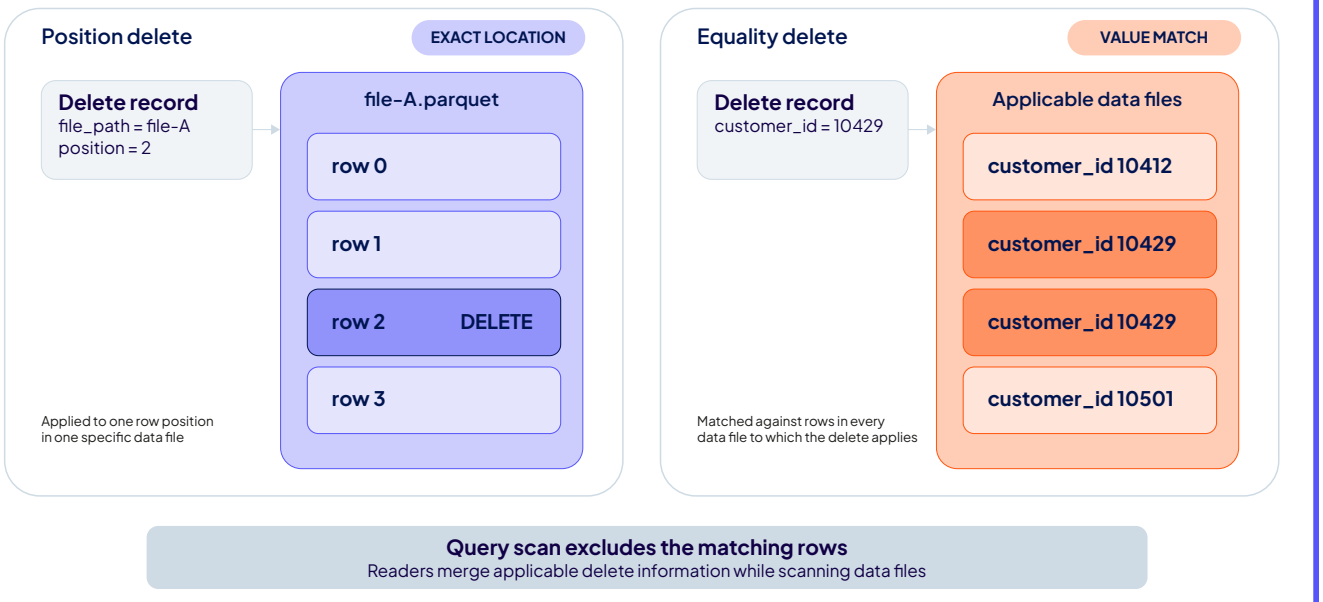
Position Deletes and Equality Deletes

Iceberg format version 2 introduced row-level delete files. These files describe rows that should no longer appear in the current table state even though the referenced data may still be present in a data file.

Iceberg defines two delete representations: position deletes and equality deletes.

How Iceberg applies row-level deletes

Position deletes identify a physical row; equality deletes match one or more field values



Position Deletes

A position delete identifies a row using the path of its data file and its zero-based row position within that file. Conceptually, a position delete contains values such as:

```
data/events/00042.parquet, position 1847

data/events/00042.parquet, position 1848

data/events/00119.parquet, position 63
```

Because a position delete refers to a specific file and row position, it can be applied without comparing business-key columns. The engine can determine that a row at a known position in a known file must be removed from the result.

Position deletes are precise, but their life cycle is connected to the data files they reference. If a data file is replaced by compaction, clustering, or another rewrite, position deletes referencing the old file are no longer needed for the replacement file. Those delete records may remain reachable through table metadata until maintenance operations and snapshot cleanup make the associated files eligible for removal.

Equality Deletes

An equality delete identifies rows using the values of one or more columns. The delete file records the field IDs used for matching and the values that define the deleted rows. A row is deleted when its values match all the equality fields in an applicable equality delete.

For example, an equality delete based on `customer_id` may contain:

```
customer_id = 10429

customer_id = 38117

customer_id = 73052
```

Any applicable data row with one of those values is treated as deleted. Multiple fields may also be used to form a compound equality condition.

Equality deletes are useful when a writer knows the identifying values of the rows being removed but does not know their physical file locations and row positions. The read path is generally more involved because the engine must evaluate the relevant columns from the data against the equality-delete values.

The Iceberg specification uses field IDs rather than column names to define equality fields. This allows delete semantics to remain tied to the logical fields as the schema evolves. Operationally, however, equality-delete columns should still be selected carefully. Columns used for equality deletes should provide stable row identification and should not be altered in a way that makes the intended match ambiguous.

Determining Which Deletes Apply

A query engine does not apply every delete file in the table to every data file. Iceberg uses partition information, sequence numbers, referenced file paths and file-level metrics to determine which delete files may apply.

Position deletes are associated with a particular data-file path. An applicable position delete can affect a data file when the referenced path matches and the sequence and partition requirements defined by the Iceberg specification are satisfied.

Equality deletes are not tied to one physical file. They may apply to compatible data files in the same partition, or more broadly when the equality delete is unpartitioned. Iceberg sequence numbers ensure that an equality delete applies only to data that existed before the delete operation. This prevents the delete from incorrectly removing rows written after the delete was committed merely because those new rows contain the same equality values.

Manifest metadata and file-level statistics allow readers to eliminate delete files that cannot overlap with a planned data scan. However, delete pruning does not eliminate the cost when many delete files legitimately apply to the same data files. Those delete files must still be planned, opened, decoded and evaluated.

How Delete Files Increase Read Cost

Delete files affect both query planning and query execution.

During planning, the engine must read manifest metadata, identify the delete files that may apply and associate those deletes with the corresponding data-file scan tasks. A large number of delete files increases the amount of metadata that must be examined and the number of file relationships that must be planned.

During execution, workers must open the selected delete files and construct the data structures required to apply them. Position-delete entries must be matched to file paths and row positions. Equality-delete values must be compared with the relevant columns in the scanned data. This consumes I/O, memory and CPU in addition to the cost of reading the underlying data files.

The overhead depends on more than the number of deleted rows. Ten thousand deleted rows stored in one reasonably sized delete file may be less expensive to process than the same number of rows spread across hundreds of small delete files. Conversely, a small number of delete files can still be costly when they remove a large fraction of the rows in the associated data files, because the engine continues to read data that will later be discarded.

The following measurements should be considered separately:

- The number of delete files associated with each partition or data file
- The total number of delete records
- The average and distribution of delete-file sizes
- The ratio of deleted rows to physical rows in the affected data files
- The number of data files to which an equality delete may apply
- The rate at which new delete files are created
- The query latency and scan cost of partitions with significant delete activity

These measurements distinguish a delete-file fragmentation problem from a high deletion-density problem. The first can often be improved by rewriting delete files. The second usually requires rewriting the affected data files.

Inspecting Delete Activity

Iceberg metadata tables expose the information required to identify partitions with accumulated deletes. In Spark, the `partitions` metadata table includes data-file counts, record counts and separate statistics for position and equality deletes.

```
SELECT
  partition,
  spec_id,
  record_count,
  file_count,
  position_delete_record_count,
  position_delete_file_count,
  equality_delete_record_count,
  equality_delete_file_count
FROM catalog.analytics.events.partitions
WHERE
  COALESCE(position_delete_file_count, 0) +
  COALESCE(equality_delete_file_count, 0) > 0
ORDER BY
  COALESCE(position_delete_record_count, 0) +
  COALESCE(equality_delete_record_count, 0)
DESC;
```

This query identifies partitions that currently contain delete files and orders them by the number of delete records. Both record counts and file counts matter. A partition with many small delete files may have a different maintenance requirement from a partition with one large delete file containing millions of deleted positions.

The current delete files can be inspected directly through the `delete_files` metadata table:

The `content` value identifies the file type: `1` represents a position delete file, while `2` represents an equality delete file. The `equality_ids` field contains the field IDs used by an equality delete and is null for position delete files.

```
SELECT
  content,
  partition,
  file_path,
  record_count,
  file_size_in_bytes,
  equality_ids
FROM catalog.analytics.events.delete_files
ORDER BY
  partition,
  content,
  file_size_in_bytes;
```

For detailed investigation of position deletes, the `position_deletes` metadata table exposes individual delete records:

```
SELECT
  file_path,
  pos,
  delete_file_path,
  partition,
  spec_id
FROM catalog.analytics.events.position_deletes
WHERE file_path = 's3://lakehouse/analytics/
events/data/00042.parquet'
ORDER BY pos;
```

This is useful for diagnosis, but it can be expensive on tables with a large number of delete records. It should not be used as a routine monitoring query without a selective predicate.

The `partitions` metadata table reports the physical table state represented by data and delete files; it does not apply the deletes to compute a fully materialized live-row count. A partition may therefore appear in the metadata table even when all of its physical rows have been logically deleted.

Rewriting Position Delete Files

When the primary problem is a large number of small position delete files, Iceberg's Spark procedure `rewrite_position_delete_files` can combine them into fewer, larger files.

```
CALL catalog.system.rewrite_position_delete_
files(
  table => 'analytics.events'
);
```

This operation performs delete-file compaction. It reduces file-open overhead, the number of delete-file entries represented in manifests and the number of delete files that readers may need to associate with scan tasks.

The procedure also filters dangling position-delete records that reference data files no longer present in the current table state. Such records may be left behind after the referenced data files have been rewritten. Readers already ignore deletes that do not apply to live data files, but removing them reduces unnecessary metadata and storage.

The target delete-file size can be configured for the rewrite:

```
CALL catalog.system.rewrite_position_delete_
files(
  table => 'analytics.events',
  options => map(
    'target-file-size-bytes', '67108864',
    'min-input-files', '5',
    'max-concurrent-file-group-rewrites', '5'
  )
);
```

In this example, the procedure targets 64 MB delete files and rewrites a file group when it contains at least five eligible input files. The concurrency setting limits the number of file groups that may be rewritten concurrently.

Iceberg also exposes the table property `write.delete.target-file-size-bytes`, whose current default is 64 MB. The rewrite procedure uses this property as its default target when `target-file-size-bytes` is not supplied as a procedure option.

A rewrite can be limited to the portion of the table that requires maintenance:

```
CALL catalog.system.rewrite_position_delete_files(
  table => 'analytics.events',
  where => 'event_date >= DATE ''2026-09-01''',
  options => map(
    'target-file-size-bytes', '67108864',
    'min-input-files', '5'
  )
);
```

Scoping the operation is useful when recent partitions receive most of the updates or when rewriting the entire table would exceed the available maintenance window.

Rewriting position delete files does not remove the underlying logical deletes. It changes their physical organization. Queries must still apply the resulting delete files to the associated data files. The operation is therefore most useful when delete-file count and file-open overhead are the dominant problems but the deleted-row ratio remains relatively low.

The procedure operates on position delete files. It does not provide equivalent compaction for equality delete files.

Applying Deletes Through Data-File Rewrites

When a large fraction of a data file has been deleted, compacting its delete files is insufficient. The query engine still reads the original data file and discards the deleted rows during execution. If 40 percent of a file's rows are deleted, the table continues to store and scan the physical representation of those rows until the data file is rewritten.

The `rewrite_data_files` procedure can select data files based on their associated deletes:

```
CALL catalog.system.rewrite_data_files(
  table => 'analytics.events',
  strategy => 'binpack',
  options => map(
    'delete-file-threshold', '5',
    'delete-ratio-threshold', '0.20'
  )
);
```

The `delete-file-threshold` option selects a data file when the number of associated deletes reaches the configured threshold. The `delete-ratio-threshold` option selects a file when the estimated ratio of deleted rows to total rows reaches the configured value. A value of `0.20` represents a deleted-row ratio of 20 percent.

Files meeting either delete-based criterion may be selected even when their physical size would not otherwise make them candidates for compaction. The procedure may also select files for the normal size and file-group conditions configured for the chosen rewrite strategy.

During the rewrite, the engine reads the affected data files, applies the relevant deletes and writes the remaining live rows to replacement files. Future queries read the replacement files without applying the accumulated deletes that were materialized by that rewrite. This reduces both the amount of obsolete data scanned and the number of delete relationships that must be processed.

A data-file rewrite requires more I/O and compute than delete-file compaction because it reads and rewrites the table's data. It may also overlap with active writers and produce commit conflicts if the same files are modified concurrently. For this reason, the operation should normally be scoped to partitions or file groups where delete density justifies the rewrite.

```
CALL catalog.system.rewrite_data_files(
  table => 'analytics.events',
  strategy => 'binpack',
  where => 'event_date >= DATE ''2026-09-01''',
  options => map(
    'delete-file-threshold', '5',
    'delete-ratio-threshold', '0.20',
    'partial-progress.enabled', 'true',
    'partial-progress.max-commits', '10'
  )
);
```

After a data-file rewrite, position delete files may still contain records referencing data files that are no longer part of the current snapshot. Running `rewrite_position_delete_files` afterward can remove those dangling records and compact the remaining position deletes.

Data files and delete files removed from the current snapshot may still be referenced by older snapshots. They are not physically eligible for deletion solely because a rewrite has completed. Snapshot expiration and orphan-file cleanup are separate maintenance operations and must be handled according to the table's retention requirements.

Deletion Vectors in Iceberg v3

Iceberg format version 3 introduces deletion vectors as a more compact representation of position-based deletes. A deletion vector is associated with one data file and records its deleted row positions using a bitmap representation stored in a Puffin file.

A data file can have at most one applicable deletion vector in a snapshot. When additional positions are deleted, a writer must merge the new positions with the existing deletion vector and produce a replacement vector. This avoids accumulating multiple independent position delete files for the same data file.

When a deletion vector exists for a data file, it replaces the prior position-delete representation for that file. Readers can use the bitmap to determine which row positions should be suppressed. This can reduce delete metadata and improve the efficiency of checking deleted positions. Iceberg v3 deprecates the creation of new position delete files, although position delete files written before a table upgrade remain valid. Equality delete files continue to be part of the format.

Deletion vectors require support from every engine and connector that reads or writes the table. Upgrading the table format alone does not provide an operational benefit if the engines in the environment cannot consume or maintain the new representation. Compatibility should therefore be verified across the complete set of production engines before adopting v3.

Tuning Delete Maintenance for the Workload

Delete maintenance should be based on whether the workload is suffering from delete-file fragmentation, high deletion density or both.

Consolidating Fragmented Position Delete Files

For tables with many small position delete files but a relatively small number of deleted rows, begin with `rewrite_position_delete_files`. This consolidates the delete metadata without paying the cost of rewriting the underlying data. It is particularly useful for streaming or micro-batch pipelines that issue frequent commits and create small delete files in each active partition.

Rewriting Data with a High Deleted-Row Ratio

For data files with a high deleted-row ratio, use `rewrite_data_files`. Rewriting the data is more expensive, but it prevents future queries from repeatedly scanning rows that will be discarded. The threshold should reflect the relative cost of maintenance I/O and recurring query I/O. A heavily queried table may justify rewriting at a lower delete ratio than an infrequently queried archival table.

Managing Equality Deletes

For equality-delete-heavy tables, monitor how broadly each delete file applies. An unpartitioned equality delete can potentially be considered against data across the table, while a partitioned equality delete has a narrower scope. Because `rewrite_position_delete_files` does not rewrite equality deletes, persistent equality-delete overhead generally requires rewriting the affected data files or using maintenance capabilities provided by the specific execution platform.

Aligning Maintenance with the Write Strategy

For workloads with frequent, latency-sensitive mutations, merge-on-read may remain the correct write strategy, but it should be paired with a predictable maintenance schedule. Maintenance should run often enough to prevent delete files from accumulating faster than they can be processed. The relevant interval may be hourly for a high-volume change-data-capture pipeline or daily for a lower-volume analytical table.

For read-intensive tables with occasional mutations, copy-on-write may be more economical. Paying the rewrite cost when the mutation occurs can avoid repeated delete-processing costs across a much larger number of subsequent queries.

Scoping Maintenance to Active Data

Recent partitions should usually be evaluated separately from historical partitions. Recent data often receives updates and late-arriving corrections, while older data is comparatively stable. Predicate-scoped maintenance allows the active portion of the table to be rewritten more frequently without repeatedly scanning stable history.

Validating Maintenance Thresholds

No delete-file count or deleted-row ratio is universally correct. Initial thresholds should be treated as operating assumptions and validated against observed behavior. At minimum, compare the following before and after maintenance:

- Position and equality delete-file counts
- Total delete-record counts
- Delete files associated with each data file or partition
- Deleted-row ratios for rewritten data files
- Planning latency
- Files and bytes scanned by representative queries
- Query execution time
- Bytes rewritten by maintenance
- Maintenance duration and commit-conflict rate

The objective is not to eliminate every delete file immediately. It is to keep the read-time cost of applying deletes below the point where recurring query overhead exceeds the cost and operational risk of maintenance.

Chapter 5

Clustering Data

Partitioning allows a query engine to eliminate broad sections of a table, but it does not determine how records are organized across the data files within each partition. As a partition grows, it may contain hundreds or thousands of files whose contents overlap across the columns used by queries.

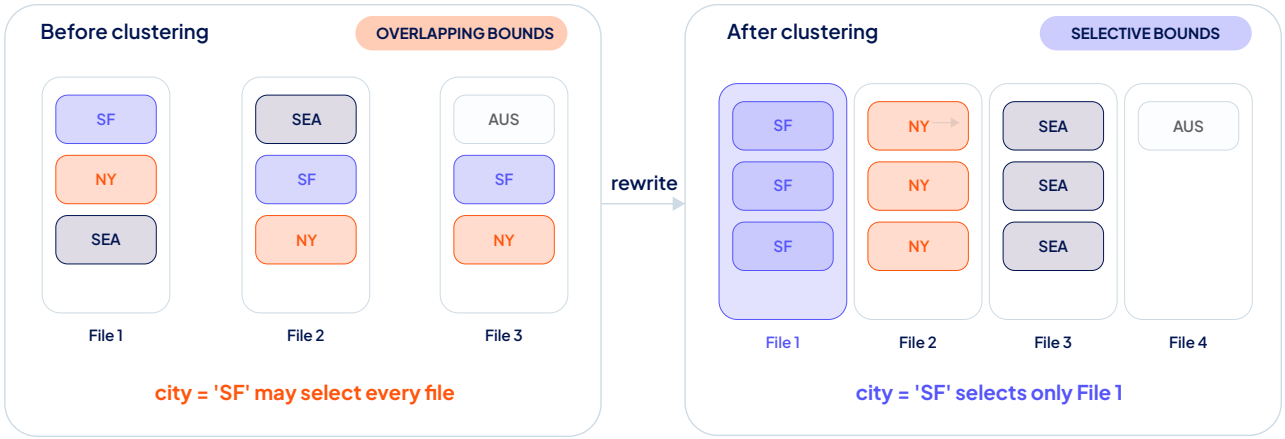
This commonly results from a mismatch between ingestion order and query access patterns. Data is usually written according to when it arrives and how write tasks are distributed. Records arriving in the same batch may be placed in the same file even when their values for commonly filtered columns, such as `customer_id, city, device_id`, or `event_time` are unrelated. Over time, values from the same logical group become distributed across many files.

Consider a table in which records for San Francisco, New York, Seattle, and Austin are mixed across several data files. A query with the predicate `city = 'San Francisco'` may need to read every file whose recorded value range could contain San Francisco, even if only a small portion of each file matches. The files may be correctly sized and stored in the appropriate partition, but their data distribution provides little opportunity for file pruning.

Clustering addresses this problem by rewriting data so that records with similar values are stored close together. After clustering by `city`, San Francisco records may be concentrated into one file while records for other cities are written to separate files. The same query can then eliminate files whose column-level statistics show that they cannot contain the requested value.

Clustering creates more selective file bounds

Grouping similar values reduces overlap and allows file-level metrics to prune more work



The benefit comes from reducing overlap between the lower and upper bounds recorded for individual data files. Tighter, less-overlapping bounds allow Iceberg and its query engines to exclude more files during planning, reducing the number of scan tasks, the volume of data read and the compute required to answer the query.

Clustering is therefore complementary to partitioning rather than a replacement for it. Partitioning provides coarse-grained pruning across groups of files, while clustering improves fine-grained pruning within those groups.

How Clustering Improves File Pruning

Iceberg manifest files contain an entry for every data file tracked by the manifest. In addition to the file path and partition values, an entry can include column-level metrics such as value counts, null counts, lower bounds and upper bounds.

During query planning, Iceberg first uses the partition specification to eliminate files from partitions that cannot satisfy the query. It can then evaluate file-level metrics to eliminate additional data files.

Assume that a table contains four data files with the following bounds for the `city` column:

DATA FILE	LOWER BOUND	UPPER BOUND
File 1	Austin	Seattle
File 2	Austin	Seattle
File 3	New York	Seattle
File 4	Austin	San Francisco

For a query filtering on `city = 'San Francisco'`, the recorded bounds do not prove that any of these files can be skipped. San Francisco falls within each range, even if some files do not actually contain a San Francisco record.

After clustering, the bounds may look different:

DATA FILE	LOWER BOUND	UPPER BOUND
File 1	Austin	Austin
File 2	New York	New York
File 3	San Francisco	San Francisco
File 4	Seattle	Seattle

The same predicate now selects only **File 3**. The query engine avoids scheduling scan tasks for the remaining files.

Lower and upper bounds are not indexes that identify every matching row. They are file-level summaries used to prove that a file cannot contain a match. Clustering makes those summaries more selective by reducing the range of values represented in each file.

The resulting improvement depends on the predicate, the clustering columns, the number of files, and the amount of overlap between their bounds. A table clustered by `city` may provide effective pruning for city predicates while offering little additional benefit for a predicate on an unrelated column such as `device_type`.

Partitioning and Clustering Solve Different Problems

As we learned, partitioning separates records into logical groups represented in Iceberg metadata. A table partitioned by `days(event_time)`, for example, groups records by event date. Queries containing compatible time predicates can eliminate complete partitions.

Clustering controls the arrangement of records inside those partitions. The same table could be partitioned by day and clustered by `customer_id`, producing files that contain narrower ranges of customer identifiers within each daily partition.

This design supports queries such as:

```
SELECT *
FROM catalog.analytics.events
WHERE event_time >= TIMESTAMP
'2026-09-01 00:00:00'
  AND event_time <  TIMESTAMP
'2026-09-02 00:00:00'
  AND customer_id = 10429;
```

The time predicate can first eliminate unrelated daily partitions. Within the selected partition, the `customer_id` predicate can eliminate files whose lower and upper bounds do not contain `10429`.

Using `customer_id` directly as a partition field could create an excessive number of small partitions when the column has high cardinality. Clustering provides locality for the column without creating one partition for every customer.

A sound layout generally uses partitioning for stable, coarse-grained elimination and clustering for selective predicates within those partitions. If a partition specification is too granular, clustering does not solve the resulting small-file and metadata problems. If a partition specification is too broad, clustering may improve file pruning but can still leave maintenance jobs operating across very large partitions.

Sort Orders in Apache Iceberg

Iceberg represents an ordered layout through a sort order stored in table metadata. A sort order contains one or more sort fields. Each field identifies a source column, an optional transform, a sort direction and the placement of null values.

The order of the fields is significant. Given the following sort order:

```
event_time ASC NULLS LAST,

customer_id ASC NULLS LAST
```

records are ordered first by `event_time`. Records with the same or nearby event-time values are then ordered by `customer_id`.

In Spark, a table sort order can be configured with `WRITE ORDERED BY`:

```
ALTER TABLE catalog.analytics.events
WRITE ORDERED BY
  event_time ASC NULLS LAST,
  customer_id ASC NULLS LAST;
```

This operation changes the table's default write order. It is a metadata change and does not rewrite existing data files. Writers that honor the Iceberg sort order can use it for subsequent writes. The precise behavior depends on the engine and write operation and the presence of a sort order in table metadata does not by itself guarantee that every engine will physically order incoming data.

The sort order also does not change the order in which query results are returned. SQL queries still require an explicit `ORDER BY` when ordered output is needed. The table sort order describes physical write organization for data-layout purposes.

Iceberg tracks sort orders using identifiers. Data files can record the sort order under which they were written, which allows files created under different layouts to coexist in the same table. Changing the table's default sort order does not invalidate older files.

A sort order can be removed for future writes with:

```
ALTER TABLE catalog.analytics.events
WRITE UNORDERED;
```

Note that removing or replacing the default order does not reorganize files that have already been written.

Write Distribution and Maintaining the Sort Order

Writing clustered data generally requires moving records between processing tasks. In Spark, Iceberg uses the `write.distribution-mode` table property to request how incoming rows should be distributed before files are written.

The supported modes are `none`, `hash` and `range`.

- **hash** distributes rows using their partition values. It is the current default when a table does not define a sort order. Hash distribution helps place rows for the same Iceberg partition into the same Spark task, but it does not globally order the data by non-partition clustering columns.
- **range** samples the input and uses partition and sort values to assign ranges to Spark tasks. Each task receives an exclusive range of values, providing the global ordering required to produce files with more distinct value ranges. Iceberg uses range distribution by default in Spark when a table has a sort order.

The distribution mode can also be set directly:

```
ALTER TABLE catalog.analytics.events
SET TBLPROPERTIES (
  'write.distribution-mode' = 'range'
);
```

Range distribution is more expensive than hash distribution because it requires sampling and a range-based shuffle. The additional write cost can be justified when the resulting layout materially reduces the cost of recurring reads.

Spark's Adaptive Query Execution can split or combine shuffle partitions after the exchange. This behavior affects the amount of data handled by each task and, consequently, the number and size of the output files. The target Iceberg file size is not sufficient by itself to guarantee well-sized clustered files; Spark task sizing and the difference between in-memory row size and compressed on-disk size also affect the result.

Lexicographic Sorting

The standard Iceberg sort order is lexicographic. With multiple sort fields, the first field has the strongest influence on the physical arrangement of the data. Each subsequent field orders records only within equal or nearby values of the preceding fields.

Consider the following order:

```
ALTER TABLE catalog.analytics.events
WRITE ORDERED BY
  event_date ASC NULLS LAST,
  customer_id ASC NULLS LAST,
  event_type ASC NULLS LAST;
```

This layout first groups records by `event_date`. Within each date, it orders records by `customer_id`. Records for the same date and customer are then ordered by `event_type`.

A query filtering on `event_date` can benefit strongly because date ranges are physically separated. A query filtering on both `event_date` and `customer_id` may benefit further. A query filtering only on `event_type` may see limited improvement because the values for `event_type` repeat across the preceding date and customer ranges.

Column order should therefore reflect predicate hierarchy, not simply the number of times each column appears in a query. If most queries first restrict time and then select a customer, ordering by time followed by customer is reasonable. If most point lookups provide a customer identifier but no time range, placing time first may scatter each customer's records across many files.

Lexicographic sorting is a strong choice when workloads share a consistent leading predicate. It is predictable, represented directly in Iceberg's sort-order metadata and can be maintained by writers that honor the table's default order.

It is less effective when queries filter on different combinations of several columns and no column is consistently the leading predicate. In that case, a multidimensional clustering method such as Z-ordering may provide a better compromise.

Z-Ordering

Z-ordering maps values from multiple columns into a single multidimensional ordering. Rather than completely ordering the data by the first column before considering the second, the algorithm interleaves representations of values from the selected columns. Records that are close across several dimensions are more likely to be written near one another.

Consider a workload that uses combinations of these predicates:

```
customer_id = ?
device_id = ?
region = ?
customer_id = ? AND region = ?
device_id = ? AND region = ?
```

A lexicographic sort on `(customer_id, device_id, region)` primarily optimizes access by `customer_id`. A query filtering only on `region` may still encounter broad and overlapping region ranges across many files.

Z-ordering on the three columns attempts to preserve locality across all three dimensions. It does not produce a complete conventional ordering for any single column, but it can reduce file overlap for queries that use different subsets of the chosen columns.

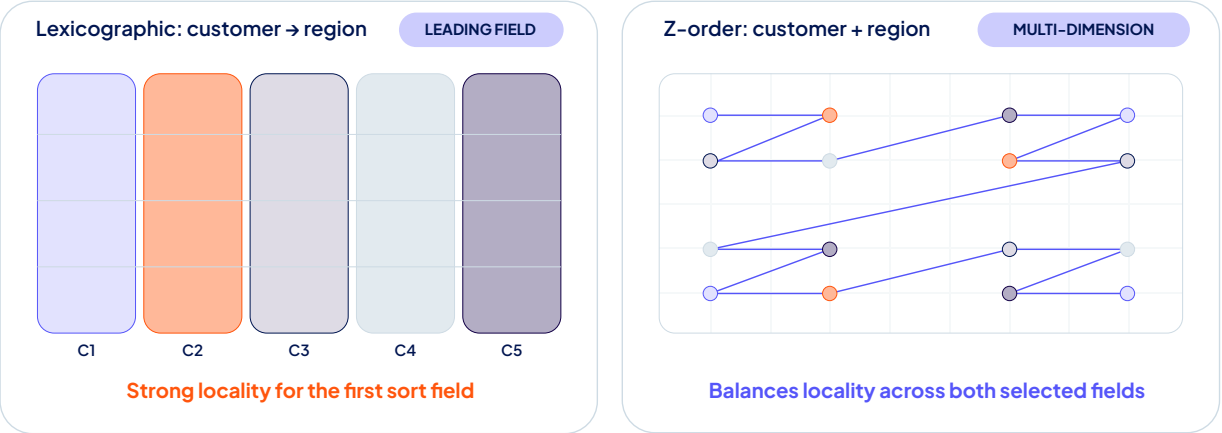
In Iceberg's Spark procedures, Z-ordering is performed through the `sort` rewrite strategy:

```
CALL catalog.system.rewrite_data_files(
  table => 'analytics.events',
  strategy => 'sort',
sort_order => 'zorder(customer_id, device_id, region)'
);
```

Z-ordering is not universally better than lexicographic sorting. Its effectiveness declines as more columns are added because a finite ordering must preserve locality across an increasing number of dimensions. Columns with weak query relevance consume part of that ordering without providing a corresponding pruning benefit. The selected columns should therefore come from observed filter patterns. A narrow Z-order expression over two or three frequently filtered columns will usually be easier to justify than an expression containing every potentially useful dimension.

Lexicographic sorting vs Z-ordering

The same two-dimensional data is organized for different predicate patterns



CLOUDERA

The implementation also exposes two Z-order-specific controls.

`var-length-contribution` defines how many bytes are considered for variable-length values such as strings and binary data. Its current default is eight bytes.

`max-output-size` limits the number of bytes interleaved by the Z-order algorithm and currently defaults to `2147483647`.

These are implementation-level controls rather than general performance targets. They should not be changed without evidence that the representation of the selected columns is limiting clustering quality or creating unacceptable processing overhead.

Unlike a standard Iceberg sort order declared with `WRITE ORDERED BY`, a Z-order expression is supplied to the Spark rewrite procedure. Maintaining that layout as new data arrives therefore requires periodic Z-order rewrites or equivalent support from the execution platform.

Clustering Existing Data

Defining a table sort order affects future writes that honor the order. It does not reorganize data files already present in the table. Existing files must be rewritten before historical data benefits from the new layout.

The `rewrite_data_files` procedure supports a `sort` strategy for this purpose:

```
CALL catalog.system.rewrite_data_files(
  table => 'analytics.events',
  strategy => 'sort',
  sort_order =>
    'event_time ASC NULLS LAST,customer_id
ASC NULLS LAST'
);
```

If `sort_order` is omitted, the procedure uses the table's configured sort order:

```
ALTER TABLE catalog.analytics.events
WRITE ORDERED BY
  event_time ASC NULLS LAST,
  customer_id ASC NULLS LAST;

CALL catalog.system.rewrite_data_files(
  table => 'analytics.events',
  strategy => 'sort'
);
```

By default, `rewrite_data_files` does not necessarily rewrite every file. Files are selected using the procedure's size, file-count,and delete-related eligibility criteria. This behavior is useful for normal compaction, but it may leave correctly sized files in their existing layout even when they are poorly clustered.

To force all files within the selected scope to be reorganized, use `rewrite-all:`

```
CALL catalog.system.rewrite_data_files(
  table => 'analytics.events',
  strategy => 'sort',
  sort_order =>
    'event_time ASC NULLS LAST,customer_id
ASC NULLS LAST',
  options => map(
    'rewrite-all', 'true'
  )
);
```

A complete rewrite can be expensive for a large table. The `where` argument can limit the operation to files that may contain matching records:

```
CALL catalog.system.rewrite_data_files(
  table => 'analytics.events',
  strategy => 'sort',
  sort_order =>
    'event_time ASC NULLS LAST,customer_id
ASC NULLS LAST',
  where =>
    'event_time >= TIMESTAMP ''2026-09-01
00:00:00''',
  options => map(
    'rewrite-all', 'true'
  )
);
```

The predicate is used to select files, not individual rows within those files. If a file may contain data matching the predicate, the entire file becomes a rewrite candidate.

Scoping by active partitions is usually preferable when recent data receives most of the writes. Stable historical partitions can retain their existing layout while newer partitions are reorganized more frequently.

Combining Clustering and Compaction

A sorted or Z-ordered rewrite also produces replacement data files. It can therefore address file sizing and data organization in the same operation.

```
CALL catalog.system.rewrite_data_files(
  table => 'analytics.events',
  strategy => 'sort',
  sort_order => 'zorder(customer_id,device_
id,region)',
  where => 'event_date >= DATE ''2026-09-01''',
  options => map(
    'rewrite-all', 'true',
    'target-file-size-bytes', '536870912',
    'max-file-group-size-bytes',
'107374182400',
    'max-concurrent-file-group-rewrites', '5',
    'partial-progress.enabled', 'true',
    'partial-progress.max-commits', '10'
  )
);
```

This operation reorganizes the selected records and targets 512 MB output files. File groups are limited to 100 GB, with up to five groups rewritten concurrently. Partial progress allows completed groups to be committed before the entire operation finishes, subject to the configured commit limit.

Combining the operations avoids reading and writing the same data once for compaction and again for clustering. However, clustering introduces a shuffle and sort phase that ordinary bin-packing does not require. It consumes additional network, memory, local storage, and CPU resources.

The same target-file-size considerations discussed for compaction still apply. Very large files reduce file count but also increase the range of values represented by each file. Even perfectly ordered data may provide limited file-level selectivity if each output file covers a large portion of the clustering-key domain. Smaller files can produce narrower bounds, but they increase metadata, task-scheduling and file-open overhead. File size and clustering granularity must be tuned together.

Sort Rewrite Configuration

Two settings are particularly relevant to sorted rewrites.

`compression-factor` adjusts how Iceberg estimates the amount of output data from the on-disk size of the input files. The default value is `1.0`. Because Spark sorts an uncompressed in-memory representation while the source and output files are stored in compressed columnar formats, input file size may not accurately predict the number of output files required.

A value greater than `1.0` causes the rewrite to plan more shuffle partitions and potentially more output files. A value below `1.0` plans fewer. The correct value depends on the compression ratio and data representation of the table.

`shuffle-partitions-per-file` controls how many shuffle partitions are used for each expected output file. Its current default is 1. Iceberg uses a custom coalesce operation to combine the sorted shuffle partitions into individual output files.

An example with both options is:

```
CALL catalog.system.rewrite_data_files(
  table => 'analytics.events',
  strategy => 'sort',
  sort_order =>
    'event_time ASC NULLS LAST,customer_id
ASC NULLS LAST',
  options => map(
    'rewrite-all', 'true',
    'compression-factor', '1.5',
    'shuffle-partitions-per-file', '2'
  )
);
```

These settings affect execution planning rather than the logical choice of clustering columns. They should be adjusted only after examining output file sizes, shuffle behavior, executor memory pressure and task duration.

Clustering Cost and Layout Degradation

Clustering is not a permanent operation. New appends, updates and merges introduce files that may overlap with the ranges of previously clustered files. The rate of degradation depends on how data arrives. Append-only data that naturally follows the clustering key may preserve its layout for a long time. Event data arriving mostly in timestamp order, for example, may already produce relatively distinct time ranges. A full clustering rewrite may offer little improvement.

Random arrivals, late data, backfills and frequent row-level updates cause the layout to deteriorate more quickly. New files may contain values spanning ranges already covered by many existing files, increasing overlap and reducing pruning efficiency.

Running clustering too frequently can waste resources by repeatedly rewriting data whose layout has changed only slightly. Running it too infrequently allows read amplification to accumulate.
The maintenance interval should be determined by the rate at which new overlapping files are created and the query cost attributable to that overlap.

Concurrency is another aspect to be considered. Iceberg uses optimistic concurrency for file rewrites. A clustering job can commit successfully only if the files it plans to replace remain valid under the operation's commit requirements. Workloads that continuously modify the same active partitions increase the likelihood of conflicts. Smaller rewrite scopes, bounded file groups and partial progress can reduce the amount of work lost when a conflict occurs.

Measuring Clustering Effectiveness

A clustering operation should be evaluated using representative query predicates rather than the visual appearance of the layout or the successful completion of the rewrite.

The most direct measurement is the number of data files selected during scan planning. For a stable query, effective clustering should reduce the number of files and bytes that remain after partition and metrics pruning.

Useful before-and-after measurements include:

- Number of data files selected by the query
- Number of files eliminated through partition and metrics pruning
- Total bytes scheduled for scanning
- Number of scan tasks
- Query-planning latency
- Query-execution time
- Output file count and size distribution
- Bytes rewritten by the maintenance operation
- Shuffle volume and maintenance duration
- Time taken for new writes to degrade the layout

File-level bounds can also be inspected through Iceberg's `files` metadata table. The `lower_bounds` and `upper_bounds` fields are maps keyed by Iceberg field ID and stored as binary values. They are valuable for programmatic diagnostics, but they are not directly human-readable without decoding values according to the column type.

For routine evaluation, query plans and engine scan metrics are generally more useful than manually inspecting encoded bounds. The goal is to determine whether representative predicates select fewer files after clustering and whether those savings justify the rewrite cost.

A reduction in execution time is not guaranteed to match the reduction in selected files. Cached data, remote-storage throughput, query concurrency, engine scheduling and downstream operators can all affect runtime. File and byte pruning provide a more direct measure of the physical-layout improvement.

Designing an Effective Clustering Strategy

An effective clustering strategy begins with query behavior rather than the table schema. The objective is to identify where physical data organization can materially reduce the number of files read by important workloads.

Start with Query Predicates

Identify the predicates responsible for the largest or most frequent scans, then determine how many files remain after partition pruning. A column is a useful clustering candidate when it appears regularly in selective filters and its values are spread across many files.

High cardinality alone does not make a column a good clustering key. It may make the column unsuitable for partitioning, but clustering it provides value only when queries frequently restrict that column. A nearly unique event identifier, for example, may produce narrow file ranges but offer little benefit if queries rarely search by that identifier.

Choose Between Lexicographic Sorting and Z-Ordering

Use lexicographic sorting when workloads have a stable predicate hierarchy. If most queries first restrict an event-time range and then filter by customer, sorting by `event_time` followed by `customer_id` gives the layout a clear leading dimension. The column order should match the sequence in which queries normally constrain the data.

Use Z-ordering when important queries filter on different combinations of several columns and no single column consistently serves as the leading predicate. Keep the selected column set narrow. Adding dimensions that are rarely filtered weakens locality for the dimensions that matter.

Account for Correlated Columns

Columns that carry similar information do not necessarily need to appear together in the clustering definition. If `city` strongly determines `region`, clustering on both may provide little additional pruning. The second column consumes part of the ordering without introducing a substantially different access dimension.

Columns that are independent and commonly filtered together are stronger candidates for multidimensional clustering. The decision should be based on whether each additional column helps eliminate a different set of files for representative queries.

Match the Rewrite Scope to Data Volatility

Recent partitions often receive late records, updates, merges, or backfills and may require periodic reclustering. Historical partitions that no longer change should not be repeatedly rewritten.

Use the `where` argument of `rewrite_data_files` to restrict maintenance to the active portion of the table. This reduces rewrite cost, limits conflicts with concurrent writers and avoids consuming resources on data whose layout remains stable.

Coordinate Clustering with Compaction

When files require both resizing and reorganization, use a sorted or Z-ordered rewrite to perform both operations in the same pass. Running bin-packing immediately before a clustering rewrite usually reads and rewrites the same data twice.

This does not mean that every compaction job should perform clustering. Bin-packing remains less expensive because it does not require the same global shuffle and sort. Clustering should be added when the expected reduction in query scans justifies that additional cost.

Balance File Size and Pruning Granularity

Larger files reduce metadata volume, file-open operations and scheduling overhead. However, each large file may cover a wider range of clustering-key values, reducing the precision of file-level pruning.

Smaller files can create narrower value ranges, but an excessive file count introduces its own planning and execution costs. The target file size must therefore balance compaction efficiency with the level of pruning required by the workload. Highly selective point queries may benefit from finer clustering granularity than broad analytical scans.

Preserve the Layout During Writes

Measure how quickly file-range overlap returns after clustering. If pruning effectiveness deteriorates within hours or days, the write path may not be preserving the intended organization.

For a standard Iceberg sort order, configuring the table's write order and using range distribution can help new writes follow the same layout. Z-ordered layouts generally require periodic rewrite operations or equivalent support from the execution platform.

Validate the Cost-Benefit Tradeoff

Compare the number of selected files, bytes scanned, planning time and execution time before and after clustering. These read-side improvements should be evaluated against the bytes rewritten, shuffle volume, maintenance duration and commit-conflict rate.

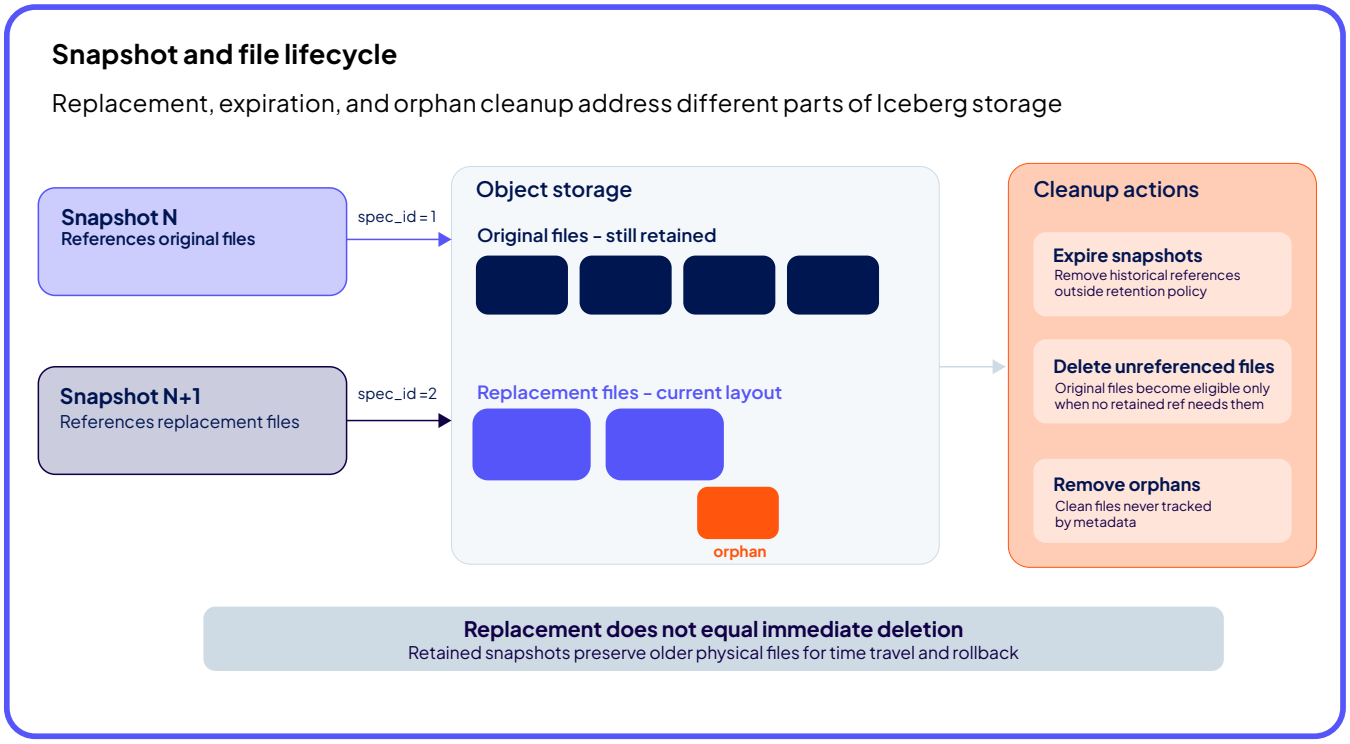
The goal is not to produce a perfectly ordered table. Clustering is justified when the recurring reduction in files and bytes scanned exceeds the compute, I/O and operational cost required to create and maintain the layout.

Chapter 6

Cleaning – Data and Metadata Maintenance

Data-file optimization does not immediately remove the files it replaces. Each append, update, delete, compaction or clustering operation creates a new Iceberg snapshot, while older snapshots continue to reference their previous data and metadata files. This history provides snapshot isolation, time travel and rollback, but it also consumes storage and increases the amount of metadata retained for the table.

A compaction job, for example, may replace hundreds of small files with a smaller set of larger files. The compacted files disappear from the current snapshot, but they cannot be physically deleted while an older snapshot still references them. As a result, the table may temporarily store both the original files and their replacements. Frequent rewrites can produce substantial storage amplification when snapshots are retained indefinitely.



Metadata has a separate lifecycle. Iceberg creates manifest lists, manifest files, and table metadata JSON files as the table changes. The current snapshot may reference more manifests than are efficient for scan planning, while older snapshots and failed writes can leave files that are no longer necessary.

Cleaning therefore serves two distinct purposes. Snapshot expiration and orphan-file removal reclaim storage, while manifest rewriting improves the organization of metadata used to plan current queries. These operations should be managed separately because they have different safety requirements and performance effects.

Expiring Snapshots

An Iceberg snapshot represents the state of a table at a particular point in time. It identifies the manifest list for that state, and the manifest list leads to the data and deletes files required to read the table.

Snapshots are immutable. Each successful table modification creates a new snapshot rather than updating the previous one in place. Older snapshots remain available until they are explicitly expired according to the table's retention policy.

The snapshot history can be inspected through the snapshots metadata table:

```
SELECT
  committed_at,
  snapshot_id,
  parent_id,
  operation,
  manifest_list
FROM catalog.analytics.events.snapshots
ORDER BY committed_at DESC;
```

The `expire_snapshots` procedure removes snapshots that fall outside the configured retention boundary. It also deletes data files, delete files, manifest files, manifest lists and statistics files that are no longer required by any retained snapshot.

```
CALL catalog.system.expire_snapshots(
  table => 'analytics.events',
  older_than => TIMESTAMP
'2026-08-01 00:00:00',
  retain_last => 10,
  stream_results => true
);
```

This example expires eligible snapshots older than August 1, 2026, while retaining at least the ten most recent ancestor snapshots. Setting `stream_results` to `true` returns deletion results to the Spark driver by RDD partition, which is recommended for large cleanup operations to reduce the risk of driver memory pressure.

Snapshot expiration is reference-aware. Iceberg does not delete a file if it remains necessary for a snapshot that has not expired. This includes files shared between old and current snapshots. The operation therefore reclaims only files that are no longer reachable through any retained snapshot.

The default Spark procedure behavior uses a cutoff of five days before the current time and retains at least one ancestor snapshot when explicit values and table expiration properties are not provided. Production tables should use deliberate retention values rather than relying on defaults. The retention period determines how far back users can time travel or roll back, and it also places a lower bound on how quickly replaced files can be removed.

Account for Branches and Tags

Branches and tags are named references to Iceberg snapshots. A tag normally retains a particular historical snapshot, while a branch identifies the head of an independent snapshot lineage. Both can prevent snapshots and their files from becoming eligible for deletion.

The current references can be inspected through the refs metadata table:

```
SELECT
  name,
  type,
  snapshot_id,
  max_reference_age_in_ms,
  min_snapshots_to_keep,
  max_snapshot_age_in_ms
FROM catalog.analytics.events.refs;
```

Before changing a snapshot-expiration policy, verify which branches and tags exist and why they were created. An audit tag may intentionally retain a snapshot for regulatory or recovery purposes. A temporary engineering branch, on the other hand, may remain after its workflow has completed and unintentionally retain a large set of files.

Snapshot expiration will not remove snapshots that are still required by a branch or tag. By default, references do not expire automatically, although retention can be configured when they are created or changed later. The `main` branch itself does not expire.

Retention should therefore be defined at two levels: the history retained along active branches and the lifetime of the references themselves. Expiring snapshots without managing stale references can produce little or no storage reduction because the referenced snapshot lineages remain reachable.

Removing Orphan Files

An orphan file exists under the table's storage location but is not referenced by the table's metadata. Orphans can be created when a distributed write produces files but fails before committing them, when a commit loses a race with another writer or when an interrupted maintenance operation leaves uncommitted output behind.

Snapshot expiration cannot reliably remove these files because they were never part of a committed snapshot, or they are no longer discoverable through the table's tracked metadata. Iceberg provides `remove_orphan_files` to compare the files present in storage with those referenced by the table.

Start with a dry run:

```
CALL catalog.system.remove_orphan_files(
  table => 'analytics.events',
  older_than => TIMESTAMP
'2026-09-01 00:00:00',
  dry_run => true
);
```

The procedure returns the locations it considers orphaned without deleting them. The results should be reviewed before running the destructive version:

```
CALL catalog.system.remove_orphan_files(
  table => 'analytics.events',
  older_than => TIMESTAMP
'2026-09-01 00:00:00',
dry_run => false,
  stream_results => true
);
```

The cutoff is a safety boundary, not simply a storage-retention setting. It must be longer than the maximum expected duration of any write, rewrite or commit operation. If the cutoff is too recent, the procedure may classify files created by an in-progress job as unreferenced and delete them before the job commits. Apache Iceberg's default cutoff is three days.

Path consistency must also be verified. Orphan-file detection compares file locations represented in table metadata with locations returned by the storage system. Differences in URI scheme or authority can cause two paths pointing to the same object to appear different. Iceberg provides prefix-mismatch and path-equivalence controls, but they should not be used to force deletion without first validating the storage configuration. An incorrect path comparison can delete valid table files.

For large object stores, recursively listing the table location may itself be expensive. The procedure can use a supplied file-list view or prefix-based listing when supported by the table's `FileIO` implementation. These alternatives change how candidate files are discovered and they do not change the requirement to compare the candidates safely against table metadata.

Cleaning Table Metadata JSON Files

Iceberg maintains table state in versioned metadata JSON files. Each successful metadata change produces a new file, and the catalog atomically updates the table to point to the latest version.

The current metadata file contains a metadata log that tracks a limited number of previous metadata files. The number retained in that log is controlled by `write.metadata.previous-versions-max`, whose current default is 100. Automatic deletion of the oldest tracked file is disabled by default.

For tables with frequent commits, automatic deletion can be enabled:

```
ALTER TABLE catalog.analytics.events
SET TBLPROPERTIES (
  'write.metadata.delete-after-commit.enabled'
= 'true',
  'write.metadata.previous-versions-max' =
'100'
);
```

After the tracked-history limit is reached, a successful commit deletes the oldest metadata file still recorded in the metadata log.

This setting does not retroactively remove metadata files that have already fallen out of the log. Once a metadata JSON file is no longer tracked, enabling automatic deletion cannot rediscover it. Untracked metadata files must be identified through orphan-file cleanup.

Reducing the number of retained metadata versions should be done cautiously. Metadata JSON files are usually much smaller than data files, so aggressive retention changes may provide limited storage savings while reducing the metadata history available for diagnosis and recovery. Tables with very high commit frequencies have a stronger reason to enable automatic cleanup than tables updated only a few times per day.

Rewriting Manifests

Snapshot expiration removes historical state, but it does not address inefficient manifest organization in the current snapshot. Iceberg uses manifest files to track data and delete files. During scan planning, the engine reads the manifest list, eliminates manifests whose partition summaries cannot match the query and then evaluates entries in the remaining manifests.

As files are added, removed, compacted and rewritten, the current snapshot may reference many small manifests or manifests whose entries span broad partition ranges. More manifests increase metadata-open and planning work. Broadly distributed entries also make manifest-level partition pruning less effective.

The `rewrite_manifests` procedure reorganizes manifest entries without rewriting the underlying data files:

```
CALL catalog.system.rewrite_manifests(
  table => 'analytics.events'
);
```

The procedure rewrites manifests in parallel and, by default, orders their entries using the fields in the partition specification. Better alignment between manifests and partition values allows queries to eliminate entire manifests before reading their individual file entries.

For a table that has evolved through multiple partition specifications, the rewrite can be limited to a specific specification:

```
CALL catalog.system.rewrite_manifests(
  table => 'analytics.events'
  spec_id => 3
);
```

Latest Iceberg releases also allow manifest entries to be clustered by selected partition transforms:

```
CALL catalog.system.rewrite_manifests(
  table => 'analytics.events'
  sort_by => array('event_date')
);
```

This is useful when queries frequently filter on a particular partition field and arranging manifest entries around that field improves manifest pruning. If `sort_by` is not supplied, Iceberg uses all partition transforms in specification order.

Manifest rewriting should be considered after substantial data-file maintenance. Compaction, clustering and delete-driven data rewrites can replace large numbers of files and introduce new manifests. Rewriting manifests afterward can consolidate and realign the metadata representing the new layout.

It should not be run automatically after every small commit. The rewrite itself requires a Spark job and a metadata commit. It is justified when manifest count, size distribution, or planning latency indicates that current manifest organization has become inefficient.

Designing a Safe Cleanup Strategy

Cleanup should begin with explicit recovery and retention requirements. Determine how long users require time travel, how many recent snapshots should remain available for rollback and whether any branches or tags have independent audit or development lifecycles.

Separate Storage Cleanup from Planning Optimization

Use snapshot expiration and orphan-file removal to reclaim storage. Use manifest rewriting to improve scan planning. Treating all three as generic cleanup can lead to unnecessary manifest rewrites or unsafe file deletion.

Run File Rewrites Before Metadata Cleanup

Complete compaction, delete-driven rewrites or clustering before expiring snapshots. These operations create replacement files, while snapshot expiration removes the superseded files once the retention boundary allows it.

If a major rewrite produces many new manifests, run `rewrite_manifests` after the data-file rewrite. This avoids optimizing manifests that will immediately be replaced by another maintenance operation.

Expire Snapshots Before Removing Orphans

Snapshot expiration is metadata-aware and can identify files that become unused when historical snapshots are removed. Orphan-file cleanup should follow it and handle files that are not reachable through snapshot metadata.

Always run orphan removal in dry-run mode first. Use a cutoff longer than the longest possible write or maintenance job, and validate path normalization before allowing deletion.

Monitor the Outcome

Record the number of expired snapshots, deleted data and delete files, deleted manifests and orphan candidates. Compare storage consumption before and after the operation, while accounting for object-store reporting delays.

For manifest maintenance, monitor manifest count, manifest size, scan-planning latency and the number of manifests opened by representative queries. A successful procedure call does not by itself demonstrate that the rewrite produced a meaningful planning improvement.

Match Frequency to Table Activity

Streaming and change-data-capture tables may require frequent snapshot expiration and periodic metadata cleanup because they create snapshots continuously. Large batch tables with one commit per day can use a much less frequent schedule.

Cleanup frequency should follow commit rate, rewrite frequency, retention requirements, and measurable storage or planning overhead. The goal is to preserve the history the workload requires without indefinitely retaining files and metadata that no active snapshot or operational process still needs.

Chapter 7

Automating Iceberg Maintenance with Cloudera Lakehouse Optimizer

The optimization techniques discussed in this paper operate at different layers of an Iceberg table. Compaction improves file sizing, clustering reorganizes records for more effective pruning, delete-file rewrites reduce read amplification, snapshot expiration reclaims storage and manifest rewriting improves metadata organization. Apache Iceberg exposes the operations required to perform this work, but deciding when and where to run them remains an operational responsibility.

That responsibility becomes difficult as the number of tables grows. A streaming table receiving thousands of small commits may require frequent compaction, while a daily batch table may remain healthy for weeks. A table serving interactive dashboards may justify clustering because small reductions in scanned data are repeated across many queries. Even within one table, recent partitions may require maintenance while historical partitions remain unchanged.

Static schedules do not account for these differences. Running every maintenance operation against every table each night consumes compute even when little can be improved. Additionally, manually inspecting each table and selecting an appropriate operation does not scale across a production lakehouse.

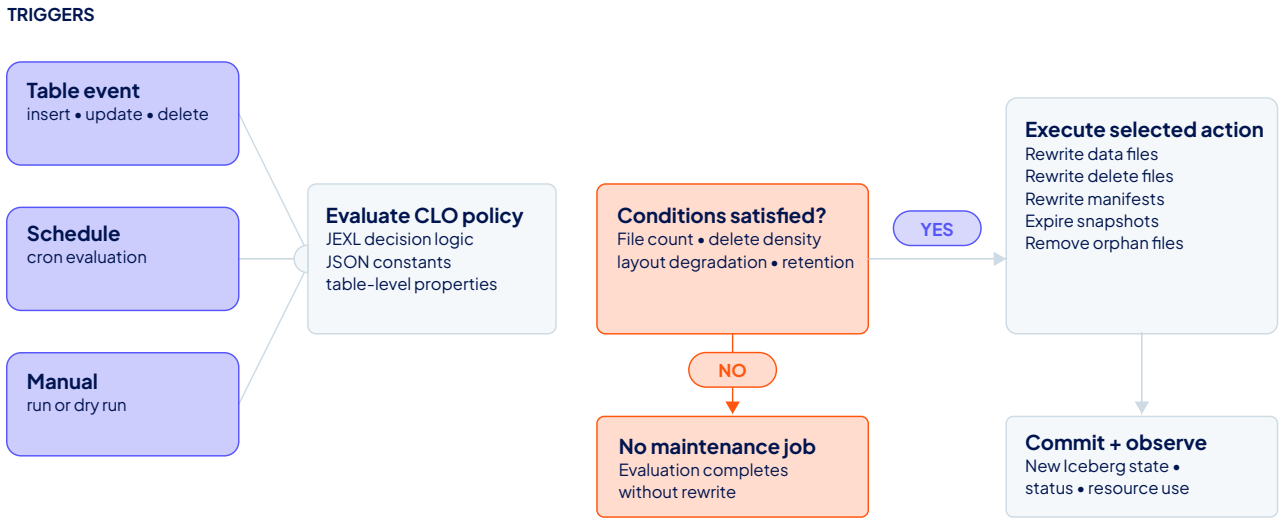
Cloudera Lakehouse Optimizer addresses this operational layer by applying policy-based maintenance to Apache Iceberg tables. It evaluates tables against configurable rules and executes the corresponding optimization actions as Spark jobs. Policies determine which tables are covered, what conditions must be satisfied, which action should run, how that action should be configured and when the policy should be evaluated. CLO provides a control plane for applying the maintenance operations consistently across many tables.

Policy-Based Maintenance

A conventional maintenance workflow often begins with a collection of scheduled Spark procedures. One job compacts selected tables each night, another expires snapshots each week, and another periodically removes orphan files. This is straightforward to implement, but the schedule itself usually contains little information about the condition of each table.

Cloudera Lakehouse Optimizer policy flow

A trigger starts evaluation; maintenance runs only when policy conditions are satisfied



Policy evaluation separates maintenance intent from maintenance execution

Consider a compaction job scheduled to run every six hours. During that interval, one table may accumulate thousands of small files while another receives no new data. Applying the same schedule to both tables can still incur orchestration, compute startup and planning overhead, even if Iceberg ultimately finds no eligible files in the inactive table. Additional logic is needed to determine whether the procedure should be invoked at all and whether compaction is the appropriate maintenance operation.

CLO separates the trigger from the optimization decision. A scheduled or table-level event can cause a policy to be evaluated, but evaluation does not mean that a maintenance action must be executed. The policy can inspect the available table information and initiate an action only when its eligibility conditions are satisfied.

This distinction allows maintenance to respond to table activity without treating every trigger as an instruction to rewrite data. The policy becomes the location where operational decisions are expressed, including minimum file counts, expected improvements, rewrite scope, retention boundaries and action-specific parameters.

CLO Architecture and Execution Model

Cloudera Lakehouse Optimizer operates on the table's Iceberg metadata and storage rather than creating a separate optimized copy of the table. Once an optimization commits a new Iceberg snapshot, engines reading the table through the catalog see the resulting file and metadata layout through the normal Iceberg snapshot model.

This preserves the engine-independent nature of the optimization. The maintenance job may execute through Spark, but its output consists of standard Iceberg data files, delete files, manifests and snapshots. Other compatible query engines can benefit from the resulting layout without requiring the query workload itself to run on Spark.

CLO can be managed through its graphical interface and REST APIs. Policy execution and status can also be monitored through Cloudera Observability. This separates the execution of maintenance from its administration: platform teams can manage policies centrally while still exposing execution state, failures and resource consumption to operational monitoring.

Policy Structure

A CLO policy contains two principal resources:

- A JEXL script that evaluates conditions and defines the optimization actions
- An optional JSON constants file containing configurable values used by the script

The JEXL script contains the decision logic. It can determine whether an action should run and supply arguments to the corresponding Iceberg maintenance operation. The constants file separates tunable values from the policy logic, allowing thresholds and action parameters to be changed without restructuring the entire script.

CLO includes a policy template named `ClouderaAdaptive.` The template provides a starting point for common Iceberg maintenance requirements and can be adapted for different table classes. It is useful as an initial operating policy, but it should not be treated as a universal configuration for every workload. File arrival rate, partition size, query frequency, retention requirements and available maintenance capacity remain workload-specific.

A policy can be applied broadly and then refined through table-level properties. This allows a platform team to maintain a common policy implementation while individual tables supply values appropriate to their workloads.

Triggering Policy Evaluation

CLO supports event-based, schedule-based and manual policy execution.

Event-Based Evaluation

An event-based policy is evaluated following table activity recorded through Hive Metastore events, including inserts, updates and deletes. This connects maintenance evaluation to changes in the table rather than to an arbitrary elapsed interval.

An event should be understood as a reason to evaluate the policy, not as a requirement to perform a rewrite. A streaming table may generate many events, but the policy can wait until enough eligible files have accumulated before launching compaction. This prevents every small append from causing another maintenance job.

Event-based evaluation is useful when tables receive data at irregular rates. A heavily updated table can be considered for optimization as its state changes, while an inactive table generates no equivalent maintenance pressure.

Schedule-Based Evaluation

A schedule-based policy is evaluated using a recurring cron schedule. This model is appropriate for operations tied to an operational window or retention interval. Snapshot expiration, for example, may be evaluated daily even if the table has received many commits during the day.

Schedules are also useful when maintenance must be restricted to periods with available compute capacity. A policy may still decide that no action is required when the scheduled evaluation occurs.

Manual Execution and Dry Runs

Policies can be initiated manually for ad hoc maintenance and validation. CLO also supports dry-run evaluation, which determines the actions that a policy would generate without starting those actions.

A dry run is particularly important when introducing a policy across an existing catalog. It allows the team to examine which tables qualify, which operations would be selected and which parameters would be supplied before compute-intensive or destructive maintenance begins.

Dry runs should also be repeated after substantial policy changes. A small change to a shared threshold or retention value can affect many tables when the policy is assigned at namespace or catalog scope.

Associating Policies with Tables

Policies can be associated with an Iceberg table through table properties. The default property used for this association is `dlm_policies`.

```
ALTER TABLE catalog.analytics.events
SET TBLPROPERTIES (
  'dlm_policies' = 'ClouderaAdaptive'
);
```

Multiple policy names can be supplied as a comma-delimited value. This can be useful when different maintenance responsibilities require different schedules. A frequently evaluated compaction policy, for example, can be separated from a weekly snapshot-expiration policy.

The policies must not perform conflicting work. Two policies that independently attempt to rewrite the same data files can duplicate compute and increase the probability of optimistic-concurrency conflicts. Separating policies by action is useful only when their responsibilities and execution windows remain clear.

Policies can also be applied at broader scopes, including namespaces and catalogs. Broad assignment reduces administrative overhead, but it makes policy safeguards more important. A catalog-level policy must account for tables with different sizes, ingestion models, retention requirements and service-level expectations.

Table-level configuration provides the mechanism for handling these differences. When a policy is written to consult CLO-specific or Iceberg table properties, the values configured on the table take precedence over the corresponding policy defaults.

The policy must explicitly reference these properties. Setting a CLO-specific property does not cause Iceberg or CLO to act on it independently of the policy logic.

Automating Compaction

Compaction policies should determine whether a rewrite can produce a meaningful improvement before allocating a Spark job. The decision can include the number of eligible files, their size distribution, the expected reduction in file count, the amount of data that would be rewritten and the scope of the affected partitions.

The following example associates several compaction settings with a table:

```
ALTER TABLE catalog.analytics.events
SET TBLPROPERTIES (
  ' 'write.target-file-size-bytes' =
'536870912',
  'write.bin-packing.min-file-count' = '5',
  'dlm.rewriteDataFiles.filesCountDrop' =
'0.15',
  'dlm.rewriteDataFiles.
partialProgressEnabled' = 'true',
  'dlm.rewriteDataFiles.
partialProgressMaxCommits' = '10'
);
```

As we learned, `write.target-file-size-bytes` is an Iceberg table property that establishes the target output file size. In this example, the target is 512 MB.

`write.bin-packing.min-file-count` requires a minimum number of eligible files before a file group is considered for bin-packing. A higher value can prevent rewrites that would consolidate only a small number of files, although it may also allow moderately undersized files to remain in the table longer.

`dlm.rewriteDataFiles.filesCountDrop` expresses the minimum expected reduction in file count required by the policy. A value of 0.15 represents a 15 percent reduction. This provides a cost-control mechanism: if rewriting the candidate files would produce little consolidation, the policy can avoid launching the operation.

The partial-progress properties allow large rewrites to commit completed file groups incrementally, subject to a maximum number of commits. They carry the same trade-offs described in the compaction section. Partial progress limits the work lost when a later group fails, but it exposes the rewrite through multiple snapshots rather than one table-wide commit.

These properties should be treated as policy inputs rather than isolated tuning knobs. A minimum file count of five may be reasonable for one table and far too low for another. The threshold must be evaluated alongside partition size, file-open cost, rewrite volume, commit frequency, and the number of tables competing for maintenance capacity.

Automating Clustering

```
ALTER TABLE catalog.analytics.events
SET TBLPROPERTIES (
  'dlm.rewriteDataFiles.zOrderColumns' =
    'customer_id,event_date,region'

);
```

CLO policies can also supply clustering instructions to data-file rewrites. The following example specifies three columns for Z-ordering:

This property is meaningful only when the associated policy reads it and initiates a data-file rewrite using the corresponding Z-order configuration. It does not change the table's physical layout when it is set, and it is not a standard Iceberg sort-order declaration.

A clustering policy should not run merely because clustering columns have been configured. It should also consider whether enough new or overlapping data has accumulated to justify the rewrite. Reorganizing a stable partition produces no continuing benefit if its file ranges are already selective.

The rewrite can be limited through a predicate:

```
ALTER TABLE catalog.analytics.events
SET TBLPROPERTIES (
  'dlm.rewriteDataFiles.where' =
    'event_date >= current_date - interval 30
days'
);
```

This example restricts the policy to recent data. As with the where argument of Iceberg's rewrite procedure, the predicate determines which files are eligible for selection. It should not be interpreted as row-level deletion or as permission to rewrite only matching rows from a file.

A rolling time predicate can work well for tables in which recent partitions receive late arrivals and updates while older partitions are stable. It is inappropriate when backfills routinely modify older data or when important queries depend on clustering across a longer period.

CLO also supports custom sort specifications for policies that use lexicographic ordering. The choice between a conventional sort and Z-ordering should continue to follow the workload analysis developed in the clustering section. Automation makes the rewrite repeatable but it does not determine whether the selected clustering columns are useful.

Coordinating Data and Metadata Maintenance

Data-file rewrites are only one part of the maintenance lifecycle. Compaction and clustering create replacement files and new snapshots. The original files remain reachable until the snapshots referencing them become eligible for expiration. Rewriting data without a corresponding retention strategy can therefore improve current query performance while allowing storage consumption to continue growing.

CLO policies can coordinate several categories of Iceberg maintenance:

Maintenance action	Primary objective	Typical policy input
Rewrite data files	Consolidate small files or reorganize data	File count, file sizes, rewrite scope, target file size
Rewrite position delete files	Reduce delete-file count and merge eligible deletes	Delete-file count, target size, minimum input files
Rewrite manifests	Consolidate and reorganize current metadata	Manifest count, manifest size, organization
Expire snapshots	Remove historical states outside retention requirements	Snapshot age and minimum snapshots to retain
Remove orphan files	Delete unreferenced files after a safety interval	Minimum file age and validated storage scope

The default `ClouderaAdaptive` policy should not be assumed to execute every available action. Some operations, including orphan-file removal and position-delete-file rewriting, may require a custom policy that explicitly adds and configures the action.

Snapshot retention can be influenced through properties such as:

```
ALTER TABLE catalog.analytics.events
SET TBLPROPERTIES (
  'dlm.expireSnapshot.retainLast' = '50',
  'dlm.expireSnapshot.
snapshotsDurationDeltaMax' = '432000000'
);
```

The first value retains a minimum number of snapshots as interpreted by the policy. The second value represents a duration in milliseconds; `432000000` corresponds to five days.

These values must reflect recovery, time-travel, branching and audit requirements. A lower retention interval may reclaim replaced files sooner, but it also reduces the historical range available to users. Automation should enforce a deliberate retention decision rather than silently substitute for one.

Orphan-file removal requires an additional level of caution. A custom policy can configure the minimum age of candidate orphan files:

```
ALTER TABLE catalog.analytics.events
SET TBLPROPERTIES (
  'dlm.deleteOrphanFiles.olderThan' =
'432000000'
);
```

The cutoff must remain longer than the maximum expected duration of active writes and maintenance jobs. Before enabling automated deletion, the policy should be validated through dry runs and the table's storage paths should be checked for consistent URI representation.

Designing Policies Around Workload Classes

A large catalog should not use a single undifferentiated maintenance profile. Tables with similar operating characteristics can instead be grouped into workload classes.

A streaming or change-data-capture table may use event-based compaction evaluation, a relatively high snapshot-expiration frequency and periodic position-delete maintenance. A large append-only fact table may compact only recent partitions and use a longer retention period.

The differences can be expressed through separate policies, table-level overrides or both. A practical classification should consider:

- Commit frequency and average data volume per commit
- Number and distribution of active partitions
- Rate of small-file and delete-file accumulation
- Frequency and selectivity of queries
- Query latency and planning-time requirements
- Time-travel, rollback and audit retention
- Acceptable maintenance cost and execution window
- Frequency of late data, backfills and row-level changes

Policies should begin with conservative thresholds. Initial executions should be observed for candidate-file count, bytes rewritten, output-file distribution, Spark resource use, commit duration and query improvement. Thresholds can then be adjusted from evidence rather than from a generic table-size category. Policy execution should also be coordinated with other maintenance systems operating on the same tables. Running multiple optimizers independently can produce overlapping rewrites, duplicate compute consumption and commit conflicts. Each table should therefore have a clearly defined maintenance owner.

Path to Continuous Maintenance

Apache Iceberg provides the primitives needed to maintain table performance, but the primitives do not determine the operating policy. A data engineer can call `rewrite_data_files`, `expire_snapshots`, `rewrite_manifests`, and `remove_orphan_files` but the harder problem is deciding when each operation is justified across hundreds or thousands of tables.

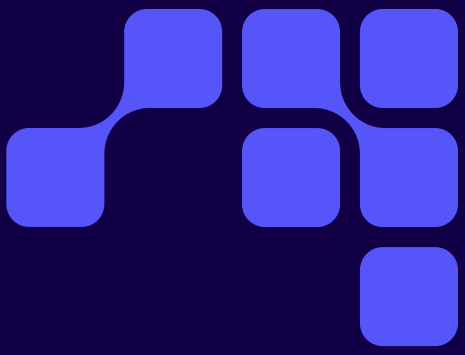
Cloudera Lakehouse Optimizer places those decisions into reusable policies.

- Event and schedule triggers determine when a table is evaluated
- JEXL policy logic determines whether an action is necessary
- JSON constants and table properties provide workload-specific parameters
- Spark jobs execute the selected Iceberg operations, while the CLO interface, APIs, and Cloudera Observability expose their status

The result is a transition from isolated maintenance commands to continuous table management. That transition does not remove the need for workload knowledge. File-size targets, clustering columns, retention boundaries and cleanup safeguards still require deliberate choices. What changes is the ability to apply those choices consistently, observe their results and refine them as table behavior changes.

An effective CLO deployment therefore begins with the same principles developed throughout this paper: understand the workload, identify the physical or metadata condition causing unnecessary work, select the appropriate optimization and measure whether its recurring benefit exceeds its cost. Automation makes that process scalable, but the quality of the outcome still depends on the quality of the policy.

About Cloudera



Cloudera is the only hybrid data and AI platform company that large organizations trust to bring AI to their data anywhere it lives. Unlike other providers, Cloudera delivers a consistent cloud experience that converges public clouds, on-prem data centers, and the edge, leveraging a proven open-source foundation. As the pioneer in big data, Cloudera empowers businesses to apply AI and assert control over 100% of their data, in all forms, improving security, governance, and real-time and predictive insights. The world's largest brands across all industries rely on Cloudera to transform decision-making and ultimately boost bottom lines, safeguard against threats, and save lives.

To learn more, visit Cloudera.com and follow us on LinkedIn and X.

CLOUDERA

Cloudera, Inc. | 6220 America Center Dr, 5th Fl, San Jose, CA 95002 USA | cloudera.com

© 2026 Cloudera, Inc. All rights reserved. Cloudera and associated marks are trademarks or registered trademarks of Cloudera, Inc. All other company and product names may be trademarks of their respective owners. 9933-V3 October 28, 2026