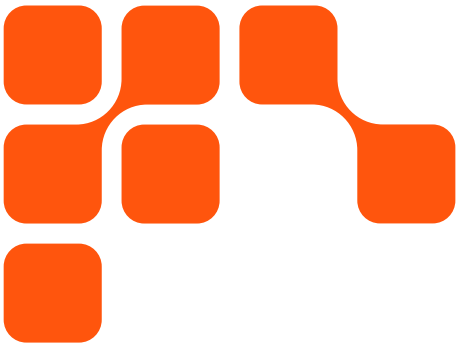
CLOUDERA

WHITEPAPER

A Guide to Debugging your Lakehouse (Apache Iceberg) Issues in Production

Table of Contents



The vision for a **Lakehouse architecture** sounds very simple. You pick an open table format of your choice, such as **Apache Iceberg**, and connect it to your preferred compute engines and catalogs. Now your data is open, your tools are interoperable, and different systems can operate on the same table. From that perspective, everything looks great!

What we do not talk about enough is what happens after table formats like Iceberg actually land in production. As tables grow to terabytes and petabytes, as pipelines begin running continuously, and as SLAs start to matter, the nature of the system begins to change. This is the point where the operational complexity becomes visible. This is also the point where engineers are forced to reason about the system more deeply—about how the system behaves under concurrency, scale, and continuous evolution.

In large-scale environments where streaming and batch workloads overlap, where multiple jobs operate on the same table, and where maintenance processes run in parallel, a recurring set of symptoms begins to emerge:

- Commit-time failures during writes
- Missing files during reads
- Maintenance jobs that run for extended periods or fail unpredictably
- A steady accumulation of metadata

The rest of this whitepaper will cover these four problems, each structured in terms of the underlying execution model, the patterns observed in production systems, and the practical ways to reason about them. We will also present some examples from the Iceberg Slack channel for each problem to shed light on how things show up in real-world deployments.

Problem 1: Commit Conflicts

Problem Context

In a typical production deployment, a single Iceberg table is rarely written by a single process—you have multiple workloads running at the same time. For example, streaming ingestion pipelines can continuously append or update data. Batch jobs may perform MERGE INTO, UPDATE, or DELETE operations. At the same time, background maintenance tasks such as compaction or clustering rewrite files to optimize layout and storage costs.

Under these conditions, commit-time failures begin to appear in the form of **validation exceptions** indicating conflicting files.

```
ValidationException: Found conflicting
files that can contain records matching...
```

Here are some real-world examples:

Figure 1

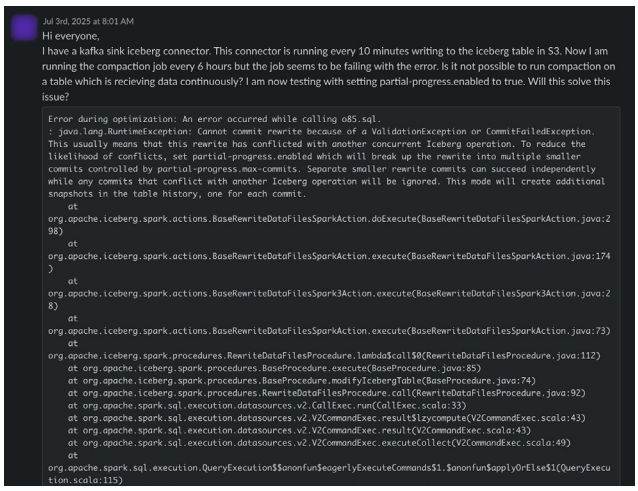
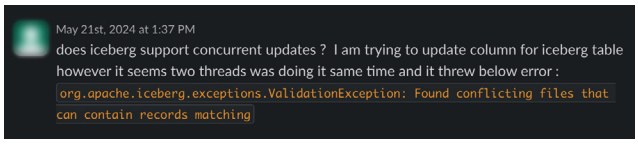


Figure 2

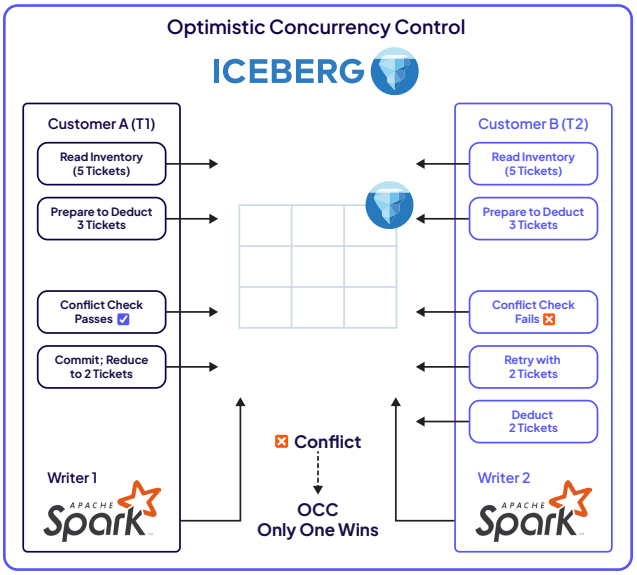


These failures can often be surprising because the operations involved may appear logically independent. For example, two jobs may be updating different rows or even different partitions, yet one of them fails at commit time. This behavior is sometimes interpreted as a limitation in concurrency support. In reality, it is a direct consequence of Iceberg's design.

Iceberg Execution Model

To understand commit conflicts, we need to look at how Iceberg actually executes write operations under concurrency. Iceberg enforces correctness using Optimistic concurrency control (OCC). Instead of locking the table during execution in Iceberg, each write operation proceeds independently and is only validated at commit time.

Figure 3

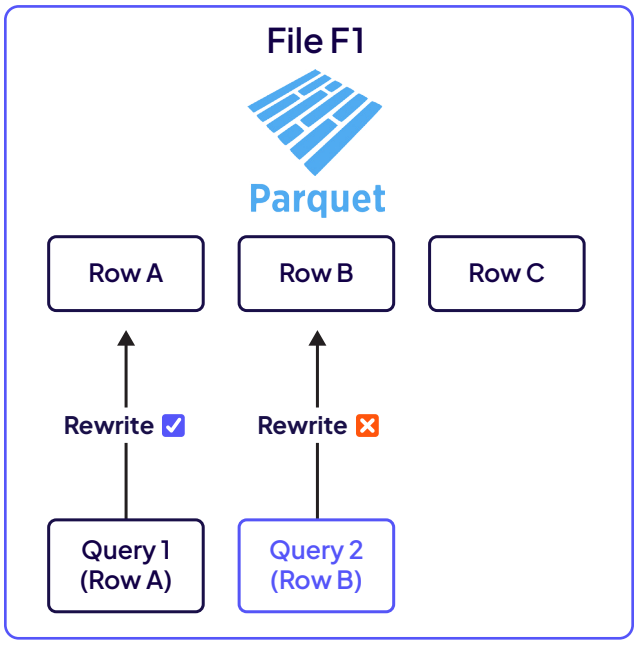


All write operations in Iceberg follow a snapshot-based execution model. A write query begins by planning against a specific snapshot of the table. During this phase, it identifies the set of data files that must be rewritten to apply the requested changes. These files form the **working set** of the operation. The system then constructs a **new snapshot** that includes newly written data files along with unchanged files. At commit time, Iceberg performs a critical validation step to ensure that the plan is still valid with respect to the current state of the table. The key constraint here is that none of the data files used during planning must have been modified or replaced since the query began.

So what this means is your entire write is validated against the state of the table at planning time. And if even one of those files has changed in the meantime, the operation is rejected. This is what guarantees correctness under concurrent writes and it also explains why seemingly independent operations can fail.

A key consequence of this model is that conflict detection operates at the level of **data files**, not individual rows. This distinction is important because most write operations—such as `MERGE INTO`, `UPDATE`, or `DELETE`—are expressed in terms of rows, but are executed by rewriting entire data files. As a result, two operations that appear independent at the row level may still conflict if they touch the same underlying file.

Figure 4



In practice, this leads to scenarios where:

- Two queries target different rows
- Both rows reside within the same data file
- Each query attempts to rewrite that file

In such cases, Iceberg treats this as a conflicting write. Since both operations depend on the original version of the file, only one can successfully commit. The other is rejected to prevent inconsistent updates.

Failure Patterns in Production

In production systems, commit conflicts tend to follow some recurring patterns. As discussed, these patterns emerge naturally from the interaction between snapshot-based execution and file-level validation under concurrent workloads.

Broad File Scans in MERGE Operations

`MERGE INTO` operations that lack selective predicates often scan and rewrite a large number of data files. Since Iceberg executes updates by rewriting entire files, this expands the working set of files involved in the commit. As the number of files touched increases, so does the probability that at least one of them will be modified by another concurrent operation before commit, leading to validation failure.

Overlapping Streaming and Batch Workloads

In many production systems, continuous ingestion pipelines operate alongside periodic batch jobs on the same table. Streaming jobs frequently append or update data in recent partitions, while batch jobs (such as MERGE INTO or DELETE) operate on overlapping time ranges. Since both workloads may rewrite files within the same partitions, they frequently contend at the file level. This overlap is exacerbated when ingestion frequency is high, increasing the likelihood that batch jobs operate on stale snapshots.

Long-Running Queries

The duration of a write operation directly impacts its susceptibility to conflicts. Long-running queries increase the window during which the underlying table state can change. As other operations commit new snapshots during execution, the set of files originally read by the query becomes progressively stale. By the time the operation attempts to commit, the probability that one or more files have been modified is significantly higher, leading to validation failure.

Conflicts Between Maintenance and Ingestion Workloads

A recurring pattern observed in production systems is the occurrence of commit conflicts between continuous ingestion pipelines and background maintenance operations such as compaction (i.e., `rewrite_data_files`). In these scenarios, streaming or micro-batch ingestion jobs continuously append or update data, while periodic compaction jobs attempt to rewrite existing data files to optimize layout and file sizes. Despite being conceptually orthogonal, both operations act on overlapping sets of data files and are therefore subject to the same commit-time validation rules.

Mitigation

Mitigating commit conflicts requires aligning workload design with file-level validation.

- The most effective approach is to minimize overlap in the sets of files that concurrent operations act upon. This can be achieved by designing partitioning schemes that isolate write domains and by scoping queries to operate on narrower subsets of data.
- Controlling write concurrency is equally important. Scheduling batch jobs to avoid overlap with streaming ingestion, or isolating workloads across partitions, reduces the likelihood of conflicts.
- Reducing job execution time also plays a role. Shorter-running operations reduce the window during which the table state can change, lowering the probability of validation failure.
- Aligning table design with update patterns is critical. Designing tables with appropriate partitioning and file sizing ensures that updates are localized—reducing the probability of file-level contention and improving overall write concurrency.
- And finally, when it comes to maintenance operations like compaction, these need to be coordinated with ingestion, because both are rewriting files. So either: run them during low ingestion periods or scope them to smaller subsets. Otherwise, you are just guaranteeing conflicts.

Problem 2: Orphan File Cleanup and OOM Failures

Problem Context

Orphan file cleanup is intended to remove files that are no longer referenced by any valid snapshot. These files are typically produced by failed writes or incomplete maintenance operations. Iceberg provides Apache Spark procedures such as `remove_orphan_files` to handle this cleanup. However, in large-scale production systems, this operation exhibits poor scalability characteristics. Engineers observe long-running jobs, unpredictable execution times, and, in many cases, driver out-of-memory errors. Here are a few snippets from the real world.

Figure 5

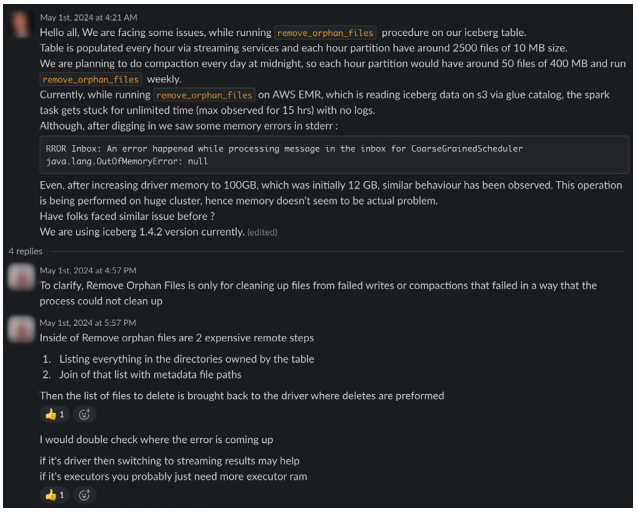
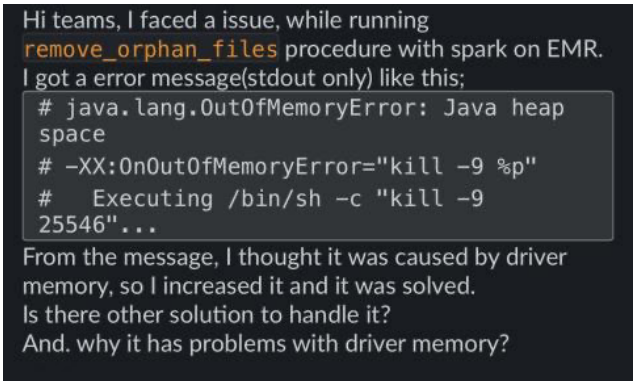


Figure 6



At first glance, this appears to be a resource issue. However, the underlying cause lies in how the operation is executed. The key point to takeaway is that the cost of orphan file cleanup is not proportional to the number of orphan files, it is proportional to the total number of files in the table's storage location.

That's a very important aspect because now, even if you have very few orphan files, the operation can still be extremely expensive. And this is exactly what shows up in production systems with high file counts—especially when ingesting small files at high frequency.

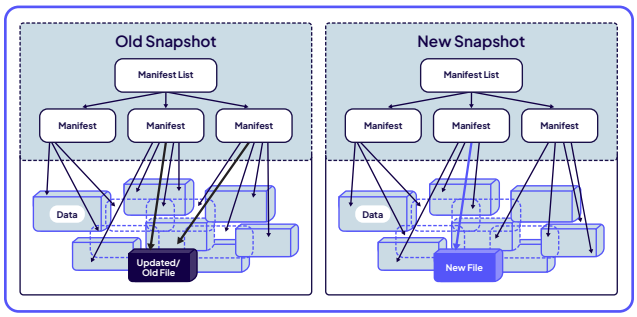
So you start seeing failures like:

- Spark driver out-of-memory errors
- Long-running or completely stalled cleanup jobs
- Excessive resource consumption during execution

So what looks like a simple maintenance task becomes one of the most expensive operations in the system.

Iceberg Execution Model

Figure 7



Unlike most Iceberg operations, which rely on metadata (manifests and snapshots) to avoid scanning storage, orphan file cleanup requires reconstructing the set of data files that exist in storage but are not referenced in metadata. This procedure involves two expensive steps:

1. **Listing all files in the table's storage location:** This includes all directories and partitions under the table root. So the cost of this step grows linearly with the total number of files.
2. **Joining this listing with metadata-referenced files**

And then you compute the difference between the files in storage and files in metadata, which gives you the orphan files. The resulting list of orphan files is then collected and processed for deletion, typically at the Spark driver.

This design leads to several scaling challenges:

- Full table listing is required, even if only a small number of orphan files exist
- The join between storage listing and metadata grows with table size
- Intermediate state (file lists) may need to be materialized, often at the driver

So the key takeaway here is: this operation completely bypasses Iceberg's metadata advantages and falls back to storage-level enumeration. Which is why Cleanup cost=O (total files in storage), not O (orphan files).

Failure Patterns in Production

Production systems exhibit several consistent patterns pertaining to Orphan File Cleanup issues. Here are a few.

Execution Stalls and Memory Failures in Large Tables

A recurring pattern observed in production systems is that `remove_orphan_files` operations degrade significantly or fail entirely when executed on large tables with high file counts. In one reported scenario, a table ingesting data hourly accumulated thousands of files per partition, resulting in a very large number of files across the table. When orphan cleanup was triggered, the Spark job remained active for extended durations—up to 15 hours, without producing logs or visible progress. Eventually, the job surfaced OutOfMemoryError exceptions.

Cost Scaling with Table Size Rather Than Problem Size

A second pattern observed across deployments is the disproportionate cost of orphan cleanup relative to the actual number of orphan files present. In practice, orphan files are typically produced by failed writes, aborted compactions, or incomplete operations, and their number is often small. However, users consistently report that cleanup operations exhibit execution times and resource consumption proportional to the total size of the table, rather than the number of orphan files being removed.

Driver-Centric Bottlenecks and Failure Localization

Although parts of the orphan cleanup process are distributed, key stages require global coordination and aggregation of file metadata. This often results in large intermediate datasets being materialized at the driver. Failures tend to localize around these stages, with memory pressure appearing either during large-scale file listing or during the set difference computation between storage and metadata views. This makes the driver a critical bottleneck, and explains why failures can occur even in otherwise well-provisioned clusters.

Misuse as Routine Maintenance in High-Scale Environments

A final pattern observed across real-world deployments is the misalignment between the intended purpose of orphan cleanup and how it is used in practice. In many deployments, orphan cleanup is scheduled as a periodic maintenance task alongside compaction and snapshot expiration. However, unlike those operations, orphan cleanup does not scale with metadata and instead incurs a storage-level cost. When executed regularly on large tables, it introduces repeated full-table scans, leading to spikes in resource utilization and increased instability.

Mitigation

For effective mitigation of orphan file cleanup failures we need to recognize that `remove_orphan_files` operates fundamentally differently from most Iceberg maintenance procedures. Hence, mitigation strategies must focus on controlling the scope, frequency, and execution characteristics of this enumeration. Let's understand a few of these strategies.

- **Restrict orphan cleanup to failure-recovery scenarios rather than routine scheduling**
  Orphan cleanup is designed to handle inconsistencies introduced by failed writes or incomplete operations. Running it as part of periodic maintenance on large tables forces repeated full-table scans, even when no meaningful orphan files exist.
- **Reduce the scope of file enumeration instead of operating on the entire table**
  The dominant cost driver in orphan cleanup is the size of the storage footprint being scanned. Limiting cleanup to specific partitions or subpaths where failures are known to have occurred significantly reduces the volume of data processed.
- **Account for driver-centric execution characteristics during planning**
  Orphan cleanup introduces stages where large intermediate state is materialized, often at the driver. Increasing driver memory may delay failures, but does not address the underlying issue. Stability depends on reducing the amount of data being aggregated rather than simply scaling resources.
- **Align cleanup strategy with table growth and ingestion characteristics**
  As tables grow and ingestion rates increase, the number of files expands continuously. A fixed cleanup schedule does not account for this growth, leading to progressively longer and more expensive executions. Aligning cleanup strategy with table scale ensures that operational cost remains proportional to actual need, rather than being driven solely by storage footprint.

Problem 3: NotFoundException

Problem Context

In production Iceberg deployments, a recurring class of failures manifests as `NotFoundException` errors during read operations. These errors typically indicate that a data or metadata file referenced by a query or processing job no longer exists at the expected storage location.

`NotFoundException` is observed during query execution and is often interpreted by users as data corruption or inconsistency within the table. However, analysis of real-world incidents reveals that these failures are not caused by corruption, but rather by inconsistencies between the snapshot state assumed by a query and the physical state of the underlying storage.

Here are some real-world examples:

Figure 8

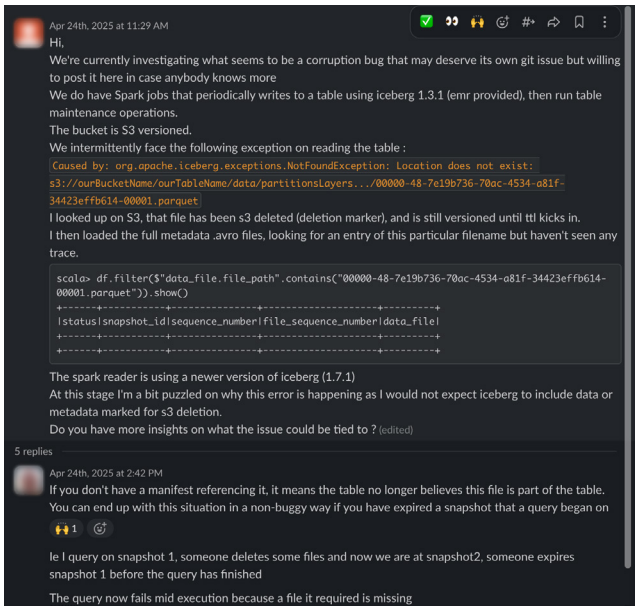
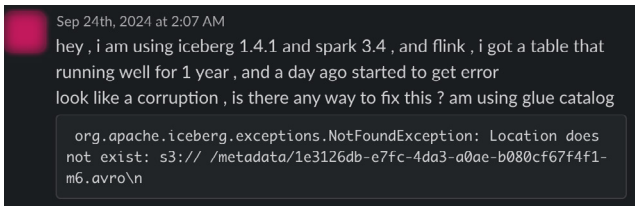


Figure 9



These errors occur across engines such as Apache Spark and Apache Flink, and frequently surface during long-running reads, streaming ingestion jobs, or query execution over large tables.

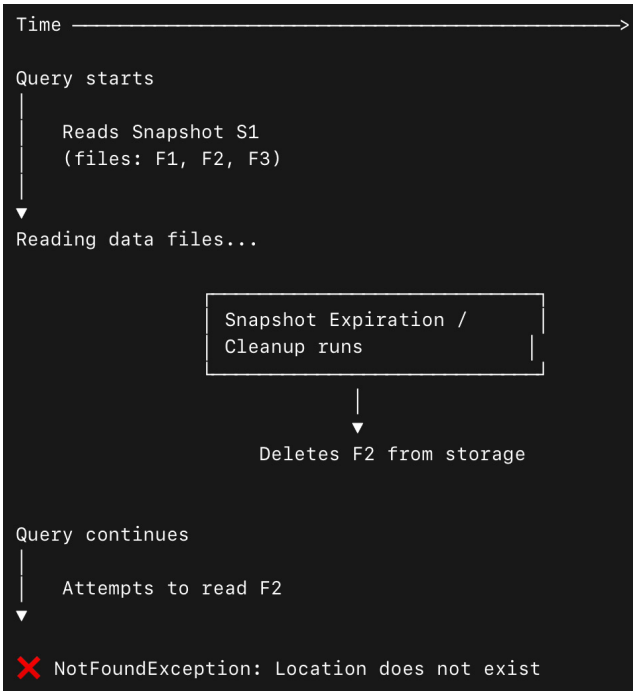
Iceberg Execution Model

To understand why this happens, we need to look at how reads work in Iceberg. Every query operates on a specific snapshot of the table. That snapshot defines: the metadata files, the manifests, and the exact set of data files to read. So when a query starts, it essentially says: "I am going to read this snapshot, and these are the files I expect to exist." Now there's an important assumption here, i.e., Iceberg assumes that all referenced files remain available for the duration of the query.

However, Iceberg doesn't track active readers and it doesn't need to know who is reading what because Snapshot isolation makes queries independent of the table. So, reads operate on immutable snapshots, and file lifecycle is governed purely by metadata. There's no coordination with query execution, and no file pinning in storage. As a result, maintenance operations (such as compaction) can remove files that are still required by in-flight queries on older snapshots.

Now if you look at the diagram below, this is exactly what happens.

Figure 10



A query starts on snapshot S1 and it begins reading files—say F1, F2, F3. Meanwhile, maintenance operations such as snapshot expiration or cleanup runs in the background. These operations advance the table state, and remove files that are no longer needed for the latest snapshot. So now the file F2 gets deleted from storage, but the query is still running, and still expects F2 to exist. So when it tries to read that file, it gives NotFoundException: Location does not exist.

So it's a race condition between snapshot-based reads and file deletion happening underneath. And specifically, it's caused by premature deletion of files that are still required by active queries.

Failure Patterns in Production

Analysis of production incidents reveals that `NotFoundException` errors arise from two distinct but related classes of failure, both of which violate the assumption that all files referenced by a query's snapshot remain accessible throughout execution. These failures occur when files become unreachable either due to physical deletion from storage or removal from the table's metadata lineage.

Physical Disappearance of Files from Storage

A common pattern observed is the physical removal of data or metadata files that are still required by active readers. This typically arises in architectures where ingestion and maintenance are executed independently, often across different engines.

In one reported setup, data ingestion was performed continuously via Apache Flink, while maintenance operations such as `rewrite_data_files`, `expire_snapshots`, `remove_orphan_files`, and `rewrite_manifests` were executed periodically using Spark procedures. Snapshots were configured to expire within very short intervals (on the order of minutes). So, over time, the Flink job began failing with errors such as:

```
NotFoundException: Location does not
exist: .../metadata/*.avro
```

The impact is particularly severe when metadata files (`*.avro`) are removed, as this prevents reconstruction of table state and can cause failures even before data access begins.

Logical Disappearance via Snapshot Evolution

A second, more subtle pattern arises when files remain physically present in storage, but are no longer referenced by the table's metadata.

In one observed scenario, a query failed while attempting to read a data file that still existed in S3 (with versioning enabled), but was no longer referenced by any manifest. Investigation showed that:

- The file had been removed from the table during a rewrite or delete operation
- The snapshot that previously referenced the file had been expired
- No metadata entry existed linking the file to the current table state

From an Iceberg perspective, if a file is not referenced by any manifest, the table no longer considers it part of the dataset, regardless of its presence in storage. Queries that began on an earlier snapshot expecting this file will fail once that snapshot is no longer retained.

Mitigation

Effective mitigation of `NotFoundException` requires aligning snapshot lifecycle management, metadata evolution, and query execution behavior.

- **Configure snapshot retention based on reader lifetime rather than storage optimization**
  Snapshot retention determines how long historical table state remains available. If retention windows are shorter than long-running queries or streaming checkpoints, readers may reference snapshots that no longer exist. Retention should therefore exceed query durations, checkpoint intervals, and recovery windows, especially in continuously ingested tables.
- **Control physical file deletion relative to snapshot retention policies**
  Snapshot expiration and file deletion are separate processes. If cleanup runs too aggressively, files required by recently expired snapshots may be removed prematurely. Introducing a buffer between expiration and deletion, and avoiding tightly scheduled cleanup, ensures files remain physically available for active readers.
- **Coordinate maintenance operations with ingestion and query workloads**
  Maintenance operations continuously evolve table state. When executed independently, they can invalidate assumptions held by active readers. Aligning maintenance frequency with workload patterns and avoiding aggressive full-table operations reduces the risk of snapshot evolution outpacing query execution.

Problem 4: Metadata Amplification Issue

Problem Context

To understand why metadata bloat becomes a real bottleneck in Iceberg, it is important to look closely at how commits are structured today.

Each commit in Iceberg is not a simple append to metadata. Instead, it produces a new **metadata.json** file, a new **manifest list**, and one or more **manifest files** that describe the data files added or removed. This layered structure is fundamental to Iceberg's snapshot isolation model, but it also introduces a form of metadata write amplification.

Even for a minimal operation, such as appending a single file or deleting a small subset of records, the system must rewrite multiple layers of metadata. In particular, operations that modify existing data often require rewriting manifest files in a copy-on-write manner. In the worst case, this can involve rewriting a large portion of existing manifests, even when the logical change is small. As a result, the cost of metadata operations becomes proportional to the size and structure of the table's existing metadata, rather than the size of the change being applied. This effect becomes more pronounced in workloads characterized by frequent commits, such as streaming or micro-batch ingestion. In these scenarios, small and incremental updates repeatedly trigger metadata rewrites that scale with the table, leading to inefficient behavior.

This manifests in two ways. First, write latency increases, because each commit must construct and atomically publish a new metadata state. Second, query planning becomes more expensive, as engines must read and process metadata structures that are frequently rewritten, reducing opportunities for reuse and efficient planning. In effect, while Iceberg avoids scanning unnecessary data files, part of the cost is shifted to metadata processing, which grows with the complexity of the metadata structures rather than the magnitude of the underlying data changes.

Here are some real world examples:

Figure 11

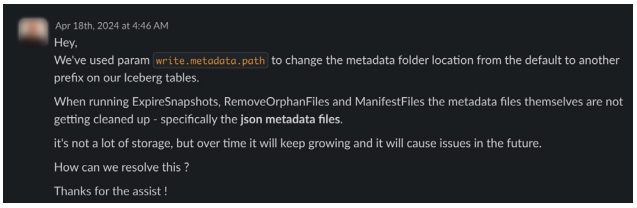
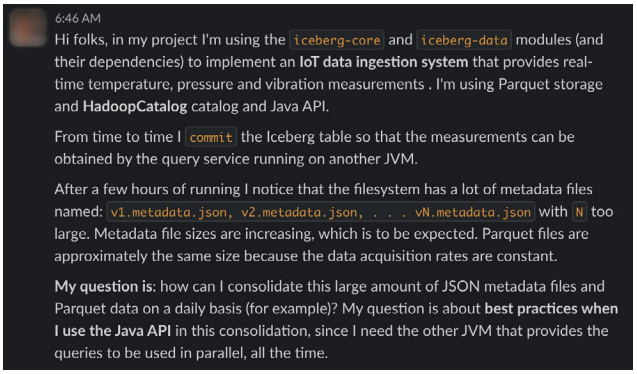
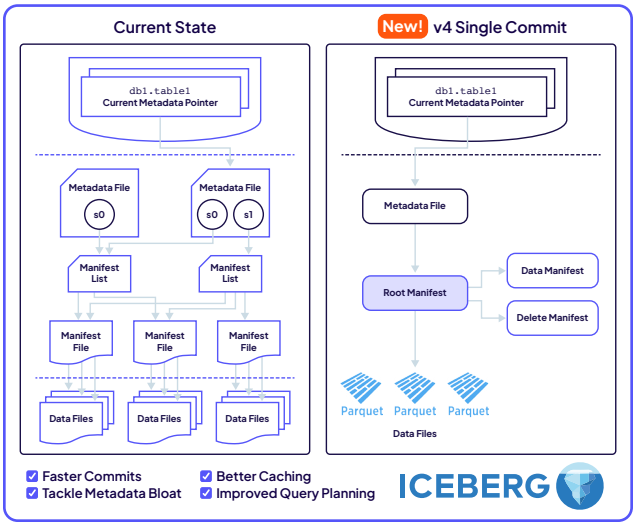


Figure 12



Proposed Direction: Collapsing the Metadata Hierarchy

Figure 13



The proposed v4 design introduces a structural simplification through the concept of a **Root Manifest**, which replaces the manifest list as the entry point for a snapshot.

Instead of maintaining a multi-level hierarchy (metadata.json -> manifest list -> manifests), the structure is flattened so that the snapshot directly references a single root manifest. This root manifest then references data manifests and delete manifests, which in turn point to the underlying files. Conceptually, the hierarchy becomes:

```
Root Manifest → Data / Delete
Manifests → Files
```

This change alters the way metadata evolves during commits. Rather than rewriting large portions of the metadata tree, each commit can now modify only the affected parts of the structure, allowing metadata changes to scale with the operation itself rather than the entire table state. The implication is significant for high-frequency workloads. Commit cost becomes proportional to the number of files changed, rather than the cumulative size of the metadata graph. This directly reduces metadata write amplification, lowers commit latency, and stabilizes performance under continuous ingestion.

Additionally, the root manifest can act as an aggregation layer for file-level statistics (e.g., min/max values, counts), enabling earlier pruning during query planning. This reduces the need to traverse deeper levels of the metadata tree, improving planning efficiency as tables grow.

Why Does This Matter in Practice?

This proposal directly addresses a pattern observed in production systems:

Metadata overhead grows with table activity, not just data size.

In workloads dominated by small, frequent commits, such as streaming ingestion pipelines, the current design leads to a disproportionate increase in metadata operations. Over time, this affects both write throughput and read latency, even when the underlying data volume remains manageable.

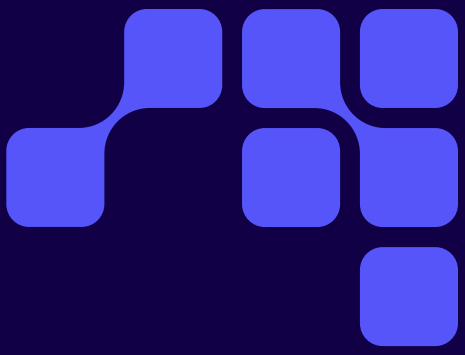
By restructuring the metadata hierarchy and localizing changes, the v4 approach aims to restore alignment between **operation size and metadata cost**. This makes Iceberg more predictable under sustained write pressure and better suited for workloads that require continuous updates rather than large batch writes.

Conclusion

What these issues make clear is that most production failures in Iceberg are a direct consequence of its design. This makes it critical to understand the core primitives that drive its behavior. The challenge is not just to fix these problems, but to reason about them correctly.

Systems that operate well at scale are the ones that align workload patterns—ingestion, maintenance, and query execution—with these underlying mechanics. Partitioning strategies, job scheduling, retention policies, and commit frequency are not just configuration choices—they shape how the system behaves under concurrency and continuous change, and ultimately determine how well Iceberg runs in production.

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