Reliable Context

RELIABLE CONTEXT / RESEARCH & PRACTICAL GUIDANCE

Delta Lake and Iceberg Time Travel Is Not an Audit Trail

> Reliable Context is a kernel of truth: a system of record for organizations that fuse private and public information and must defend every value they act on. One canonical record per real thing, a cited source and an observation date behind every value, unknowns declared rather than left blank, and any past state of the record reproducible.

Published by Reliable Context · 2026-10-29 · delta lake time travel · iceberg snapshots time travel · time travel vs audit trail · delta lake vacuum retention · reproduce past table state

Original article: https://reliablecontext.com/articles/lakehouse-time-travel-delta-iceberg-is-not-an-audit-trail



In this article

1. What time travel does
2. Delta Lake: versions, timestamps and the transaction log
3. Apache Iceberg: snapshots, logs and references
4. Expiry and VACUUM: history is a retention setting
5. What time travel does not record
6. Worked example (fictional)
7. Adding what time travel lacks
8. A reproducibility test
9. Decision table
10. Sources

What time travel does

Time travel is the ability to query a table as it stood at an earlier committed version. Delta Lake and Apache Iceberg both implement it the same basic way: every write produces a new immutable table version that points at a set of data files, and old versions stay readable for as long as their metadata and files survive.

Delta's own documentation lists "re-creating analyses, reports, or outputs" among its uses and adds that this "could be useful for debugging or auditing, especially in regulated industries" (Delta Lake, Batch reads and writes). It is also easy to over-read. An **audit trail** is a reviewable record of actions and changes affecting a defined process or asset, from which a reviewer can reconstruct who or what did something, when, and to which object (Reliable Context, Audit trail). Time travel answers a narrower question: what bytes did this table contain at version N, provided version N still exists?

Delta Lake: versions, timestamps and the transaction log

A Delta table is Parquet data files plus a `_delta_log` of JSON commit files and checkpoints. Time travel uses SQL `AS OF` clauses (Delta Lake, Batch reads and writes):

```
SELECT * FROM default.people10m TIMESTAMP AS OF '2018-10-18T22:15:12.013Z';
SELECT * FROM delta.`/tmp/delta/people10m` VERSION AS OF 123;
```

The version number comes from `DESCRIBE HISTORY`, which returns, per write, the version, commit timestamp, `userId`, `userName`, operation, operation parameters, job and notebook details, `readVersion`, operation metrics and `userMetadata` (Delta Lake, Table utility commands). That looks like an audit record. Three details qualify it.

First, the sample output shows `userId` and `userName` as `null`, noting that "some of the columns may be nulls because the corresponding information may not be available in your environment" (Delta Lake, Table utility commands).

Second, the protocol makes commit provenance optional. The `commitInfo` action is described as "additional provenance information" that a commit file "can optionally contain", and "implementations are free to store any valid JSON-formatted data" in it (Delta Transaction Log Protocol). It is the writer's self-description, not an independent record, and checkpoints "do not preserve commit provenance information", so `commitInfo` lives only in the JSON commit files (Delta Transaction Log Protocol).

Third, by default the timestamp of version N "depends on the timestamp of the log file", so timestamp-based time travel "can break if you copy the entire Delta table directory to a new location", while version-based time travel is unaffected (Delta Lake, Batch reads and writes). The optional In-Commit Timestamps feature instead writes a monotonically increasing `inCommitTimestamp` into `commitInfo` (Delta Transaction Log Protocol).

Apache Iceberg: snapshots, logs and references

Each Iceberg **snapshot** records a snapshot ID, an optional parent snapshot ID, a sequence number, `timestamp-ms` ("used for garbage collection and table inspection"), a manifest list and a `summary` map whose only required key is `operation` (`append`, `replace`, `overwrite` or `delete`) (Apache Iceberg Table Spec). Its optional summary fields are counts plus operational keys such as `engine-name`; none identifies a person or a reason.

Table metadata also keeps an optional `snapshot-log` of timestamp and snapshot ID pairs recording changes to the current snapshot. The spec warns that the snapshot-log and the parent-child lineage "might indicate different snapshot IDs for a specific timestamp" and directs point-in-time queries to the snapshot-log (Apache Iceberg Table Spec).

Spark reads earlier states with `TIMESTAMP AS OF` or `VERSION AS OF`, where the version may be a snapshot ID or a branch or tag name (Iceberg, Spark queries):

```
SELECT * FROM prod.db.table TIMESTAMP AS OF '1986-10-26 01:21:00';
SELECT * FROM prod.db.table VERSION AS OF 10963874102873;
SELECT * FROM prod.db.table.history;   -- made_current_at, snapshot_id, parent_id, is_current_ancestor
```

Iceberg also has **references**: tags label individual snapshots and branches are mutable named references, each with its own retention policy (Apache Iceberg Table Spec).

Expiry and VACUUM: history is a retention setting

A version can be read only while both its metadata and its data files exist, and routine maintenance removes both.

Setting	Format	Documented default	Effect on time travel
`delta.logRetentionDuration`	Delta	interval 30 days	Log entries older than this are cleaned up when a checkpoint is written

Setting	Format	Documented default	Effect on time travel
`delta.deletedFileRetentionDuration`	Delta	interval 7 days (table reference: 1 week)	Removed data files older than this become candidates for `VACUUM`
`history.expire.max-snapshot-age-ms`	Iceberg	432000000 (5 days)	Default maximum snapshot age kept when snapshots are expired
`history.expire.min-snapshots-to-keep`	Iceberg	1	Default minimum snapshots kept per branch
`history.expire.max-ref-age-ms`	Iceberg	Long.MAX_VALUE (forever)	Default age limit for tags and non-main branches
`write.metadata.previous-versions-max`	Iceberg	100	Number of previous metadata files tracked

Sources: Delta Lake, Batch reads and writes; Delta table properties reference; Iceberg, Configuration.

For Delta, the docs say that by default "you can time travel to a Delta table up to 30 days old" unless you have run `VACUUM` or changed these properties; `VACUUM` is "not triggered automatically", and once it runs, "the ability to time travel back to a version older than the retention period is lost" (Delta Lake, Batch reads and writes; Table utility commands). A table vacuumed on schedule with default settings therefore keeps about seven days of readable versions, not thirty; the docs advise setting `delta.deletedFileRetentionDuration` to 30 days to keep a month through `VACUUM`. Log cleanup can also strand versions inside the window, because Delta needs every log entry since the previous checkpoint.

For Iceberg, "snapshots accumulate until they are expired", and expired snapshots "are no longer available for time travel queries" (Iceberg, Maintenance). The Spark `expire_snapshots` procedure defaults `older_than` to five days ago and `retain_last` to 1; snapshots referenced by branches or tags are not removed, and "the main branch never expires" (Iceberg, Spark procedures).

In both formats, how far back you can look is whatever an operator configured and a maintenance job last enforced: a storage-cost decision, not a record-keeping decision made against a retention schedule.

What time travel does not record

Assume every version is kept forever. Four things an audit trail carries are still absent or optional.

Actor. Delta's `userName` and Iceberg's summary map hold whatever the writing engine chose to put there. Neither specification requires an authenticated identity, and in a pipeline it is often a service account, not the person who approved the change.

Reason. Nothing in either format asks why a write happened. A `MERGE` that corrects a data-entry error and a `MERGE` from the nightly load produce the same kind of commit.

Source. A version records which files make up the table, not which upstream document, extract or API response the values came from.

Valid time. Martin Fowler distinguishes actual (valid) history, what was true in the world, from record (transaction) history, what we knew and when (Fowler, Bitemporal History). A table version timestamp is a transaction time for the whole table: when the commit landed. It says nothing about when the facts in a row applied. A late correction committed today shows up around today's commit, not the date the fact became true (Reliable Context, Snapshots vs bitemporal history).

Versions are also table-level: to learn when one row changed, you diff whole versions.

For contrast, in the United States, where 21 CFR Part 11 applies to FDA-regulated electronic records, 21 CFR 11.10(e) calls for "secure, computer-generated, time-stamped audit trails to independently record the date and time of operator entries and actions that create, modify, or delete electronic records" and requires the trail to be retained at least as long as the records themselves (21 CFR 11.10, via Cornell LII). History bounded by a seven-day or five-day default, with an actor field the writer fills in for itself, is a different thing.

Worked example (fictional)

Halden Therapeutics AG (fictional) keeps a Delta table `supplier_status` that a release workflow reads. Supplier Norrvik Chemicals (fictional) was set to `suspended` on 3 March. On 12 March a pipeline merged a qualification letter dated 10 March, effective 1 March, and the row became `qualified`. A batch was released on 13 March. In September an inspector asks what status the release relied on, who changed it, and on what evidence.

The table vacuums weekly with default properties. By September, the March versions are gone: their log entries passed the 30-day window and their data files the 7-day threshold. Had they been kept, time travel would show `qualified` after 12 March and `suspended` before, and `DESCRIBE HISTORY` a `MERGE` by the pipeline's service account. It would not show the letter, who accepted it, or that the status had applied since 1 March.

The answer lies in records the table does not hold: the source document, a decision record, and both time axes.

Adding what time travel lacks

Commit metadata. Attach a reason and a reference to every write. Delta accepts a user-defined string through the `userMetadata` writer option or a session setting, readable in history (Delta Lake, Batch reads and writes):

```
SET spark.databricks.delta.commitInfo.userMetadata=overwritten-for-fixing-incorrect-data;
INSERT OVERWRITE default.people10m SELECT * FROM morePeople;
```

Iceberg adds custom keys to the snapshot summary with the Spark write option `snapshot-property.<key>`, or with `CommitMetadata` around a SQL statement, which the docs describe as useful "for purposes such as auditing or change tracking" (Iceberg, Spark configuration). Use a structured value such as a change-ticket ID and an approver. It is still self-asserted and expires with the version: an index into a durable record, not the record.

Pinned versions. Iceberg tags keep a snapshot past routine expiry; the docs suggest tags "for retaining important historical snapshots for auditing purposes" (Iceberg, Branching and tagging):

```
ALTER TABLE prod.db.table CREATE TAG `EOM-01` AS OF VERSION 30 RETAIN 180 DAYS;
```

The Delta documentation reviewed has no tag equivalent; the options are longer retention properties (set with `ALTER TABLE ... SET TBLPROPERTIES`) or a copy of the version taken when the output is produced.

History in the data. Keep history as rows, not as table versions. Kimball's slowly changing dimension type 2 adds a new row per change with, at minimum, a row effective date, a row expiration date and a current-row indicator (Kimball Group, Type 2). Those rows survive `VACUUM` and expiry because they are current data. To answer both "when was it true" and "when did we know", carry both axes: valid-from and valid-to for the fact, recorded-from and recorded-to for the record, as in Fowler's bitemporal model. Add a source reference and an actor column per row.

Database-native versioning, for comparison. SQL Server system-versioned temporal tables keep a history table and two period columns that the engine sets from the transaction's begin time, queryable with `FOR SYSTEM_TIME AS OF` (Microsoft Learn, Temporal tables). As documented, that is engine-maintained, row-level transaction time, with no actor, reason or valid-time columns.

A separate decision log. Record approvals and corrections in an append-only store with its own retention, referencing version or snapshot IDs. The lakehouse then answers "what did the table hold"; the log answers "who decided what, why, on which evidence".

A reproducibility test

Point-in-time reproducibility means a past result can be rebuilt from the information, rules and source versions available at a declared point (Reliable Context, Point-in-time reproducibility). Run this test on one real output per quarter; the fuller method is in Reproducing a past state.

1. **Pick an output** at least as old as your longest required look-back: a filed report, a model score, a release decision.
2. **Read its declared basis.** It should name table, version or snapshot ID (not only a timestamp), query text and engine version. If it names only a date, record that as a finding.
3. **Check survival.** Confirm the version appears in `DESCRIBE HISTORY` or `table.snapshots`. Compare its age with the effective window: for Delta, the smaller of log retention and deleted-file retention if `VACUUM` runs; for Iceberg, the expiry policy unless a tag holds it.
4. **Re-run by version.** Query `VERSION AS OF` the recorded ID and compare row count and an order-independent checksum with values stored when the output was produced.
5. **Re-run by timestamp.** Query `TIMESTAMP AS OF` the output's production time and confirm it resolves to the same version. A mismatch points to copied directories, clock issues or a rollback.
6. **Inspect the commits around it.** For the versions that produced the recorded state, check whether actor, reason and source reference are present and whether they are filled in by the writer or by an independent system.

7. **Trace upstream.** Confirm the source documents or extracts behind the changed rows can still be produced.
8. **Rehearse maintenance.** On a copy, run the production `VACUUM` or `expire_snapshots` configuration and confirm the version survives.

Steps 4 and 5 test reproduction; steps 6 and 7 test what an audit trail needs. A table can pass the first pair and fail the second.

Decision table

Need	Is time travel sufficient?	What to add
Undo a bad write within days	Yes, within the retention window	Nothing beyond sane `VACUUM` / expiry settings
Re-run last week's report exactly	Usually, if pinned by version	Record version or snapshot ID with each output
Reproduce a quarter-end or filing state	Only if retention or tags cover it	Iceberg tag with `RETAIN`; Delta longer retention or a frozen copy
Know who changed a value	No	Authenticated actor in a decision log; commit metadata as an index
Know why it changed	No	Reason and ticket in commit metadata, backed by a durable record
Know which source a value came from	No	Per-row source reference
Know when a fact was true, not when it loaded	No	Valid-time columns (SCD type 2 or bitemporal)
Keep history as long as the records	Not by default	Retention set from the records schedule; history held as rows or in a separate log

This article is educational and is not legal or regulatory advice; check which record-keeping rules apply to your records in your jurisdiction.

Reliable Context is built around one canonical record per real thing, a cited, dated source behind every value, unknowns declared and any past state reproducible, running inside the customer's perimeter.

Sources

1. Delta Lake documentation, Batch reads and writes (accessed 2026-10-28) — time travel syntax and use cases, timestamp caveat, data retention defaults, `userMetadata`.
2. Delta Lake documentation, Table utility commands (accessed 2026-10-28) — `VACUUM` behaviour and default threshold, `DESCRIBE HISTORY` columns and null note, `RESTORE`.
3. Delta Lake documentation, Table properties reference (accessed 2026-10-28) — defaults for `delta.logRetentionDuration` and `delta.deletedFileRetentionDuration`.

4. Delta Transaction Log Protocol, delta-io/delta, master branch (accessed 2026-10-28) — optional `commitInfo`, checkpoints not preserving it, In-Commit Timestamps.
5. Apache Iceberg Table Spec (accessed 2026-10-28) — snapshot fields, summary operations, snapshot-log, metadata-log, references, retention evaluation, point-in-time reads.
6. Apache Iceberg, Configuration (accessed 2026-10-28) — `history.expire.*` and metadata property defaults.
7. Apache Iceberg, Maintenance (accessed 2026-10-28) — snapshot expiry removes time travel; metadata file cleanup.
8. Apache Iceberg, Spark procedures (accessed 2026-10-28) — `expire_snapshots` defaults; refs not expired; main never expires.
9. Apache Iceberg, Spark queries (accessed 2026-10-28) — `AS OF` syntax, history metadata table.
10. Apache Iceberg, Spark configuration (accessed 2026-10-28) — `snapshot-property.` write option and `CommitMetadata`.
11. Apache Iceberg, Branching and tagging (accessed 2026-10-28) — tags for audit retention, `CREATE TAG ... RETAIN`.
12. Martin Fowler, Bitemporal History, 7 April 2021 — actual (valid) versus record (transaction) history.
13. Kimball Group, Type 2: Add New Row — SCD type 2 minimum columns.
14. Microsoft Learn, Temporal tables (SQL Server) — system-versioned period columns, history table, `FOR SYSTEM_TIME`.
15. 21 CFR 11.10, Controls for closed systems, eCFR — audit trail requirement in 11.10(e) (United States, FDA).

Source links

Links cited in this article, in order of first appearance. Full addresses are provided for readers using extracted text.

1. https://docs.delta.io/delta-batch
2. https://docs.delta.io/delta-utility
3. https://github.com/delta-io/delta/blob/master/PROTOCOL.md
4. https://iceberg.apache.org/spec
5. https://iceberg.apache.org/docs/latest/spark-queries
6. https://docs.delta.io/table-properties
7. https://iceberg.apache.org/docs/latest/configuration
8. https://iceberg.apache.org/docs/latest/maintenance
9. https://iceberg.apache.org/docs/latest/spark-procedures
10. https://martinfowler.com/articles/bitemporal-history.html
11. https://www.ecfr.gov/current/title-21/chapter-I/subchapter-A/part-11/subpart-B/section-11.10
12. https://iceberg.apache.org/docs/latest/spark-configuration
13. https://iceberg.apache.org/docs/latest/branching
14. https://www.kimballgroup.com/data-warehouse-business-intelligence-resources/kimball-techniques/dimensional-modeling-techniques/type-2

15. https://learn.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables

THE PUBLISHER

About Reliable Context

Reliable Context is a system of record for organizations that fuse private and public information and must defend every value they act on.

The guarantees

One canonical record per real thing: duplicates resolve by rule, ambiguity is shown rather than hidden, and related organizations relate without ever being merged. A verbatim, cited source and an observation date sit behind every value. Unknowns are declared as not found, blocked or not applicable with a reason, never left as an empty cell. Any past state of the record is reproducible, together with what changed since.

Where it runs

It runs inside the customer's perimeter, one database per organization. Permitted sources, model routing, reader clearance and retention are set as the customer's own policies, and every value is exportable with its chain of custody for an auditor or regulator.

What it is not

It is not a vector database or a RAG framework, not a data catalog, MDM or data-quality suite, not a CRM or data-enrichment vendor, and not a chatbot or AI assistant. Where a reader needs one of those, the honest answer is to say so and point them there.

Visit

See Reliable Context.

Reliable Context is in production use in two sectors; more than six thousand of its identity decisions were made by rule and none has been reversed. Articles published here explain general practice, cite their primary and academic sources, and name the jurisdiction of any regulatory rule they describe. They are not legal, regulatory or compliance advice for a specific organization, and requirements change.

Website: https://reliablecontext.com

This article is educational. Verify consequential medical, legal, financial, regulatory and technical decisions with appropriate professionals and the cited primary sources. Company services are described separately from the article's research findings.