Iceberg REST Catalog Idempotency

Author: Huaxin Gao

Motivation

Requests and responses for Iceberg REST APIs typically go through multiple components: load balancer, REST catalog server, persistent storage. Mutation calls (like commit) can fail for various transient failures: network, LB, catalog server, or persistent. When failures happen, the mutation may or may not have been committed to the persistent.

It is generally not safe for clients to retry the mutation requests in this case, as it can lead to duplicate resource creation or unintended side effects. Here is an example with updateTable REST API:

- An Iceberg client sends a `POST updateTable` request.
- The catalog server successfully committed the change to persistence. But a transient network error between the catalog server and the persistence leads the catalog server to return an error.
- The client cannot tell if the commit succeeded, so it retries the same commit request.
- On retry, the server detects that the latest table state doesn't match the base from update table request and responds with 409 Conflict.
- The client interprets the 409 as a failed commit and deletes the associated metadata files (manifest list).
- Result: metadata corruption — the table snapshot references a deleted manifest list file.

We have been marking those failures as non-retryable to avoid potential table state corruption. The downside is that now the client cannot recover from transient server failures. This proposal is trying to make the mutation API idempotent, where a request can be retried with no additional side effects. Clients can assume that any non-validation error can be retried until it succeeds, which improves fault tolerance and reduces the likelihood of CommitStateUnknown errors. This doesn't fully fix every failure mode, but it significantly lowers duplicate-effect risk and CommitStateUnknown events.

Goals

- Provide REST Catalog's with a client provided token that they can use to identify retried requests
- Make support for handling idempotency optional so that existing client/server behavior remains unchanged

Non-Goals

- Specify server internals. We do not define how REST Catalogs persist or reconcile idempotency records.
- Replace existing client logic. We do not replace existing client logic; this header is optional and additive.

Proposal

This proposal introduces **idempotency key** for Iceberg REST mutation APIs (e.g., `POST /updateTable`, `POST /createTable`). The goal is to make mutation retries safe in the presence of transient failures, preventing duplicate commits and metadata corruption.

Scope of Idempotency-Key support

Idempotency-Key applies to mutation endpoints only. See Appendix A: Affected REST Endpoints for the complete list.

Standards alignment

This proposal aligns with the IETF HTTPAPI *Idempotency-Key* Internet-Draft: same header name and "first request wins; subsequent requests with the same key return the original finalized result" semantics. Unlike the draft's payload fingerprint, Iceberg's baseline is key-only; no payload hashing/fingerprints are required. Duplicate recognition is by key within the (operation type, resource id) scope. Support is advertised via `idempotency-key-lifetime` (ISO-8601 duration); presence implies support. The Idempotency-Key header remains optional in Iceberg, and clients only enable automatic same-key retries when discovery is present.

Client Responsibilities

- Generate a unique idempotency key per mutation (e.g., each `updateTable` attempt).
- Attach the key in the request header (`Idempotency-Key`).
- Reuse the same key only when retrying the *exact same* request~~ payload~~.
- Do not attempt a retry when the time between the first request and the retry request exceeds the server's advertised idempotency timeout. The server applies a grace period; clients shouldn't add their own.

Server Responsibilities (behavioral, implementation-flexible)

- ~~Bind the key to the request's canonical payload upon first acceptance.~~
- Associate the key **K** with the operation type and resource id upon first acceptance.
- ~~Detect duplicates for the same key:~~ Duplicate handling is key-only within the same (operation, resource) scope:
    - Same key --> do not re-execute; return the stored/prior response.
    - ~~Different payload -> 422 Unprocessable Content (`IdempotencyKeyConflict`).~~
- Record retention/cleanup is implementation-specific, not mandated by the spec.
- Honor window + grace. For a given `(operation, resource, Idempotency-Key)`, the server MUST apply duplicate handling for at least the advertised `idempotency-key-lifetime`, measured from first acceptance of the key, and SHOULD include a grace margin so retries sent near the deadline are not rejected due to transit/skew.

The sequence below summarizes the client/server behavior for Idempotency-Key handling:

Initial flow:

```
Client                          REST Catalog Server
|                                |
|-- POST (Idempotency-Key=K)--> |  first time with K?
|  execute mutation
|                                |  bind K
|<-------------------------------|  Final result (200 OK or terminal
4xx)
```

Retry: same key

```
Client                          REST Catalog Server
|                                |
|-- POST (Idempotency-Key=K)--> |  K known
|<-------------------------------|  original result
```

Error case from Motivation (with Idempotency-Key)

Commit succeeded but response returned 503; duplicate request should not re-execute.

```
Client                          REST Catalog Server

|                                |
```

```
|-- POST (Idempotency-Key=K,    | first time with K? yes
|          op=commit, tbl=T) -->| execute mutation; commit snapshot
|                               | finalize K for (commit,T)
|<------------------------------| 503 CommitStateUnknown(response path timed
out)
|
|-- POST (Idempotency-Key=K,    | K known & FINALIZED for (commit,T)
|          op=commit, tbl=T) -->| DO NOT re-execute; replay prior result
|<------------------------------| 200 OK (replayed)
```

API Changes

New optional HTTP header:

```
idempotency-key:
  name: Idempotency-Key
  in: header
  required: false
  schema:
    type: string
    Format: uuid
    minLength: 36
    maxLength: 36
    example: "550e8400-e29b-41d4-a716-446655440000"
  description: |
    Optional client-provided idempotency key for safe request retries.
```

~~When provided, the server guarantees at-most-once execution for requests with the same key. If a request with this key has already been processed~~

~~successfully, the server returns the original result instead of reprocessing.~~

When present, the server ensures no additional effects for requests that carry

the same Idempotency-Key within the same operation/resource scope. If a prior

request with this key has been finalized, the server returns the previously

finalized response instead of re-executing the mutation.

Finalization rules:

- Finalize & replay: 200, 201, 204, and deterministic terminal 4xx

- Do not finalize (not stored/replayed): 5xx, 409 request_in_progress.

Key Requirements:

- Must be unique per client mutation operation (e.g., updateTable, createTable)

- ~~Should be generated randomly (e.g., UUID v4)~~Key format: UUID (V7 preferred)

- Scoped to operation type and resource path

- Catalogs may expire keys according to the advertised token lifetime.

Best Practices:

- Use UUID.randomUUID() or equivalent

- Reuse the same key for retries of the same logical operation

- Generate new keys for new operations

Note: HTTP header names are case-insensitive; `Idempotency-Key` is the canonical form used in this specification.

Error subtype (for error responses)

Servers should include a machine-readable type field in error JSON for idempotency cases.

```
ErrorResponse.subtype:

  description: Machine-readable error subtype.

  enum: [request_in_progress, idempotency_replay_failed]
```

`request_in_progress` -> use with **409** when a duplicate arrives while the original is still running.
`idempotency_replay_failed` -> use with **5xx** when a prior finalized result cannot be replayed (do not re-execute).

Server behavior:

- If `Idempotency-Key` is provided:
    - Return the original ~~success~~ response for duplicate requests~~ with the same canonical payloads.~~
        - Success: 200/201/204
        - Terminal 4xx~~: e.g., validation/business conflict~~
            - 409 Conflict / AlreadyExists
            - 404 NotFound
            - 422 Unable to process
        - Transient 5xx ~~MUST~~ SHOULD NOT be stored
    - ~~Return 422 Unprocessable Content. Conflict for duplicates with different payloads.~~

Capability Discovery:

- Catalogs SHOULD expose whether idempotency key is supported by returning an ISO-8601 duration for the window:

```
{
  "idempotency-key-respected": true,
  "idempotency-key-lifetime": "30mPT30M"
}
```

idempotency-key-lifetime is the server's acceptance window for duplicates, measured from the moment the server first accepts the key. Catalogs MUST honor duplicates for at least this window and SHOULD include an internal grace period to account for clock

skew and network/queueing delays, so a retry sent just before expiry is still accepted even if it arrives slightly after.

- Client behavior:
    - If the discovery block is absent, or `idempotency-key-lifetime` is missing/invalid, Iceberg clients MUST treat idempotency keys as unsupported and MUST NOT enable automatic same-key retries.
    - Only when `idempotency-key-lifetime is present` and valid may clients enable automatic same-key retries (bounded by the advertised lifetime).

Implementation Scope:

Only the API definition changes and client-side behavior will be implemented in Iceberg. The server-side behaviors described are included for REST Catalog compliance and will be handled by implementations.

Required Iceberg Client Changes

The following updates are required on the Iceberg client side to support idempotent mutation requests:

REST Client Implementation

- Add support for including an optional `Idempotency-Key` header in all REST Catalog mutation requests (POST, PUT, DELETE).
- Allow callers (e.g., Iceberg API users) to specify the key per request.
- Ensure internal HTTP retries send the same key when retrying an operation.
- Use capability discovery from the server to decide whether to retry automatically after transient failures.

Catalog Providers

- Update `RESTCatalog` and any other catalog provider implementations to pass through the `Idempotency-Key` from client configuration or API calls into the REST client layer.

OpenAPI Specification Update

- Modify `open-api/rest-catalog-open-api.yaml` to document the optional `Idempotency-Key` parameter for all applicable mutation endpoints.

Testing

Use a mock REST catalog and mirror the same checks in the Catalog Compatibility Test Kit:

1. **Discovery gating** — When discovery returns `{ "idempotency-key-lifetime": "PT30M" }`, enable idempotent retries; otherwise do not.
2. **Duplicate (finalized)** — First commit succeeds; retry with same key returns the original 2xx and does not re-execute (assert single execution).
3. **In-flight duplicate** — While request #1 runs, request #2 with the same key -> **409** `request_in_progress`.
4. **Lifetime bound** — After the advertised lifetime, client stops auto retries and surfaces `IdempotencyWindowExpired`.

Optional Client-Side Deconfliction Fallback

- For catalogs that do not support idempotency keys, the client may perform lightweight post-commit verification for table updates:
    - If commit status is unknown, fetch the latest table metadata and verify the snapshot matches the intended commit. If a match is found, treat the operation as successful.

Server-Side Implementation (Informational)

When an Idempotency-Key is present, the REST Catalog server identifies duplicates by key within the (operation type, resource id) scope; it does not inspect or compare request payloads. Upon first acceptance, it records an idempotency entry containing the key and its scope (operation type + resource id), plus status (e.g., IN_PROGRESS/FINALIZED) and timestamps. This minimal record is sufficient to recognize later requests with the same key within the same scope as duplicates.

On a request with an Idempotency-Key (within the same operation/resource scope):

• Key unknown -> treat as a new request; create an IN_PROGRESS record and attempt the mutation.

• Key known & FINALIZED -> do not re-execute; return the previously finalized response (2xx or deterministic terminal 4xx). 5xx are not stored.

• Key known & IN_PROGRESS -> return 409 request_in_progress (server MAY block and then return the final result; MAY include Retry-After`).

• Replay failure (finalized but prior response cannot be produced) -> do not re-execute; return 5xx with error type idempotency_replay_failed.

~~Request Hashing (Server-Side)~~

~~When an `Idempotency-Key` is present, the REST Catalog server must SHOULD determine whether a repeated key refers to the same request payload or a different one.~~

~~The server computes:~~

~~`SHA-256(canonicalPayload)`~~

- ~~`canonicalPayload` = JSON serialization of the request body with consistent field ordering and no extraneous whitespace.~~
- ~~This hash is stored with the key for future comparisons.~~

~~On a duplicate request:~~

- ~~If the key is known and the hash matches -> return the stored response (Final response only; transient errors are not stored)~~
- ~~If the key is known but the hash differs -> reject with 422 Unprocessable Content~~
- ~~If the key is unknown -> treat as a new request and attempt the mutation/commit.~~

Required Server-Side Persistence

The server should persist the following for each idempotency record:

- **Scoped key** — combination of operation type, resource path, and `Idempotency-Key`.
- **Status** — IN_PROGRESS or FINALIZED.
- **Stored response** — the HTTP status

- **Timestamps** — creation time (and optionally last accessed time).

Expiration & Cleanup (Server-Side)

Idempotency records should expire after a configurable time period to prevent unbounded storage growth. Expired records should be removed automatically by the server's storage layer or a background cleanup process.

~~Related Work~~

~~This proposal aligns with the IETF HTTPAPI *Idempotency-Key* Internet-Draft: same header name and "first request wins; subsequent requests with the same key return the original finalized result" semantics. Unlike the draft's optional payload fingerprint, Iceberg's baseline is key-only; no payload hashing/fingerprints are required. Duplicate recognition is by key within the (operation type, resource id) scope. Support is advertised via `idempotency-key-lifetime` (ISO 8601 duration); presence implies support. The Idempotency-Key header remains optional in Iceberg, and clients only enable automatic same-key retries when discovery is present.~~

Examples

Example 1a — `updateTable` retry causing corruption

Request 1 (initial commit):

```
POST /v1/tables/my_table/commit
```

- Server commits snapshot `snap-001`.
- Responds with 503 CommitStateUnknown due to a timeout

Client retry (no idempotency key):

```
POST /v1/tables/my_table/commit
```

- Server detects the snapshot already committed.
- Returns 409 conflict.
- Client interprets as a failed commit -> deletes snapshot files from S3.
- **Corruption:** Metadata references `snap-001`, but files are missing.

Example 1b — Same flow with Idempotency-Key

Request 1 (initial commit with key):

```
POST /v1/tables/my_table/commit

Idempotency-Key: abc123
```

- Server commits snapshot `snap-001`.
- Responds with 503 CommitStateUnknown.

Client retry (same key):

```
POST /v1/tables/my_table/commit

Idempotency-Key: abc123
```

- Server looks up `abc123`, finds stored success.
- Returns 200 OK (cached result).
- Client does not delete files.
- No corruption: Metadata and data remain consistent.

Example 2 — Duplicate `createTable` request

Request 1:

```
POST /v1/tables

Idempotency-Key: key-xyz
```

Result: Table created successfully.

Request 2 (retry with same key):
Server returns cached response: **200 OK (table already created)**.

Discussions:

Note: This section illustrates possible implementation choices and edge cases. It is informative and not required for REST Catalog compliance.

In-Flight Duplicates & Key Binding

Scenario. R1 arrives with `Idempotency-Key = K`. While R1 is processing, R2 arrives with the same `K`.

Binding. When the server accepts R1, it **reserves** `K` by creating an `IN_PROGRESS` record for key `K` scoped to `(op, resource)`.

Behavior for subsequent requests using `K`:

- `IN_PROGRESS` -> **409 Conflict** (`request_in_progress`). The server **MAY** include `Retry-After` or block and return the finalized result when available.

When R1 finishes:

- If R1 **succeeds**, mark the key `FINALIZED` and store the original success response (or a sufficient summary to reproduce it). Future requests with `K` return **2xx** with the original body/headers.
- If R1 **fails** with a terminal client error (4xx), store that terminal response. Future duplicates return the same 4xx. Transient 5xx responses **SHOULD NOT** be stored.

Token Lifetime & Client Retry Behavior

What the catalog advertises:

- Via capability discovery, catalogs MAY return: `idempotency-token-lifetime: "<duration>"` (e.g., `"PT30M"`)

What the client tracks:

- When generating a key, the client records firstAttemptTime.
- On each retry, the client compares now – firstAttemptTime to the advertised token lifetime.

Client behavior (intentionally open-ended for apps).

- If elapsed ≤ lifetime -> retry with the same key.
- If elapsed > lifetime -> raise `IdempotencyWindowExpired` and stop automatic retries.

    - The application decides what to do next: resubmit as a new operation with a new key, or give up.

- Servers measure the window from first acceptance and include grace for skew/queueing.

Commit succeeded but idempotency record wasn't finalized

Problem:
A mutation commits successfully, but writing the idempotency record (or finalizing it) fails.

Approach: reserve-then-finalize.

1. Reserve the key before work: create an entry `K` with `status=IN_PROGRESS`.
2. Execute the mutation/commit.
3. Finalize the key to `FINALIZED` and store the result (or minimal metadata).
    - If finalizing fails, the `IN_PROGRESS` row still exists.

Retry behavior (same key).

- Lookup `K`. If it's IN_PROGRESS:
    1. Reconcile by checking table state derived from the operation (e.g., for `updateTable`, confirm the snapshot-id is present with the expected parent-snapshot-id).
    2. If the state reflects the request change, mark `K -> FINALIZED` and return success (original result).
    3. If it doesn't, safely apply the mutation and then finalize.
- If `K` is FINALIZED, return the stored response.
- If the prior response cannot be produced, do not re-execute; return 5xx with error type `idempotency_replay_failed`.

Appendix

Appendix A — Affected REST Endpoints (Mutations Only)

Idempotency-Key is defined for the endpoints below. Read-only endpoints (e.g., GET/list/load/exists) are out of scope.

Tables

- `POST /v1/tables` — create/register
- `POST /v1/tables/rename` — rename
- `DELETE /v1/tables/{identifier}` — drop
- `POST /v1/tables/{identifier}/commit` — commit/updateTable
- `POST /v1/tables/{identifier}/properties` — set properties
- `DELETE /v1/tables/{identifier}/properties` — remove properties

Namespaces

- `POST /v1/namespaces` — create
- `DELETE /v1/namespaces/{namespace}` — drop
- `POST /v1/namespaces/{namespace}/properties` — set properties
- `DELETE /v1/namespaces/{namespace}/properties` — remove properties
- `POST /v1/namespaces/{namespace}/register` — register a table (by metadata file)

Views *(if implemented)*

- `POST /v1/views` — create/register
- `POST /v1/views/rename` — rename

- `DELETE /v1/views/{identifier}` — drop
- `POST /v1/views/{identifier}/versions` — commit/replace definition
- `POST /v1/views/{identifier}/properties` — set properties
- `DELETE /v1/views/{identifier}/properties` — remove properties