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. It can significantly simplify client code. 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.

However, this introduces some complexity on catalog server implementation. The key challenge is how to identify that a request is a repeat of some previous mutation request.

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.

Client Responsibilities

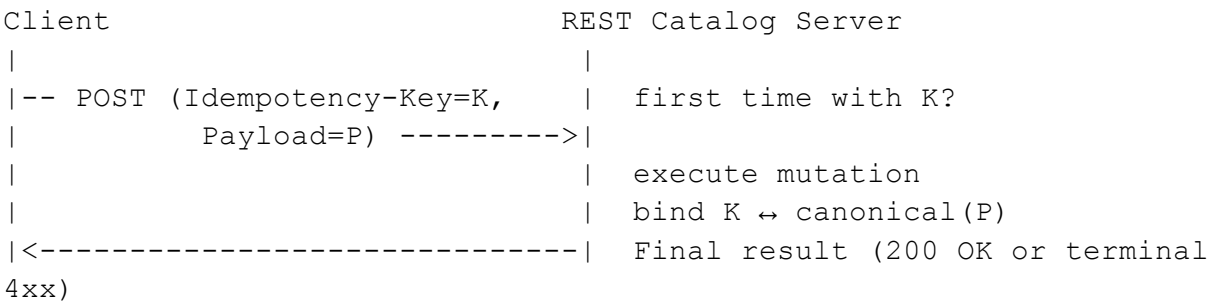
- Generate a unique idempotency key per mutation (e.g., each `updateTable` attempt).
- Attach the key in the request header (`Idempotency-Key`).
- Reuse the same key only when retrying the *exact same* request payload.
- Do not reuse keys across unrelated operations or different payloads.
- Do not attempt a retry when the time between the first request and the retry request exceeds the server's advertised idempotency timeout.

Server Responsibilities (behavioral, implementation-flexible)

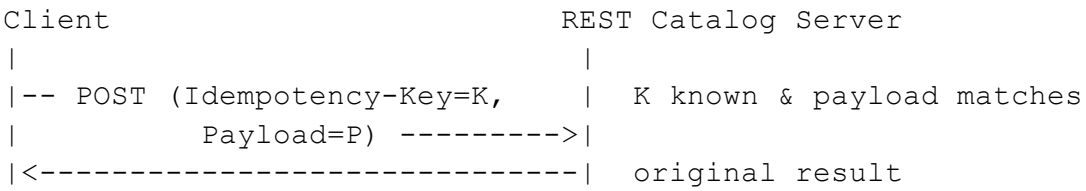
- Bind the key to the request's canonical payload upon first acceptance.
- Detect duplicates for the same key:
    - Same payload -> do not re-execute; return the stored/prior response.
    - Different payload -> 422 Unprocessable Content (`IdempotencyKeyConflict`).
- Record retention/cleanup is implementation-specific, not mandated by the spec.

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

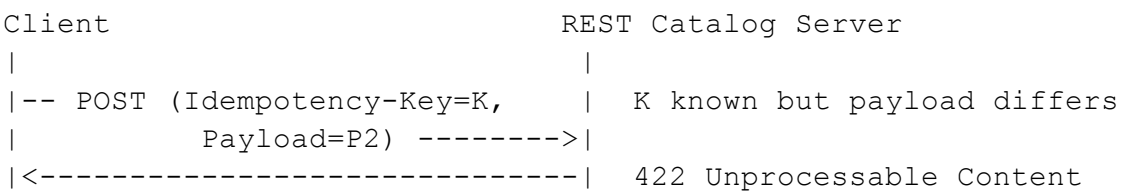
Initial flow:



Retry: same key + same payload



Retry: same key + different payload



API Changes

New optional HTTP header:

```
idempotency-key:
  name: Idempotency-Key
  in: header
  required: false
  schema:
    type: string
    pattern: '^[a-zA-Z0-9][a-zA-Z0-9_.-]*$'
```

```
    minLength: 1

    maxLength: 255

    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.

    Key Requirements:

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

    - Should be generated randomly (e.g., UUID v4)

    - Scoped to operation type and resource path

    - Catalogs may expire keys according to the advertised token life time.

    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.

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

        - Transient 5xx MUST NOT be stored

    - Return 422 Unprocessable Content. Conflict for duplicates with different payloads.

Capability Discovery:

- Catalogs SHOULD expose whether idempotency keys are supported and their retention period in the configuration response:

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

- Client behavior:
    - If the discovery block is absent, or `idempotency-keys-supported` is false/missing, 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 both fields are 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-keys-supported": true, "idempotency-key-lifetime": "30M" }`, enable idempotent retries; otherwise do not.
2. **Duplicate (finalized)** — First commit succeeds; retry with same key+payload returns the original 200/201 and does not re-execute (assert single execution).
3. **In-flight duplicate** — While request #1 runs, request #2 with same key+payload -> **409** `request_in_progress`.
4. **Key conflict** — Same key, different payload -> **422** `idempotency_key_conflict`.

- **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)

Request Hashing (Server-Side)

When an `Idempotency-Key` is present, the REST Catalog server must 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 must persist the following for each idempotency record:

- **Scoped key** — combination of operation type, resource path, and `Idempotency-Key`.
- **Request hash** — `SHA-256` of canonical payload.
- **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; duplicates return the original result" semantics. Our canonical payload hash maps to the draft's idempotency fingerprint. We add explicit scoping (operation + resource path) and a lightweight capability-discovery signal so clients know whether keys are honored and for how long. The Idempotency-Key header remains optional in Iceberg.

Examples

Example 1a — `updateTable` retry causing corruption

Request 1 (initial commit):

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

Payload: { snapshot_id: "snap-001" }
```

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

Client retry (no idempotency key):

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

Payload: { snapshot_id: "snap-001" }
```

- 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

Payload: { snapshot_id: "snap-001" }
```

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

Client retry (same key + payload):

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

Idempotency-Key: abc123

Payload: { snapshot_id: "snap-001" }
```

- Server looks up `abc123`, finds identical payload + 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

Payload: { schema: A }
```

Result: Table created successfully.

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

Example 3 — Conflict with different payload

Request 1:

```
POST /v1/tables

Idempotency-Key: key-xyz

Payload: { schema: A }
```

Result: Success.

Request 2 (same key, different payload):

```
POST /v1/tables

Idempotency-Key: key-xyz

Payload: { schema: B }
```

Result: 422 Unprocessable Content — key already used with different requests**.**

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` and payload `P1`. While R1 is processing, R2 arrives with the same `K` but payload `P2`.

Binding. When the server accepts R1, it **reserves** `K` by creating an `IN_PROGRESS` record with `hash = H(P1)` (where `H` is the SHA-256 of the canonical payload).

Behavior for subsequent requests using `K`:

- `IN_PROGRESS` + **same payload** (`H(P) == H(P1)`) → **409 Conflict** (`request_in_progress`). The server **MAY** include `Retry-After` or block and return the finalized result when available.

- `IN_PROGRESS` + **different payload** (`H(P) != H(P1)`) → **422 Unprocessable Content** (`idempotency_key_conflict`). Do **not** execute `P2`.

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` + same payload return **200/201** 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.

- If callers want to submit a **different** payload after a failure, they **MUST** use a **new** idempotency key.

Scenario:

- R1 arrives with Idempotency-Key = K and payload = P1.

- While R1 is still processing, R2 arrives with the same key K but payload = P2.

Decision:

- The server binds K to P1 as soon as R1 is accepted (creates an IN_PROGRESS record with `hash = H(P1)`).

- Any later request using K is checked against that binding:

    - Same key + same payload (P1) while R1 is still running -> 409 Conflict (request in progress) *(a catalog may optionally block and then replay the final result when R1 finishes)*.

    - Same key + different payload (P2) -> 422 Unprocessable Content. Do not re-execute with P2.

Why return an error even if R1 might fail later?

- An idempotency key must map to one exact request. Allowing K to switch from P1 -> P2 creates races and breaks the at-most-once guarantee.

- When R1 finishes:

    - If R1 succeeds, future (K, P1) retries get the stored success.

    - If R1 fails, callers should resubmit P2 with a new key if they want to try a different payload.

Token Lifetime & Client Retry Behavior

What the catalog advertises:

- Via capability discovery, catalogs MAY return:
    - `idempotency-tokens-respected: true|false`
    - `idempotency-token-lifetime: "<duration>"` (e.g., `"30m"`)

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 and same canonical payload.
- 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.
- Do not silently continue retrying with an expired key.

Commit succeeded but idempotency record wasn't finalized

Problem:
A mutation commits successfully, but writing the idempotency record (or finalizing it) fails. How do we avoid "commit done but key missing"?

Approach: reserve-then-finalize.

1. Reserve the key before work: create an entry `K` with `status=IN_PROGRESS` and `hash=H(P)` (canonical hash of the request payload).
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 + same payload).

- Lookup `K`. If it's IN_PROGRESS and `H(P)` matches:
    1. Reconcile by checking table state derived from the payload (e.g., for `updateTable`, confirm the snapshot-id is present with the expected parent-snapshot-id).
    2. If the table already reflects P, mark `K -> FINALIZED` and return success (original result).
    3. If the table does not yet reflect P, safely apply the mutation and then finalize.
- If `K` is FINALIZED and hashes match -> return the stored response.
- If hashes differ -> reject (e.g., 422 Unprocessable Content / idempotency key conflict).

No need to persist key in snapshot

- We do not need to persist the `Idempotency-Key` in snapshot metadata.
- The retry sends the same payload; the server compares its hash to the IN_PROGRESS record and uses that trusted payload to verify table state.