Iceberg Fine-Grained Metadata Commits

Author: Drew Gallardo (dru@amazon.com)

Motivation

Today, committing data to an Iceberg table means the Iceberg client must construct manifests, manifest lists, and a snapshot that represents the file-level changes (new data files, removed data files, and delete files). That complexity is a barrier for lightweight services that can produce data files like Parquet but do not implement Iceberg's metadata machinery.

At the same time, the catalog's current role is narrow: validate a few table requirements (e.g., snapshot id) and atomically swap metadata.json. The catalog does not see intent (append vs overwrite vs compaction) unless it traverses metadata and computes a diff.

But catalogs have complete context about the table. With the right API, we can let clients simply declare the file level changes via the REST API, and the catalog can perform concurrency checks and construct the new state of the table. This opens the door for lightweight clients without weakening Iceberg's commit guarantees.

In this proposal, we aim to extend the UpdateTable REST API with actions that enable fine-grained metadata commits. Clients submit file level changes, and the catalog constructs and commits the snapshot while enforcing Iceberg's isolation guarantees.

Goals

Goal 1: Enable lightweight Client Integration

Services can write to Iceberg tables by sending the files they produced and the isolation levels to enforce to the catalog, without requiring a dependency on the iceberg library or metadata construction logic.

For example, No Iceberg library dependency required, just ability to produce Parquet files

- **Before**: Iceberg library implements manifest construction, manifest list building, snapshot creation, and commit process.
- **After**: Client sends files to be appended over to catalog

Goal 2: Enable Intent-Aware Capabilities

Once the catalog interprets intent from the request payload, it can act in real time without diffing snapshots. Making intent explicit unlocks opportunities for:

- **Governance Enforcement**: check privileges directly.
- **Downstream System Integration**: emit structured change events (e.g., “append these files”) to CDC pipelines, caches, or lineage trackers.
- **Incremental MV Refresh**: materialized view engines can consume file-level changes directly, avoiding snapshot traversal.

For example, suppose the catalog supports the following write privileges (not standardized here, just illustrative):

- **INSERT:** Users can only submit payloads that add new DataFiles to the table.
- **MODIFY:** This allows users to make changes to existing data, including removing DataFiles or adding DeleteFiles.
- **ADMIN:** Admins have full access to perform all operations against a table.

Non-Goals

To clarify the scope, the following are **not** part of this proposal:

Non-Goal 1: Asynchronous Streaming Append API

A popular use case is performing many append operations to a table, and committing asynchronously in batches, such as when doing Kafka streaming to Iceberg. An asynchronous streaming append API like `POST /namespaces/{namespace}/tables/{table}/append` could allow for appending files in the background, executing asynchronously without waiting for immediate confirmation of the commit. This proposal does not support this use case, as the UpdateTable API is still kept synchronous after our proposed changes, with all the synchronous transactional semantics of the API kept unchanged.

Non-Goal 2: Asynchronous UpdateTable API

Another related use case is to make the UpdateTable API asynchronous for table commits to further improve conflict resolution. Currently, if the server cannot finish the table commit before the synchronous UpdateTable API timeout which is typically very short, the client has to retry the commit. This could lead to the client retrying multiple times, eventually exhausting the retry chances and having to rerun the entire job. If UpdateTable is asynchronous, the server could decide to let one commit wait, resolve any conflicts at server side and then let it safely commit. The client just continuously polls for commit completion status during the process. This would further increase the success rate of concurrent Iceberg table commits. This proposal does not cover this use case, but it is an improvement on top of this proposal that could be explored in the future.

Proposal: Iceberg REST Spec Changes

To enable fine-grained metadata commits in Iceberg via REST API, we propose a new TableUpdate action: `ProduceSnapshotUpdate`. This model is designed for fine-grained file-level operations that modify the table's current snapshot. It supports adding new files, removing existing files, and applying row-level filters for deletions. The ProduceSnapshotUpdate action allows for precise control over table updates, resulting in a new snapshot that accurately reflects the desired changes.

The REST API captures file-level changes and necessary information to ensure successful updates.

- **File Changes**: Files to add, files to remove (optionally delete filter).
- **Commit Properties**: Snapshot properties such as branch, or custom metadata properties.
- **Validations List**: List of validations to prevent concurrent modifications conflicts.

ProduceSnapshotUpdate Schema

```
description:
  Produce a new snapshot by applying file-level changes to the current table
  state optionally on a branch. If all commit-validations pass, the server
  commits or stages a new snapshot.
required:
  - action
properties:
  action:
    type: string
    description: The intent of this update.
    enum:
      - append
      - replace
      - overwrite
      - delete
  add-data-files:
    description:
      List of `DataFile` objects to be added to the new snapshot. These files represent new
      data that will be added to the snapshot metadata as part of the operation.
    type: array
    items:
      $ref: '#/components/schemas/DataFile'
  add-delete-files:
    description:
      List of `DeleteFile` objects to be added to the new snapshot. These files represent
      new row-level deletes that will be added to the snapshot metadata as part of the
      operation.
    type: array
    items:
      $ref: '#/components/schemas/DeleteFile'
  remove-data-files:
    description:
      List of `DataFile` objects to be excluded from the new snapshot. These files contain
      old data that is replaced or deleted as part of the operation.
    type: array
    items:
      $ref: '#/components/schemas/DataFile'
  remove-delete-files:
    description:
      List of `DeleteFile` objects to be excluded from the new snapshot. These files contain
      old row-level deletes that are replaced or deleted as part of the operation.
    type: array
    items:
      $ref: '#/components/schemas/DeleteFile'
  delete-row-filter:
    description:
      A filter expression used to identify DataFiles to be removed as a part of this
      operation. All files that match the filter are to be excluded as part of this
      operation.
    $ref: '#/components/schemas/Expression'
  stage-only:
    description:
      Indicates if the new snapshot should be staged in the table metadata
    type: boolean
  branch:
    description:
      The branch where the update should be applied. Defaults to the main branch.
    type: string
  summary:
    description:
      snapshot summary properties to be set on the new snapshot update.
    type: object
    additionalProperties:
      type: string
  commit-validations:
    description:
        validation clauses that must hold at commit time.
    $ref: '#/components/schemas/CommitValidation'
```

Validation Model Design

The goal is to let clients simply describe what changed while the server enforces all concurrency guarantees. Instead of mirroring Java APIs, we expose a small set of orthogonal clauses that cover the known concurrent modification anomalies in Iceberg commits: conflicting new data, conflicting deletes, dangling file references, and unsafe rewrites.

Validations are sent to the server as part of the request and the commit only succeeds if all validations hold. All conflict checks are evaluated from a provided **base-snapshot-id** and are scoped by the provided **filter** or **list of paths**.

Validation Clause	Used For	What it checks	Purpose
not-allowed-added-data-files	Filter-scoped overwrite (copy-on-write), partition replacement, row-level changes (merge-on-read)	No new **DataFiles** matching the conflict filter have been added since the base snapshot.	Prevents **lost updates** by failing the operation if a concurrent operation added **DataFiles** in the filter scope of this operation. Preventing overwrites of unseen concurrent changes.
not-allowed-added-delete-files	Filter-scoped overwrite (copy-on-write), partition replacement, row-level changes (merge-on-read)	No new **DeleteFiles** matching the conflict filter have been added since the base snapshot.	Prevents **lost deletes** by failing the current operation if a concurrent operation added delete files in the filter scope. Preventing commits that would ignore concurrent delete operations.
required-data-files	Data file deletion by path, row-level change submissions (for referenced data files)	All **referenced DataFiles** still exist at commit time.	Prevents acting on data files that have been removed by concurrent operations.
required-delete-files	Delete file deletion by path, row-level change submissions (for referenced delete files)	All **referenced DeleteFiles** still exist at commit time.	Prevents **lost updates** by failing the operation if a concurrent operation removes a DeleteFile that the current operation removes.
not-allowed-new-deletes-for-data-files	File rewrites/compaction, overwrites that replace existing files	No new **DeleteFiles** have been applied to the specific **target DataFiles** since the base snapshot.	Protects against a **lost update** or specifically delete-vs-rewrite.

Implementation oriented checks (e.g., append-only or added-files-match-filter) are treated as constraints and enforced by the catalog using the declared operation, not as concurrency protections. By unifying these into explicit clauses, the REST model is simpler, easier to extend, and still maps directly to Iceberg's mechanism to prevent anomalies. **See Appendix 2 for more information about the existing validations.**

How existing Java API validation methods map to the REST payload

If we strip away validateAddedFilesMatchOverwriteFilter() and validateAppendOnly(), the core validations Iceberg protects against boil down to:

Existing Validations	Validation Payload Fields (notes in appendix 2)

validateFromSnapshot(snapshotId)	base-snapshot-id (common to all)
validateNoConflictingData()	not-allowed-added-data-files
validateNoConflictingDeletes()	not-allowed-added-delete-files
validateNoConflictingDataFiles()	not-allowed-added-data-files
validateNoConflictingDeleteFiles()	not-allowed-added-delete-files
validateFilesExist()	required-data-files
validateDataFilesExist()	required-data-files
validateDeletedFiles()	required-data-files with allowed-remove-operations -> [OVERWRITE, REWRITE]
failMissingDeletePaths()	required-delete-files
validateNoNewDeletesForDataFiles()	not-allowed-new-deletes-for-data-files

Commit Validation Schema

Validations are provided in the REST API as a list of objects instead of a single object with flags similar to the table requirements design today. Each validation declares its own type and parameters. This makes the model composable, extensible, and easier for catalogs to evaluate independently.

```
CommitValidation:
  description: Preconditions the server must enforce at commit time.
  type: object
  discriminator:
    propertyName: type
  oneOf:
    - $ref: '#/components/schemas/NoAllowedAddedDataFiles'
    - $ref: '#/components/schemas/NoAllowedAddedDeleteFiles'
    - $ref: '#/components/schemas/RequiredDataFiles'
    - $ref: '#/components/schemas/RequiredDeleteFiles'
    - $ref: '#/components/schemas/NotAllowedNewDeletesForDataFiles'

NotAllowedAddedDataFiles:
  type: object
  properties:
    type: { enum: ["not-allowed-added-data-files"] }
    filter: { $ref: '#/components/schemas/Expression' }

NotAllowedAddedDeleteFiles:
  type: object
  properties:
    type: { enum: ["not-allowed-added-delete-files"] }
    filter: { $ref: '#/components/schemas/Expression' }

RequiredDataFiles:
  type: object
  properties:
    type: { enum: ["required-data-files"] }
    filter: { $ref: '#/components/schemas/Expression' }
    file-paths:
      type: array
      items: { type: string }
    allowed-remove-operations:
      description:
        Controls which operation types are permitted to have removed the
        required files.
      type: array
      items: { type: enum: ["DELETE", "OVERWRITE", "REPLACE"] }

RequiredDeleteFiles:
  type: object
  properties:
    type: { enum: ["required-delete-files"] }
    filter: { $ref: '#/components/schemas/Expression' }
    file-paths:
      type: array
      items: { type: string }

NotAllowedNewDeletesForDataFiles:
  type: object
  properties:
    type: { enum: ["not-allowed-new-deletes-for-data-files"] }
    file-paths:
      type: array
      items: { type: string }
    filter: { $ref: '#/components/schemas/Expression' }
```

Proposed Iceberg Library Changes

Implementing the REST Models in Java

The Java library will adapt its existing operations to support the REST ProduceSnapshotUpdate contract. Rather than sending constructed snapshot, these operations will send file-level changes and let the catalog construct the metadata.

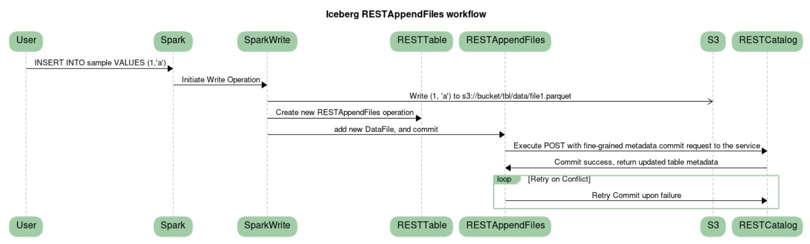
REST Operation Implementations

Each Java operation will implement REST-aware behavior that translates its current state into the ProduceSnapshotUpdate format:

Java Class	REST Action	File Changes	Validation Mapping
AppendFiles	append	add-data-files	None
DeleteFiles	delete	remove-data-files	required-data-files
OverwriteFiles	overwrite (COW)	add-data-files, remove-data-files,	not-allowed-added-data-files, not-allowed-added-delete-files, required-data-files, not-allowed-new-deletes-for-data-files
RowDelta	overwrite (MOR)	add-data-files, add-delete-files	not-allowed-added-data-files, not-allowed-added-delete-files, required-data-files, required-delete-files
RewriteFiles	replace	All four file change types	required-data-files, not-allowed-new-deletes-for-data-files
ReplacePartitions	overwrite	add-data-files	not-allowed-added-data-files, not-allowed-added-delete-files,

Each operation constructs and performs POST requests to the REST service, providing the necessary details, such as the file-level changes, and commit properties. **For more information on how engines interact with these operations refer to Appendix 1.**

For example, when a user performs `INSERT INTO sample VALUES (1,'a')` in Spark-Iceberg, the workflow looks like this:


Iceberg RESTAppendFiles workflow
User
Spark
SparkWrite
RESTTable
RESTAppendFiles
S3
RESTCatalog
INSERT INTO sample VALUES (1,'a')
Initiate Write Operation
Write (1, 'a') to s3://bucket/tbl/data/file1.parquet
Create new RESTAppendFiles operation
add new DataFile, and commit
Execute POST with fine-grained metadata commit request to the service
Commit success, return updated table metadata
loop
[Retry on Conflict]
Retry Commit upon failure

1. `Spark` resolves the query and starts the data writing process, invoking `SparkWrite` to handle the file operations.
2. `SparkWrite` writes the data to storage and initiates a new `RESTAppendFiles` operation to append the new file to the table.
3. `RESTAppendFiles` sends a `POST` request with the file changes to the `RESTCatalog` service.
4. The `RESTCatalog` validates the request, applies the changes, and returns the updated table metadata.
5. `RESTAppendFiles`, then updates the current metadata to reflect the changes.

Appendix

Appendix 1: Engine Interactions with Data Operations

Spark

The following table shows how Spark's DML operations interact with Iceberg's SnapshotUpdate classes and methods for both merge-on-read and copy-on-write tables.

SPARK SQL	Snapshot Update Class	File-Level Changes	Snapshot Property methods	Validations
INSERT (BATCH)	AppendFiles (MergeAppend)	`appendFile(DataFile file)`	`operation.set("spark.app.id", applicationId)` `additionalProperties.forEach(operatio`	

			`n::set)`
			`operation.set(SnapshotSummary.STAGED_WAP_ID_PROP, wapId)`
			`operation.stageOnly()`
			`operation.toBranch(branch)`	

INSERT (STREAMING)	AppendFiles (FastAppend)	`appendFile(DataFile file)`	`operation.set("spark.app.id", applicationId)`
			`additionalProperties.forEach(operation::set)`
			`operation.set(SnapshotSummary.STAGED_WAP_ID_PROP, wapId)`
			`operation.stageOnly()`
			`operation.toBranch(branch)`
			
			`snapshotUpdate.set(QUERY_ID_PROPERTY, queryId)`
			`snapshotUpdate.set(EPOCH_ID_PROPERTY, Long.toString(epochId))`	

DELETE (STREAMING)	DeleteFiles (StreamingDelete)	`deleteFromRowFilter(Expression expr)`	`operation.set("spark.app.id", applicationId)`
			`additionalProperties.forEach(operation::set)`
			`operation.set(SnapshotSummary.STAGED_WAP_ID_PROP, wapId)`
			`operation.stageOnly()`
			`operation.toBranch(branch)`	

DELETE/UPDATE/MERGE (OverwriteByFilter BATCH)	OverwriteFiles (BaseOverwriteFiles)	`overwriteByRowFilter(Expression expr)`
		`appendFile(DataFile file)`
			`operation.set("spark.app.id", applicationId)`
			`additionalProperties.forEach(operation::set)`
			`operation.set(SnapshotSummary.STAGED_WAP_ID_PROP, wapId)`
			`operation.stageOnly()`
			`operation.toBranch(branch)`
				`validateFromSnapshot(Long snapshotId)`
				`validateNoConflictingDeletes()`
				`validateNoConflictingData()`

DELETE/UPDATE/MERGE(Dynamic Batch)	ReplacePartitions (BaseReplacePartitions)	`appendFile(DataFile file)`	`operation.set("spark.app.id", applicationId)`
			`additionalProperties.forEach(operation::set)`
				`validateFromSnapshot(Long snapshotId)`
				`validateNoConflictingData()`
				`validateNoConflictingDelete`

			operation.set(SnapshotSummary.STAGED_WAP_ID_PROP, wapId)
			operation.stageOnly()
			operation.toBranch(branch)
				s()

DELETE/UPDATE/MERGE(Copy-On-Write)	OverwriteFiles (BaseOverwriteFiles)	appendFile(DataFile file)
		deleteFile(DataFile file)
			operation.set("spark.app.id", applicationId)
			additionalProperties.forEach(operation::set)
			operation.set(SnapshotSummary.STAGED_WAP_ID_PROP, wapId)
			operation.stageOnly()
			operation.toBranch(branch)
				validateFromSnapshot(Long snapshotId)
				validateNoConflictingData()
				validateNoConflictingDeletes()
				conflictDetectionFilter(Expression expr)

(StreamingOverwrite)	OverwriteFiles (BaseOverwriteFiles)	overwriteByRowFilter(Expressions.alwaysTrue())
		appendFile(DataFile files)
			operation.set("spark.app.id", applicationId)
			additionalProperties.forEach(operation::set)
			operation.set(SnapshotSummary.STAGED_WAP_ID_PROP, wapId)
			operation.stageOnly()
			operation.toBranch(branch)
			
			snapshotUpdate.set(QUERY_ID_PROPERTY, queryId)
			snapshotUpdate.set(EPOCH_ID_PROPERTY, Long.toString(epochId))
				validateNoConflictingDeletes()

DELETE/UPDATE/MERGE(Merge-On-Read)	RowDelta (BaseRowDelta)	addRows(DataFile file)
		addDeletes(DeleteFile file)
			operation.set("spark.app.id", applicationId)
			additionalProperties.forEach(operation::set)
			operation.set(SnapshotSummary.STAGED_WAP_ID_PROP, wapId)
			operation.stageOnly()
			operation.toBranch(branch)
				conflictDetectionFilter(Expression expr)
				validateDataFilesExist(Iterable<? extends CharSequence> referencedFiles)
				validateFromSnapshot(Long snapshotId)
				validateDeletedFiles()
				validateNoConflictingDeleteFiles()
				validateNoConflictingDataFiles()

Trino

Trino currently only supports merge-on-read operations in the Iceberg connector. Meaning Trino, does not rewrite entire data files when performing DELETE/UPDATE/MERGE statements. Instead it tracks row-level-changes using position based deletes. The following table shows how

Trino interacts with Iceberg Snapshot Update operations:

TRINO SQL	Snapshot Update Class	File-Level Changes	Snapshot Property methods	Validations
INSERT	AppendFiles (BaseAppendFiles/newFastAppend)	`appendFile(DataFile file)`	`operation.set(TRINO_QUERY_ID_NAME, session.getQueryId());`	
DELETE	DeleteFiles (StreamingDelete)	`deleteFromRowFilter(Expression expr)`	`operation.set(TRINO_QUERY_ID_NAME, session.getQueryId());`	

DELETE/UPDATE/MERGE	RowDelta (BaseRowDelta)	`addDeletes(DeleteFile file)`
		`addRows(DataFile file)`
			`operation.set(TRINO_QUERY_ID_NAME, session.getQueryId());`	`validateFromSnapshot(Long snapshotId)`
				`validateNoConflictingDataFiles()`
				`validateDeletedFiles()`
				`validateNoConflictingDeleteFiles()`
				`conflictDetectionFilter(Expression expr)`

Appendix 2: Java Validations

For reference, here is how the java validations are used today for each operation.

Validation	DeleteFiles	OverwriteFiles	RowDelta	ReplacePartitions	RewriteFiles	Depends On	Description
validateFromSnapshot(Long snapshotId)	✓	✓	✓	✓	✓		Validations will check changes after this snapshot ID
conflictDetectionFilter(Expression expr)		✓	✓	✓	✓		Sets a conflict detection filter used to validate concurrently added data and delete files
validateFilesExist()	✓					`removed-data-files`	Validates the removed DataFiles still exist upon commit
validateAddedFilesMatchOverwriteFilter()		✓				`overwrite-filter` `added-data-files`	Validates that the added DataFiles match the overwrite-filter

validateNoConflictingData()		✓		✓		`validateFromSnapshot()`
						`conflictDetectionFilter()`
							Validates that no DataFiles have been added concurrently that match the conflict detection filter

validateNoConflictingDeletes()		✓		✓		`validateFromSnapshot()` `conflictDetectionFilter()`	Validates that no DeleteFiles have been added concurrently that match the conflict detection filter
validateDataFilesExist(Iterable<? extends CharSequence> referencedFiles)			✓			`validateFromSnapshot()`	Adds the DataFiles referenced by the DeleteFiles to be validated
validateDeletedFiles()			✓			`validateDataFilesExist(...)`	Validates that the DataFiles referenced by the new DeleteFiles exist
validateNoConflictingDataFiles()			✓			`validateFromSnapshot()` `conflictDetectionFilter()`	Validates that no DataFiles have been added concurrently that match the conflict detection filter
validateNoConflictingDeleteFiles()			✓			`validateFromSnapshot()` `conflictDetectionFilter()`	Validates that no DeleteFiles have been added concurrently that match the conflict detection filter
validateAppendOnly()				✓		`added-data-files`	Validate that no partitions will be replaced and the operation is append-only

It's important to note that while `validateNoConflictingData()` and `validateNoConflictingDataFiles()` or `validateNoConflictingDeletes()` and `validateNoConflictingDeleteFiles()` are listed separately in the table, they essentially serve the same purpose: validating that no `DataFiles` or `DeleteFiles` have been added concurrently that match the conflict detection filter. The key difference is the specific operation in which they are used.

Furthermore, both `validateDataFilesExist()` and `validateDeletedFiles()` validations have been merged into a single row, as they both depend on one another and serve the purpose of validating that the `DataFiles` referenced by the new `DeleteFiles` exist.