Iceberg UpdateTable Fine-Grained Metadata Commit Support

Author: Drew Gallardo (dru@amazon.com)
WIP PR: https://github.com/apache/iceberg/pull/9237

Motivation

Based on our experience implementing the REST catalog specification we are proposing these changes to improve performance and enrich API semantics. This document is a breakdown of the data operations within Iceberg and a proposal to streamline snapshot creation to the REST catalog service.

Currently, the process of committing data to an Iceberg table is linked to the 5 data operations, namely AppendFiles, OverwriteFiles, DeleteFiles, RewriteFiles and RowDelta. See ⚓ Appendix 1: Iceberg Data Commit Operations for more details about these data operations. Ultimately these are responsible for managing changes to table metadata by adding, or removing data to table metadata, and are critical to creating a new snapshot for reflecting these changes.

By shifting the responsibility of the data operations to the REST catalog, we are essentially granting catalogs with fine grained control over the tables metadata, allowing for several key benefits.

Goals

- Standardize fine-grained metadata commits through a well-defined REST API for appends, deletes, overwrites.
- Ensure **synchronous** processing for all data operations, where clients receive updated metadata after a commit.
- Enable centralized governance by enforcing permissions and user intent (e.g., inserts, deletes, overwrites).
- Enable the REST Catalog to improve conflict resolution strategies.

Goals

by moving the client side data commit logic to the service side, the use cases enable us to have control of the data committing process. See ⚓ Appendix 3: Iceberg Commit Workflow for more details on the current client-side data committing process.

Use Case 1: Improve commit conflict resolution mechanism

By shifting the data commit process to the service, we empower the catalog to take full control of the conflict resolution, allowing it to manage operations more effectively. This gives the catalog the flexibility to determine how to handle conflicts in real time, ensuring smoother concurrent operations without manual intervention from the client.

For instance, we have seen a common issue arise for Spark users, when multiple operations, such as rewrite (compaction) and overwrite operations, are executed concurrently. These operations often conflict because both try to modify the table's metadata at the same time, causing a commit conflict validation failure.

Example: Compaction and Update conflict

Nowadays, users have the option to leverage third-party solutions to perform compaction on their tables or they can set up Spark procedures to run automatically in the background. While these jobs optimize the datafiles and metadata, by merging smaller files into larger ones, they can conflict with overwrite operations, as both attempt to modify the same table metadata concurrently. In the current model, this typically results in one operation failing, requiring users to manually retry or resolve the conflict.

How the Catalog can Manage this Conflict

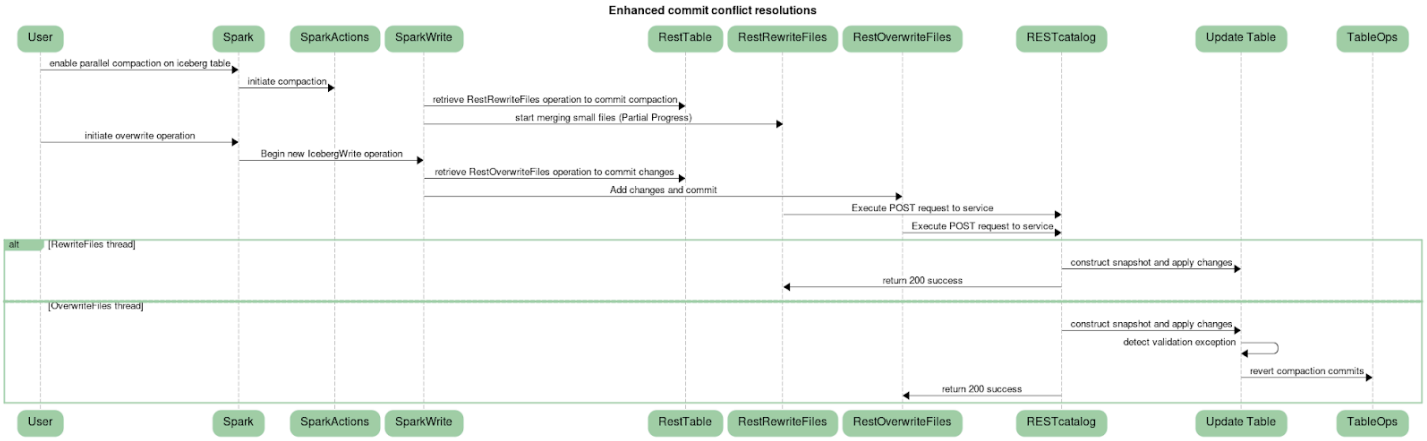
With the commit enabled catalog, the catalog has the flexibility to detect and manage such conflicts. In the case where a rewrite operation is already in progress and an overwrite operation is initiated, we can introduce a check. This check is responsible for identifying if the failure is due to compaction, if so we will:

1. Revert the rewrite operation/s
2. Allow the overwrite operation to commit it's change
3. rerun the compaction operation

By introducing this new change we can reduce the amount of exceptions thrown to the user, and ensure table updates are always reflected.

Some Iceberg REST catalog vendors like Tabular implement this feature with the current UpdateTable API, but we believe the proposed changes will drive more efficiency and also allow more graceful handling of other commit conflicts.

Enhanced commit conflict resolution workflow



Use Case 2: Improve concurrent append files operations

Append operations, just add new data to a table without modifying existing data. Unlike overwrite operations, which can lead to conflicts if concurrent changes modify the table. Therefore, In situations where users are

frequently committing data to a table especially in parallel workloads, append operations can be robust and conflict free as they dont interfere with existing data.

By shifting the data commits to the REST catalog we can allow the service to implement their own mechanism on top of Icebergs retry logic to ensure operations that should succeed have a higher success rate.

Concurrent Write Performance

Given this context, we explore the success rate of committing data to a table using Icebergs Glue catalog and our implementation of the REST catalog with data commits enabled. The Glue catalog allows for up to 4 retries for data operations.

With our restful data commits the retry mechanisms are as follows:

Native Iceberg client-side retry logic: This involves leveraging Iceberg's exponential retry logic in the context of restful data operations. This process allows for up to 4 retries and initiates retries when a CommitFailedException is returned by the server.

REST service with retry logic: this involves the service ingesting the files from an append files request, and invoking their own retry logic.

(threads, operations)	GlueCatalog	REST (with data commits enabled)
(5, 20)	Failures: 5	Failures: 0
(10, 50)	Failures: 23	Failures: 0
(20, 100)	Failures: 72	Failures: 0
(25, 200)	Failures: 152	Failures: 0
(30, 1000)	Failures: 813	Failures: 1
(30, 2000)	Failures: 1643	Failures: 2

This analysis clearly demonstrates that the REST catalog outperformed the Glue catalog in terms of success rate.

Use Case 3: Iceberg integration with non-JVM languages

Additionally, moving data operations into the REST catalog enables easier integration of Iceberg with non-JVM languages. Moving the logic to the catalog service lightens the load on the client, because the catalog service now handles the process of committing data and constructing a snapshot as opposed to the client. This means by following the REST OpenAPI specification the catalog service will be responsible for ingesting the changes and following the process of writing manifests, constructing the diff and creating a manifest list even if written in non-JVM languages.

Use Case 4: Enforcing Governance

With the new protocol, we directly capture the user's intent behind each operation. Whether it's an **INSERT**, **DELETE**, or **UPDATE**. giving the catalog insights into the user's actions and allowing it to enforce permissions more effectively.

- **Enforcing Permissions**: The catalog knows whether a user is appending new data, deleting files, or performing an overwrite. It can strictly enforce permissions according to the user's actions.
- **Governance**: Policies can be enforced more precisely, such as allowing users to append data but blocking deletes or overwrites, creating stronger governance.

Example Operation Breakdown:

- **INSERT**: Only data files are added.
- **DELETE**: Involves adding delete files, removing data files, or applying a delete filter.
- **UPDATE**: A combination of the INSERT and DELETE payloads.

Example: Enforcing Permissions Through Intent

Suppose User A grants User B access to a table, but only with READ and INSERT permissions. In this scenario:

- If User B submits a payload with only DataFiles for appending, the catalog recognizes this as an INSERT and successfully updates the table.
- If the payload contains DeleteFiles, DataFiles for removal, or a delete filter, the catalog detects that User B is attempting to modify or delete existing data. Since User B only has INSERT permissions, the catalog rejects the operation.

This ability to interpret the user's intentions from the payload ensures that the catalog enforces governance in real time, preventing unauthorized operations and aligning with the permissions granted by User A.

Non-Goals

- Async operation commits executing in the background or in a job where updates are processed behind the scenes without returning the updated metadata immediately.
- Async polling for operation completion where the client would wait for the server for completion.

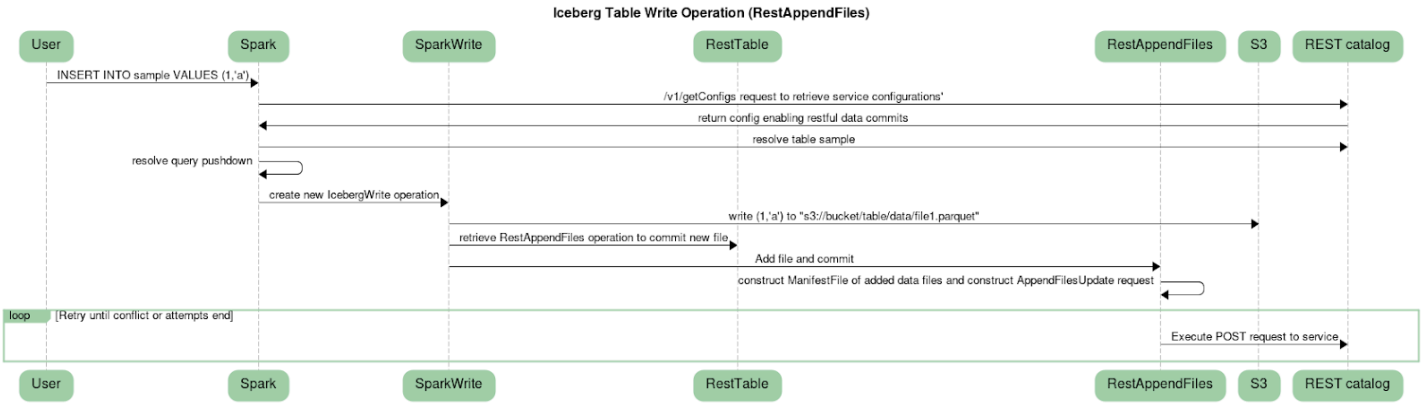
Proposal

Proposal Part 0: overall workflow between client and service

The current workflow for committing data happens on the client within the Iceberg library. The current workflow can be found here: ⚓ Appendix 4: Current data commit workflow

By shifting the data commit process to the REST service. With the introduction of restful data commits we can make a POST request to the service with a metadata update of one of the 5 data operations.

When a user performs `INSERT INTO sample VALUES (1,'a')`.



Proposal Part 1: Restful Data Operations

To achieve this, we are proposing restful data operations to enable restful catalog implementers with the ability to manage data operations. In this proposal we are focusing on these 5 data operations:

- **RESTAppendFiles**: Handles appending data files to tables.
- **RESTDeleteFiles**: Manages deletions from a table whether it be through an expression or the deletion of data files from tables.
- **RESTOverwriteFiles**: Overwrites files from a table including expressions for overwrites, and conflicts
- **RESTRewriteFiles**: Adding a set of files and removing another set of files.
- **RESTRowDelta**: Handles appending set of files and adding a set of position or equality delete files

Each operation constructs and performs POST requests to the REST service, equipped with all necessary details for the service to execute against the table. This necessary information includes the DataFiles and Expressions, and other metadata.

Proposal Part 2: Managing DataFiles with Manifests

In our approach to manage DataFiles in the restful data operations, we are relying on the client to create the DataFiles and send the file locations to the REST service. It will then be up to the service to ingest the files and commit the changes to the table.

To manage this effectively, we propose these three strategies:

1. **DataFiles**: Send the service a direct list of DataFile locations constructed during a commit.
2. **ManifestFile**: When committing changes, this strategy compiles a list of ManifestFiles that hold the DataFile changes for table operations. It then sends the locations of these ManifestFiles. This process leverages the RollingManifestWriter, which determines both the size and the number of the ManifestFiles created. Meaning, users have the flexibility to configure the size of these Manifests using the `commit.manifest.target-size-bytes` property.
3. **ManifestList**: In scenarios involving extensive data operations, we can construct a ManifestList, which would be a single file consisting of Multiple of the aforementioned ManifestFiles containing the DataFile changes.

For our initial prototype, we have opted for the ManifestFile strategy. This strategy offers a balanced approach, providing reasonable performance, and allows for configurability with the target size property. The choice between these strategies can ultimately be configurable, allowing users to select the most suitable approach based on their operational scale and requirements.

Proposal Part 4: REST Catalog Changes

To commit files, a client first indicates its intention to append files to a table. It then retrieves the operation from the Table returned from the REST catalog, which is an instance of Icebergs `BaseTable` class. Therefore, to leverage the aforementioned REST data operations, it's essential to introduce a REST table inheriting the BaseTable class. This REST Table determines if the commit should be delegated to the server.

Furthermore, we will also introduce a configuration named `rest-data-commit-enabled`. When initializing the rest catalog, the client requests the server for configurations. The server then decides whether to enable this configuration. We then will pass this into the REST table and if enabled the new data operations will be returned.

Now we can see that commit conflict/resolution logic is now delegated to the service. Which allows the server to have full freedom to modify the conflict resolution workflow.

Proposal Part 5: client side OpenAPI change

In order to make data operations requests against the service we will need to improve the OpenAPI models

UpdateTable API Model changes

It's crucial to introduce specific table update models that correspond to each data operation. Furthermore, since all these models are considered Table Updates they will all share the same endpoint and request JSON structure.

Following the structure of the UpdateTable API we will be extending the updates section of the request to include the changes for each model. Thus, the following models will be rolled out:

AppendFilesUpdate

The AppendFilesUpdate model is designed to add new data files via a Manifest containing the appended DataFiles to a table. Its structure is as follows:

```
AppendFilesUpdate:
  type: object
  required:
    - appended-manifests
  properties:
   action:
     type: string
   appended-manifests:
      type: array
```

```
      items:
        type: string
    description: Manifest files of DataFiles appended to a table
```

DeleteFilesUpdate

The DeleteFilesUpdate model is responsible for the removal of specific data files from a table. The model can either require a manifest of DataFiles to be deleted or an expression specifying which files to remove. Its structure is:

```
DeleteFilesUpdate:
 type: object
  - oneOf:
   - required:
     - deleted-manifests
   properties:
     deleted-manifests:
       type: array
       items:
         type: string
       description: Manifest files of DataFiles deleted from a table
     - required:
       - delete-expression
     properties:
       delete-expression:
        $ref: '#/components/schemas/Expression'
  properties:
   action:
     type: string
   case-sensitive:
     type: boolean
```

OverwriteFilesUpdate

The OverwriteFilesUpdate model is for situations where data files in a table are both added and removed. Providing an expression for both conflict detection, and overwrite by row filter. Its structure is given by:

```
OverwriteFilesUpdate:
 type: object
  properties:
   action:
     type: string
   appended-manifests:
     type: array
     items:
       type: string
     description: Manifest files of DataFiles appended to a table
```

```
    deleted-manifests:
      type: array
      items:
       type: string
      description: Manifest files of DataFiles deleted from a table
   overwrite-by-row-filter-expression:
      $ref: '#/components/schemas/Expression'
    conflict-filter:
     $ref: '#/components/schemas/Expression'
   case-sensitive:
     type: boolean
```

RowDeltaUpdate

The RowDeltaUpdate model is for situations at a row level. We track added DataFiles and DeleteFiles. Providing an expression for conflict detection. Its structure is given by:

```
RowDeltaUpdate:
 type: object
  properties:
   action:
     type: string
   appended-manifests:
     type: array
     items:
       type: string
     description: Manifest files of DataFiles appended to a table
    appended-delete-manifests:
      type: array
      items:
       type: string
      description: Manifest files of DataFiles deleted from a table
    conflict-filter:
     $ref: '#/components/schemas/Expression'
   case-sensitive:
     type: boolean
```

RewriteFilesUpdate

The RewriteFilesUpdate model is for situations where data files and delete files are both added and removed from a table. Its structure is given by:

```
RewriteFilesUpdate:
 type: object
  properties:
   action:
     type: string
```

```
  appended-manifests:
    type: array
    items:
      type: string
    description: Manifest files of DataFiles appended to a table
  appended-delete-manifests:
     type: array
     items:
      type: string
     description: Manifest files of DeleteFiles deleted from a table
  deleted-files-manifests:
    type: array
    items:
      type: string
    description: Manifest files of DeleteFiles deleted to a table
   deleted-manifests:
     type: array
     items:
      type: string
     description: Manifest files of DataFiles deleted from a table
```

These models will encapsulate the necessary information and requirements of their respective operations, ensuring the service has the necessary information to infer what change is taking place.

Appendices

Appendix 1: Iceberg Data Commit Operations

The Iceberg UpdateTable APIs are designed to commit data changes to a table while providing concurrency control preventing conflicting operations. If no conflicts occur these operations will result in a new Snapshot committed to the table. Currently there are 5 data operations we are focusing on.

Operation Type	Description	Important methods	API link
AppendFiles	API for appending new files in a table.	• appendFile(DataFile file)	Link
DeleteFiles	API for removing a set of files or delete based on a filter expression. This is typically used for a copy-on-write DELETE, when the delete filter is a bijection to a set of files (e.g. deleting an entire partition).	• deleteFile(DataFile file) • deleteFromRowFilter(Expression expr)	Link

OverwriteFiles	API for adding a set of files and removing a set of files from a table. The contents of the files are logically different. This is typically used for copy-on-write DELETE/MERGE/INSERT/UPDATE.	• addFile(DataFile file) • deleteFile(DataFile file)	Link
ReplaceFiles/ RewriteFiles	API for adding a set of files and removing another set of files. The content of added files and removed files are logically equivalent. This is typically used for file compaction.	• addFile(DataFile file) • deleteFile(DataFile file) • rewriteFiles(Set<DataFile> dataFilesToReplace, Set<DeleteFile> deleteFilesToReplace, Set<DataFile> dataFilesToAdd, Set<DeleteFile> deleteFilesToAdd)	Link
RowDelta	API for removing a set of files and adding a set of position or equality delete files. This is typically used for a merge-on-read DELETE/UPDATE/MERGE.	addDeletes(DeleteFile deletes)addRows(DataFile inserts)	Link

Appendix 2: Iceberg UpdateTable Rest API

In Iceberg, the process of committing updates to a table is divided into two parts, requirements and updates.

1. **Requirements:** These are assertions that must be validated before any changes are made and committed to the table. They act as safety checks to ensure the state of the table aligns with expectations before updates are applied. An example is `assert-ref-snapshot-id`, which verifies if a ref's snapshot ID matches an expected value.
2. **Updates:** These represent the actual changes or modifications to be applied to a table's metadata. For example, after asserting that the current main ref is at a particular snapshot, a commit can append a new child snapshot and then update the reference to the new snapshot ID.

The updates correspond to Iceberg's MetadataUpdate API. The requirements are in place to ensure updates are conflict-free and consistent.

Per the OpenAPI specification, all table updates use a common endpoint. The specific changes are within the request payload. Successful updates receive a 200 status, returning the table's updated metadata. If any requirements aren't met, a 409 status (CommitFailedException) is returned allowing the client to retry. For the full list of responses, see the spec here.

Example: Inserting data into a table would invoke the endpoint containing the payload below

Endpoint:

- **POST**: `/v1/{prefix}/namespaces/{namespace}/tables/{table}`

- **prefix**: catalog ID
- **namespace**: namespace name
- **table**: table name


```
{
 "identifier": {
   "namespace": ["default"],
   "name": "sample"
 },
 "requirements": [
   {
     "type": "assert-table-uuid",
     "uuid": "3aa0de73-af7e-4644-be01-8e356261ddb5"
   },
   {
     "type": "assert-ref-snapshot-id",
     "ref": "main"
   }
 ],
 "updates": [
   {
     "action": "add-snapshot",
     "snapshot": {
       "snapshot-id": 3668384965176171500,
       "timestamp-ms": 1698360493429,
       "manifest-list": "s3://bucket/sample/metadata/snap-uuid.avro",
       "summary": {
         "changed-partition-count": "1",
         "added-data-files": "1",
         "total-equality-deletes": "0",
         "added-records": "1",
         "total-position-deletes": "0",
         "added-files-size": "845",
         "total-delete-files": "0",
         "total-files-size": "845",
         "total-data-files": "1",
         "total-records": "1",
         "operation": "append"
       },
       "sequence-number": 1,
       "schema-id": 0
     }
   },
   {
     "action": "set-snapshot-ref",
     "ref-name": "main",
     "type": "branch",
```

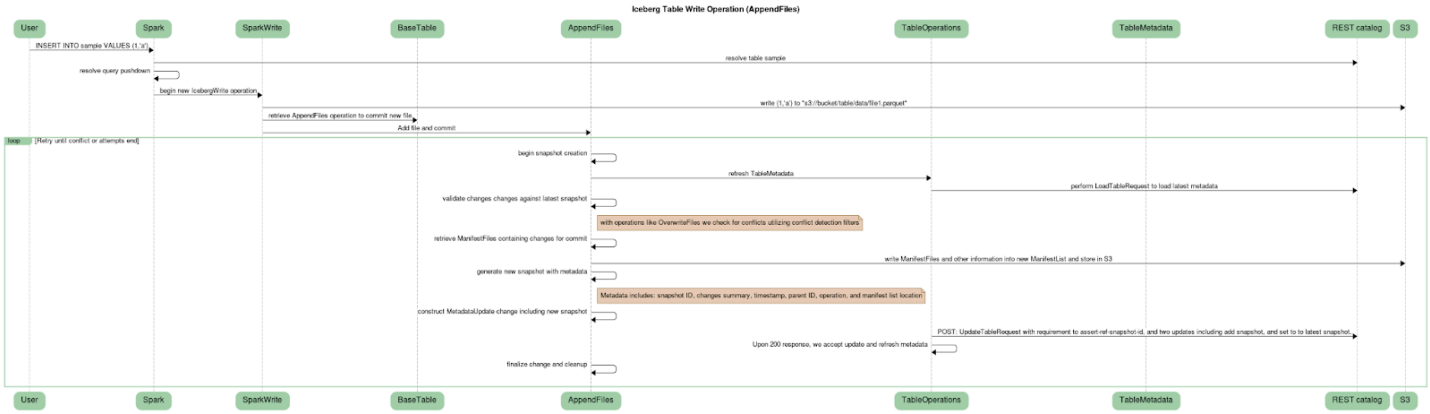
```
      "snapshot-id": 3668384965176171500
    }
  ]
}
```

Appendix 3: Iceberg Commit Workflow

Iceberg's data operations are managed by the SnapshotProducer class, which oversees the creation and committing of table snapshots. A snapshot represents the state of a table at a point in time, including a comprehensive list of DataFiles and the related metadata for operation. During a data operation like AppendFiles, after providing the DataFiles, the SnapshotProducer forms a snapshot, verifies concurrent operations, and then prompts the REST catalog to commit this new snapshot to the table's metadata.

Current Iceberg commit workflow for inserting into an Iceberg table

Consider an Iceberg table named `sample` with columns `(id int, data string).` If we run `INSERT INTO sample VALUES (1, 'a')`, the AppendFiles operation is invoked, and the workflow can be seen below:



In this diagram, you can see how AppendFiles is responsible for collecting the Data files produced by the SparkWrite class and committing the changes to the table. When initiating the commit, the data operation begins the snapshot creation process. After its completion, the client will construct a MetadataUpdate with this new snapshot. The client interprets the update to send to the REST service as a UpdateTableRequest. Which includes the current table UUID to be a requirement to commit the new snapshot, the new snapshot to be added to the table, and an update set then new snapshot as the current snapshot.

The REST service would then ingest this request, assert the current snapshot ID to be equivalent to the one sent in the request. and apply the new snapshot to the table, update it to be the current. and atomically swap the old metadata with the new one containing the changes.

Appendix 4: Optimistic Concurrency Locking Strategy

In the process of committing updates to a table, Iceberg adopts an optimistic concurrency locking strategy.

Meaning the writer (AppendFiles) assumes that the table version won't change before their updates are committed. Writers will generate the necessary metadata files, and attempt to commit their changes by swapping the metadata file pointer from the existing version to the newly created version.

If a conflict arises, meaning the base snapshot our changes are being built against is no longer current. This means the writer must retry by re-applying its changes based on the new current snapshot. For some data operations such as AppendFiles this is okay because we'd likely have no conflicts to the table.

However, a change that rewrites files can be applied to a new table snapshot if all of the rewritten files are still in the table. An example of this would be running an overwrite statement (MERGE/UPDATE/DELETE) at the same time of compaction.

Commit Failed Exception workflow

Given this information, If we create a session and run a compaction on our previously created table `sample`, and meanwhile, we create another session and run a `MERGE` statement (overwrite) we would expect one of the commits to fail.