Multi-Table/Statement Transactions in Iceberg

Motivation

Iceberg currently has support for transactions at the individual table level, allowing reads from a point in time consistent snapshot of a table, and committing updates atomically to a single table. Iceberg also includes an API currently exclusive to the REST Catalog for committing a batch of table commits across multiple tables in a single atomic operation. However, this API only addresses the commit path, and doesn't enable the client to read a consistent point-in-time snapshot across multiple tables.

To address these gaps, we propose two potential approaches for enabling multi-table transactional semantics in Iceberg:

1. A **Global Catalog Sequence Number** model, where the clients operate relative to a pinned sequence of the catalog state.
2. A **Transaction Context API**, where the catalog manages transaction-scoped metadata, enabling clients to stage and isolate table updates, as well as read point in time consistent snapshot across tables.

The goal of this document is to propose new Iceberg Rest Catalog (IRC) APIs to enable engines to build Multi Table and Multi Statement transactions.

Note: This document refers to clients or engines interchangeably. By client/engine, we mean an engine like Spark or a client like PyIceberg that is accessing Iceberg tables via IRC APIs.

Background

What Does It Mean for a Catalog to Support Transactions?

In order for the catalog to support multi-table transactions, the catalog must:

1. Provide functionality for clients to load multiple tables as they existed at a single point in time (SNAPSHOT ISOLATION for reads).
2. Provide functionality to perform conditional atomic commits across multiple tables.

In this document, we focus on two isolation levels for Iceberg multi-table transactions.

Snapshot Isolation

From https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-95-51.pdf

"These discussions naturally suggest an isolation level, called Snapshot Isolation, in which each transaction reads data from a snapshot of the (committed) data as of the time the transaction started, called its Start-Timestamp. This time may be any time before the transaction's first Read. A transaction running in Snapshot Isolation is never blocked attempting a read as long as the snapshot data from its Start-Timestamp can be maintained. The transaction's writes (updates, inserts, and deletes) will also be reflected in this snapshot, to be read again if the transaction accesses (i.e., reads or updates) the data a second time. Updates by other transactions active after the transaction Start-Timestamp are invisible to the transaction."

Read Requirements

For a single transaction, the client should be able to load multiple tables as they existed at a single point in time, likely the time of first table load - called Start-Timestamp. When loading new tables, clients should have the ability to ignore the table snapshots from commits after Start-Timestamp. A read-only multi-table transaction running under SNAPSHOT ISOLATION should not have conflicts, as it simply reads from a point-in-time state of tables in Catalog. Changes from other concurrent transactions should be invisible to this transaction.

Write/Commit Requirements

For a write transaction running under SNAPSHOT isolation, the client should be able to verify that there is no write-write conflict at the time of commit. Iceberg already provides SNAPSHOT isolation for single-table commits, and it needs to be extended to multi-table commits by using the existing Multi-Table commit API in IRC.

Serializable

From https://web.cecs.pdx.edu/~len/sql1999.pdf

"The execution of concurrent SQL-transactions at isolation level SERIALIZABLE is guaranteed to be serializable. A serializable execution is defined to be an execution of the operations of concurrently executing SQL-transactions that produces the same effect as some serial execution of those same SQL-transactions. A serial execution is one in which each SQL-transaction executes to completion before the next SQL-transaction begins."

Read Requirements

Serializable Snapshot Isolation can be implemented using Snapshot Isolated reads, with different conflict detection for writes (discussed below). For single-table transactions, Iceberg already

offers both SNAPSHOT and SERIALIZABLE isolation levels, as all the reads are from a single snapshot of the table. This needs to be extended to Multi Table Transactions..

Write/Commit Requirements

For a write transaction under SERIALIZABLE isolation, the client should be able to assert that there is no read-write conflict - no other transaction has modified the data read by this transaction since Start-Timestamp. This can be done using the existing Multi-Table commit API, by asserting that tables read by transaction have not been modified - Multi-Table commit API in IRC.

Note on conflict detection

As discussed above, for SNAPSHOT isolation the client needs to guarantee/assert that there are no write-write conflicts with other transactions, while for SERIALIZABLE isolation the client needs to guarantee that there are no read-write or write-write conflicts. Different clients can implement this differently - for example some clients can simply assert that the "snapshot id" of the tables being written have not changed since Start-Timestamp. While other clients may employ more fine-grained conflict detection, e.g. if a transaction reads data from a specific partition then client may assert that there have been no writes to that specific partition. Different clients can implement different granularity of conflict detection, but the important part is that Catalog provides an API which clients can use to atomically commit across tables after verifying there are no conflicts.

Limitations Today for Multi Table/Statement Transactions

This section discusses what limitations prevent engines from implementing Multi Table or Multi Statement transactions on Iceberg tables.

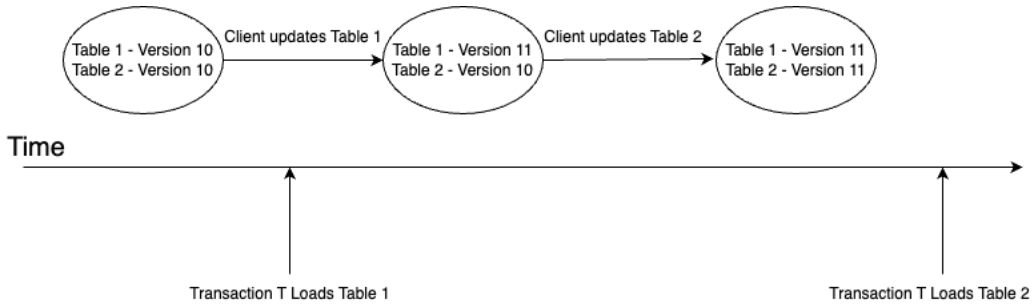
Limitation 1 - No functionality to load **multiple** tables in a single transaction with SNAPSHOT ISOLATION consistency.

Limitation 2 - No functionality to commit to multiple tables atomically, the "compare-and-swap" operation is at table level. This point is here for completeness - an IRC API has been added for this use case already in https://github.com/apache/iceberg/pull/7741. But the PR to use this API is not merged: https://github.com/apache/iceberg/pull/6948 . This PR also adds functionality for clients to "Read their own writes" with-in transaction, which is important for multi-statement transactions.

Example for Limitation 1

Today, a single-statement transaction can read multiple Iceberg tables, with a READ_COMMITTED view of tables. If multiple clients are writing to Iceberg tables in a namespace, there is no Iceberg API to read SNAPSHOT ISOLATION consistent view of multiple tables in the namespace.

Let's assume a namespace with two tables, both at version 10. Then two clients perform 2 sequential writes to each table. At the same time, there was a read transaction T, which was doing a JOIN across those tables. Let's assume the following timeline of operations:



In the above example Transaction T will use Table1-Version10 and Table2-Version11, if it loads Table 1 before the writes and Table 2 after the writes. This read query is now operating on state (Table 1 – Version 10, Table 2 – Version 11) which never existed at any point in time, given the sequential write order above. In the above example, 3 SNAPSHOT ISOLATION consistent states of the namespace were:

- Table 1 Version 10, Table 2 Version 10
- Table 1 Version 11, Table 2 Version 10
- Table 1 Version 11, Table 2 Version 11

In order to support multi table transactions, we propose catalog API changes to load multiple tables from a consistent state.

Can we use “AS OF” operation to achieve this?
Iceberg provides functionality to read historical snapshot of table using “AS OF” query semantics. The timestamp used to find the snapshot is the timestamp provided by client/engine when writing a new snapshot (Reference below).
This cannot be used to read consistent state across tables, as different machines writing to Iceberg tables may have different clock skews. This timestamp does not provide strict ordering of commits across tables.

https://github.com/apache/iceberg/blob/main/core/src/main/java/org/apache/iceberg/SnapshotProducer.java#L301

Requirements

This document proposes new IRC APIs that engines can use to implement Multi-Table Multi-Statement transactions. The new proposed APIs should:

1. Provide functionality for transactions to operate on SNAPSHOT ISOLATION consistent view of all tables in a catalog. Multi-Table Transactions should not see the effect of other transactions with commit time after this transaction's start time - Start-Timestamp.

Non goals

- Provide functionality to implement DDL changes with-in multi-table/multi-statement transactions.

Prior Art

This section references an existing database implementation which provides SNAPSHOT isolation for multi-table transactions. We discuss this at a very high level just to provide context. The description here is by no means complete.

PostgreSQL

PostgreSQL assigns a sequential Transaction ID to every write transaction, in order of how they start writing to the database. Each row version in PostgreSQL has xmin and xmax fields.. xmin is the Transaction ID that created the row version, xmax is the Transaction ID that deleted it, null otherwise.
When a new transaction starts in PostgreSQL, it creates an in memory data structure with these 3 key values, which are used to provide SNAPSHOT ISOLATION reads to transactions:

xmin - *Lowest transaction ID that was still active. All transaction IDs less than xmin are either committed and visible, or rolled back and dead.*
xmax - *One past the highest completed transaction ID. All transaction IDs greater than or equal to xmax had not yet completed as of the time of the snapshot, and thus are invisible*.
xip_list - *Transactions in progress at the time of the snapshot. A transaction ID that is xmin <= **X** < xmax and not in this list was already completed at the time of the snapshot, and thus is either visible or dead according to its commit status.*

Using this data structure, the database knows which version of row to read for a given transaction. Even though this is at row level, the same concept can be used for providing SNAPSHOT isolation for Iceberg Multi-Table Transactions to make sure the transaction reads the right table version.

Details from: https://www.postgresql.org/docs/current/functions-info.html

IRC API Proposal

Option 1 - CatalogSequenceNumber

As discussed in the community sync, an alternative to the Transaction Context model is to introduce a monotonically increasing, catalog-wide sequence number. It acts as a logical clock for the objects in the catalog, and preserves a consistent ordering of write events to be observed by the client. This is a **global** sequence number across the catalog.

In this approach there are two sets of responsibilities in order for this to work.

Catalog Responsibilities

The catalog must support the concept of an strictly monotonically increasing sequence number that reflects the logical progression of catalog changes. The sequence number must provide a stable view of the catalog at a particular sequence number, while implementation details are left to the catalog, it must satisfy the following guarantees:

1. Incrementing the sequence number for each change to the catalog (e.g. creating, dropping or updating catalog objects).
2. Exposing the catalog sequence number to the client.
3. Embed the catalog sequence number in each snapshot.
4. Must have support for the atomic multi-table commit API.

Client Responsibilities

The client's expectation is to coordinate the transactional behavior with the catalog. Specifically the client will be responsible for:

1. Pinning the catalog's current CatalogSequenceNumber at either first table load or first statement.
2. Performing all reads and writes against the catalog "as-of" that sequence. (similar to time travel)
3. Handle all validations for the isolation guarantees and submit all updates using the multi-table commit API.

As long as the catalog guarantees correct CSN ordering and lookup, clients can achieve full snapshot isolation by rolling back metadata or snapshots to the pinned sequence, without requiring stateful transaction contexts proposed in Option 2.

Catalog Based Metadata Versioning

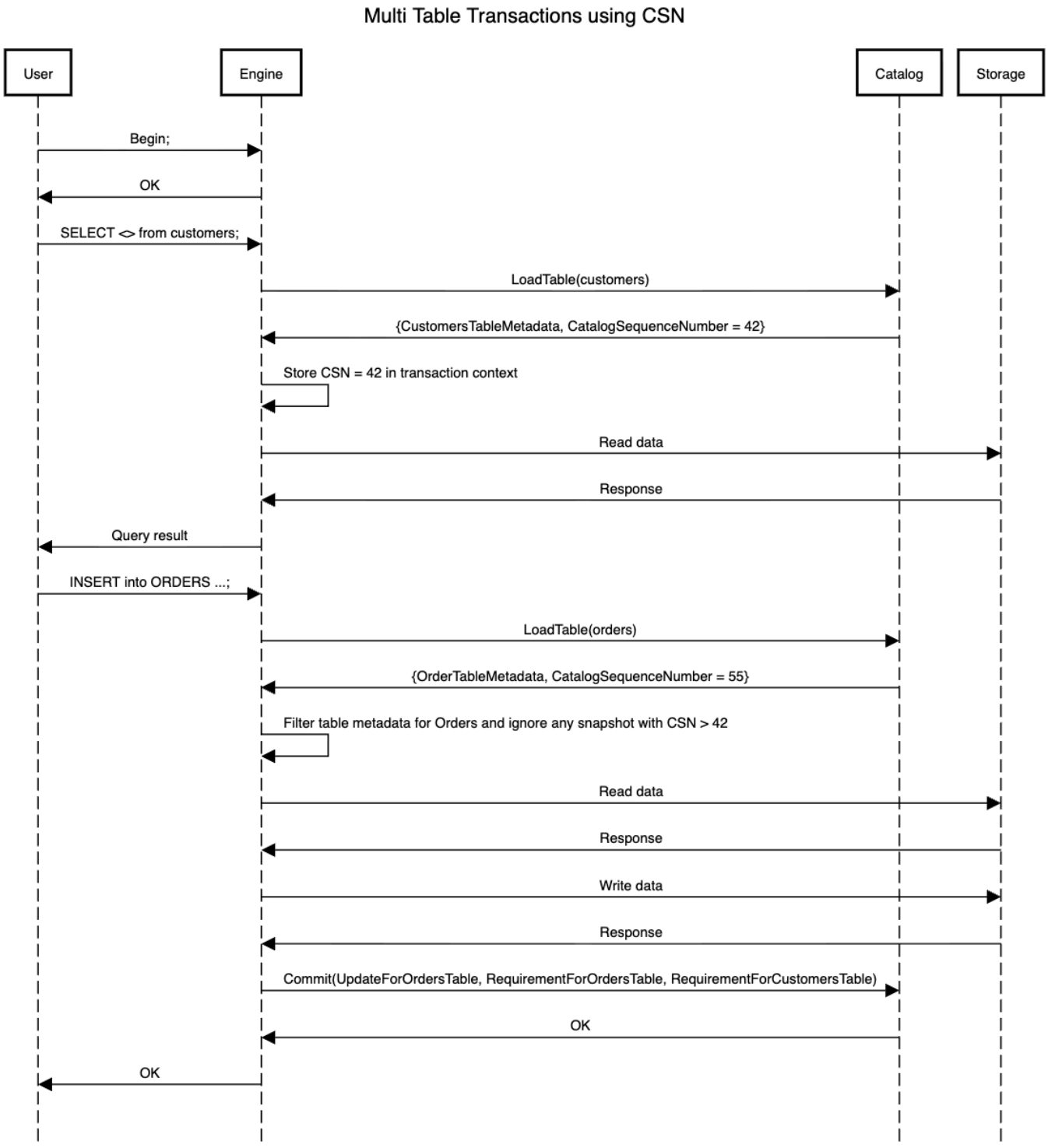
Catalogs also embed their catalog sequence number whenever they ingest new table updates. Each new snapshot record carries the CSN at which it was created, allowing the client/engine to rewind operations and replay updates at any past sequence number while still leveraging the MTT commits.

Note: Initially we only version new snapshot updates and later we can investigate if we need to apply CSN version for all versioned TableUpdates (e.g. schemas, partitions) but I believe snapshots provide the source of truth for data consistency.

Example: AddSnapshot

1. catalog-sequence-number = 99
2. increment catalog sequence number to 100
3. Embed 100 into SnapshotUpdate
4. Apply SnapshotUpdate to latest table metadata and perform atomic swap in catalog

All metadata snapshot shot entries are thus versioned by the catalog CSN. Clients can load the latest metadata, and during a transaction they can time-travel by using the snapshot that is <= to the pinned sequence during the start of the transaction.



REST API Changes

To enable transactional consistency for catalogs that **SUPPORTS** the Catalog Sequencing model, the REST API must provide a way for clients to:

- Obtain the current catalog sequence number to be pinned for subsequent reads
- Read operations as they existed at a specific sequence number

This enables clients to perform all reads and writes within a transaction against a consistent catalog state.

Obtaining the Latest Catalog Sequence Number

Pinning to a catalog sequence number is essential for ensuring point-in-time read consistency. There are two options for clients to get the current catalog sequence number in order to pin for subsequent reads:

Option 1 [PREFERRED]: All Reads include the latest Catalog Sequence Number in response

Every catalog read (e.g. loadTable, listTables, loadNamespace, etc.) includes the catalog sequence number at which the data was read. This sequence number can be leveraged after the first load in the transaction.

Example LoadTable Response

```
LoadTableResult:
 description: ...
 type: object
 required:
   - metadata
 properties:
   metadata-location:
     type: string
     description: May be NULL if the table is staged as part of a transaction
   metadata:
     $ref: '#/components/schemas/TableMetadata'
   config:
     type: object
     additionalProperties:
       type: string
   storage-credentials:
     type: array
     items:
       $ref: '#/components/schemas/StorageCredential'
   latest-catalog-sequence-number:
     type: int
     format: int64
```

Option 2 : Expose a dedicated API CurrentCatalogSequenceNumber()

The REST Catalog provides a dedicated endpoint that returns the current catalog sequence number. This can be leveraged to call the endpoint at the start of a transaction to explicitly pin the latest state.

```
GET /v1/{prefix}/catalog-sequence-number

Example response:
{
  "catalog-sequence-number": "999999999"
}
```

Example Snapshot API Schema in TableMetadata

```
Snapshot:
 type: object
 required:
   - snapshot-id
   - timestamp-ms
   - manifest-list
   - summary
 properties:
   snapshot-id:
     type: integer
     format: int64
   parent-snapshot-id:
     type: integer
     format: int64
   sequence-number:
     type: integer
     format: int64
   timestamp-ms:
     type: integer
     format: int64
   manifest-list:
     type: string
     description: Location of the snapshot's manifest list file
   summary:
     type: object
     required:
       - operation
     properties:
```

```
      operation:
        type: string
        enum: ["append", "replace", "overwrite", "delete"]
      additionalProperties:
        type: string
  schema-id:
    type: integer
  catalog-sequence-number: <- new field added
    type: integer
    format: int64
    description: CSN at snapshot creation
```

Option 2 - TransactionContext APIs

TransactionContext

New API Object that would be managed by Catalog. When an engine/client running multi-table or multi-statement transaction accesses a table first time in a catalog, it will register a TransactionContext with Catalog. Catalog will record a snapshot of tables internally when a TransactionContext is created. The engine/client can read a SNAPSHOT ISOLATION consistent view of all tables using this TransactionContext. It can also commit changes to the TransactionContext to implement “Read your writes” in multi statement transaction. (More details later in the document on how engines/clients will use the TransactionContext).

RegisterTransactionContext - New API

New API that a client/engine calls to register a new TransactionContext within Catalog. Internally, catalog stores a reference/snapshot of state of each table. This snapshot is then used to load tables to provide SNAPSHOT ISOLATION consistent reads using the same transaction context.

Catalog should return a unique transaction-context-id (GUID) for every call to register a transaction context.

```
POST /v1/{prefix}/transaction-contexts
Example response:
{
    "transaction-context": {
        "transaction-context-id": "550e8400-e29b-41d4-a716-446655440000"
    }
}
```

DeleteTransactionContext - New API

Engine calls this API when a transaction finishes (at COMMIT or ROLLBACK) to let Catalog know it can delete the TransactionContext and other metadata for it. Catalog will also implement GC in case client/engine never deletes a TransactionContext.

```
DELETE /v1/{prefix}/transaction-contexts/{transaction-context-id}
```

LoadTables - API change to load table as of TransactionContext

New API field for engine/client to provide a transaction context when loading a new table. Catalog will return the table snapshot as of TransactionContext. Client can update the Table's snapshot in a TransactionContext using the commit table API discussed next.

New parameter in LoadTable API
https://github.com/apache/iceberg/blob/main/open-api/rest-catalog-open-api.yaml#L922

```
- name:transaction-context-id
    in: query
    description:
    An optional transaction-context-id.
    If not provided or empty, load table will return the most up to date
snapshots of table.
    If provided, catalog will return table metadata for the given
transaction-context.
    required: false
    allowEmptyValue: true
    type: string
    example: "550e8400-e29b-41d4-a716-446655440000"
```

Existing Multi-Table commit API

Commit updates to multiple tables in an atomic operation.

```
POST /v1/{prefix}/transactions/commit
```
Reference:
https://github.com/apache/iceberg/blob/main/open-api/rest-catalog-open-api.yaml#L1328

How engines will use Transaction Context APIs?

During a transaction, when an engine needs to load a table for the first time, the engine will call the API to register a new transaction context scoped to that namespace. Engine will keep track of the transaction-context-id during the lifetime of the transaction. This transaction context will be used in two key places:

1. When loading a table from the catalog, the engine will provide the transaction context for the transaction.

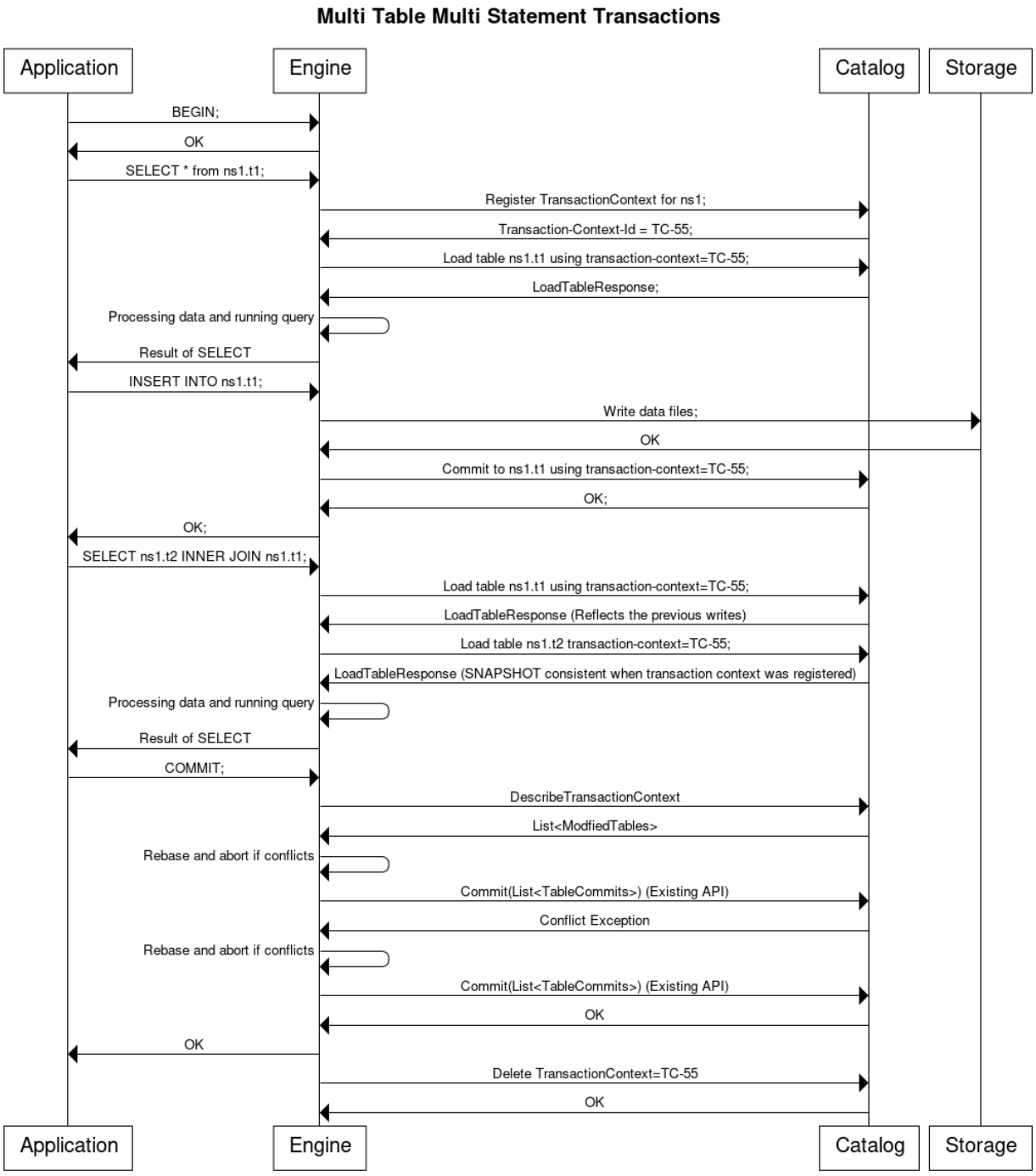
2. When writing changes to the table from a statement, the engine will provide the transaction context in the commit table request.

Using this logic, the catalog will be able to return a SNAPSHOT consistent view of namespace to the transaction, and allows engines to write intermediate changes to tables which are reflected in reads later - assuming the same transaction context is provided by engine to load those tables.

At commit time, the engine will use the described TransactionContext API to list all the changes made by the transaction, and rebase and commit to all the modified tables.

At rollback time, the engine will simply delete the transaction context from Catalog.

Sequence diagram illustrating how engine will use these APIs -



How does this provide a SNAPSHOT consistent view of tables across namespaces?
When a transaction first accesses a namespace, it registers a new TransactionContext. Catalog internally stores a reference to catalog state at that time. Further commits by other transactions to the tables are not visible when loading tables using this TransactionContext ID. Hence transactions have a SNAPSHOT isolation consistent view of tables during the lifetime of the

transaction.

How does this API help implement “Read your writes within transaction” ?
For table level commits of an on-going transaction, clients/engines actually store the new table state in memory in the proposed design.

Does this API allow both SERIALIZABLE and SNAPSHOT isolation levels?
Yes. Engines may choose to validate both Read Sets and Write Sets or just Write Set during rebase phase. Thus both SERIALIZABLE and SNAPSHOT ISOLATION can be supported if engines track that Read set in memory appropriately.

How long does the catalog maintain transaction contexts?

If the engine registers a new transaction-context, it should delete it either on ROLLBACK or COMMIT of transaction. In case the engine crashes and loses track of transaction-context, we propose to add a new Catalog property to allow catalogs to clean up orphaned transaction contexts -

history.expire.max-transaction-context-age-ms - Default max age of transaction contexts to keep track of.

Breaking Changes

This API spec adds three APIs to operate on TransactionContext, and adds optional fields to LoadTable and CommitTable APIs. These are designed to be backwards compatible in the sense that:

1. If a Catalog does not provide these APIs, the engine can fallback to existing APIs and keep current behaviour.
2. If Catalog provides new APIs and the engine does not use them, the result should be existing behaviour. Existing APIs should work as is.

Preferred Approach

While both the TransactionContext model and the CatalogSequenceNumber model provide viable paths to implementing multi-table transactional behavior, we recommend proceeding with **Option 1: the Catalog Sequence Number** approach, for the following reasons:

- The Catalog Sequence Number model offers a simpler interface, avoiding stateful coordination logic within the catalog.
- It aligns more closely with existing MVCC transactional systems, allowing transactional behavior to be managed on the client side in Iceberg.

This model fits well with Icebergs existing client-side validation's put in place today. Where the client is responsible for enforcing the isolation guarantees against single tables, and replaying operations if need be. The sequence number is just a minimal change to allow the client to know if the catalog supports the multi-table transactions.

Alternatives Considered

If an engine/client knows upfront all the tables involved in a transaction, then the issue of loading tables from a consistent snapshot can be solved by implementing a new LoadTables API, that takes as input a list of tables to load. But this prevents engines and protocols where the engine does not know about all the tables that can be accessed by transaction. One example is interactive transactions where an end user runs transactions in BEGIN……COMMIT format. Hence this document explores TransactionContext at the Catalog level.

Core API Changes to Mediate Transactions [EXAMPLE to help understand workflow]

SupportsCatalogTransactions

This is the main interface for a catalog that enables the Catalog Sequence Number transactions, which gives two essential capabilities. Load table a table snapshot at a specific sequence number, and a way to begin a catalog transaction.

```
public interface SupportsCatalogTransactions {

  /**
   * returns the current global catalog sequence number.
   */
 Long catalogSequenceNumber();

  /**
   * Load a table as of a specific sequence number.
   */
 Table loadTable(TableIdentifier identifier, long sequenceNumber);

 /**
  * Begin a transaction with a view of the catalog as of the specified
  * sequence number on first table load
  */
```

```
 CatalogTransaction beginTransaction(IsolationLevel isolationLevel);

 /*
 * Optionally we can surface the multi table commit API
 */
 void commitTransaction(List<TableCommit> commits);
}
```

CatalogTransaction

Taking inspiration from the doc by [External] Multi-Table Transactions in Iceberg. The `CatalogTransaction` provides a centralized client-side place to manage the **transactional state,** meaning the sequence number will be pinned here for the point-in-time view of the catalog and tracking table updates, and committing the multiple changes with the correct validations.

Rather than requiring each catalog to implement logic to stage table updates, handle validations, and commit through the multi-table commit api, we propose this consistent catalog interface, built around the catalog sequence number, with a common tracking, validation, and commit protocol.

Commit Validation and Isolation Guarantees

In order to ensure the respective isolation guarantees upon commit, the catalog transaction must perform validations.

- **Serializable Isolation:** By definition of Serializable Isolation engine will need to track the read set of the tables read during the transaction. Upon commit, the engine will validate the tables haven't changed against the latest in the catalog.
- **Snapshot Isolation:** For snapshot isolation, the client will attempt to commit the transaction optimistically. if a conflict is detected, the client may retry. This retry involves a rebase of refreshing the latest metadata for all involved tables at the latest sequence and replaying the TableUpdates, and aborting if there is a conflicting update on any table.

This replay can fail early if inconsistencies are detected. Iceberg's safeguards such as table UUID validation will prevent applying changes to a dropped and recreated table. However, it's important to note that Iceberg doesn't have a way to uniquely identify namespaces on the client side for validation, this limitation exists both inside and outside transactions.

Transactional State Tracking

For tracking updates and tables read in the catalog transaction, we leverage the BaseTransaction class. A BaseTransaction is created on first load for each table and can be used to:

- Collect and stage pending updates

- Satisfy our isolation guarantees by tracking read and write sets
- Supports a rebase operation by replaying staged pending updates against refreshed metadata when retrying.
- Cleans up any uncommitted files or state if a transaction fails or is abandoned
- Can be extended to read our writes for table scans in a catalog transaction

This mechanism ensures that all table interactions within a transaction share a consistent, point-in-time view and that commits meet our isolation guarantees.

Catalog Transaction Interface

Note: Similar to Eduard's approach, this method can expose a snapshot catalog designed to provide a consistent view of read requests by utilizing the sequence number from first read.

```
public interface CatalogTransaction {

  /**
   * The current {@link IsolationLevel} for this transaction.
   */
  IsolationLevel isolationLevel();

 /**
  * Return a point-in-time snapshot {@link Catalog} leveraging the
  * sequence number participating in the transaction.
  */
 Catalog asCatalog();

 /**
  * Commit all changes made in this transaction based on the following:
  * - Collecting all TableUpdates made across tables
  * - Performing appropriate validations based on isolation level
  * - Executing rebasing logic when permitted by the isolation level
  * - Ensuring atomic commit of all changes
  *
  * @throws CommitFailedException If the transaction failed
  * @throws SerializationConflictException If the transaction cannot be
  * retried under isolation
  */
 void commitTransaction();
}
```

Example Core API Usage

```
SupportsCatalogTransactions catalog = (SupportsCatalogTransactions) catalog;

// Begin a transaction
CatalogTransaction tx = catalog.beginTransaction();

// Load tables through the transaction to use consistent sequence number
Catalog txnCatalog = tx.asCatalog()

Table table1 = txnCatalog.loadTable(TableIdentifier.of("db", "table1")); ←start-ts
Table table2 = txnCatalog.loadTable(TableIdentifier.of("db", "table2"));

// Make changes to multiple tables within the transaction
table1.updateSchema()
   .addColumn("colb", Types.StringType.get())
       .commit();

table2.newAppend()
   .appendFile(dataFile1)
   .appendFile(dataFile2)
   .commit();

// Commit all changes atomically
// The commit will handle validation and retry logic based on isolation level
tx.commitTransaction();
```