[Public][External] Design Doc: Catalog-Managed Delta Table Feature

Authors: Scott Sandre , Ryan Johnson
Last modified: 2025-04-25

Part I: Design sketch

Motivation

Today's Coordinated Commit (CCv1) Delta Table Feature design is based on some major assumptions and product requirements that are no longer valid.

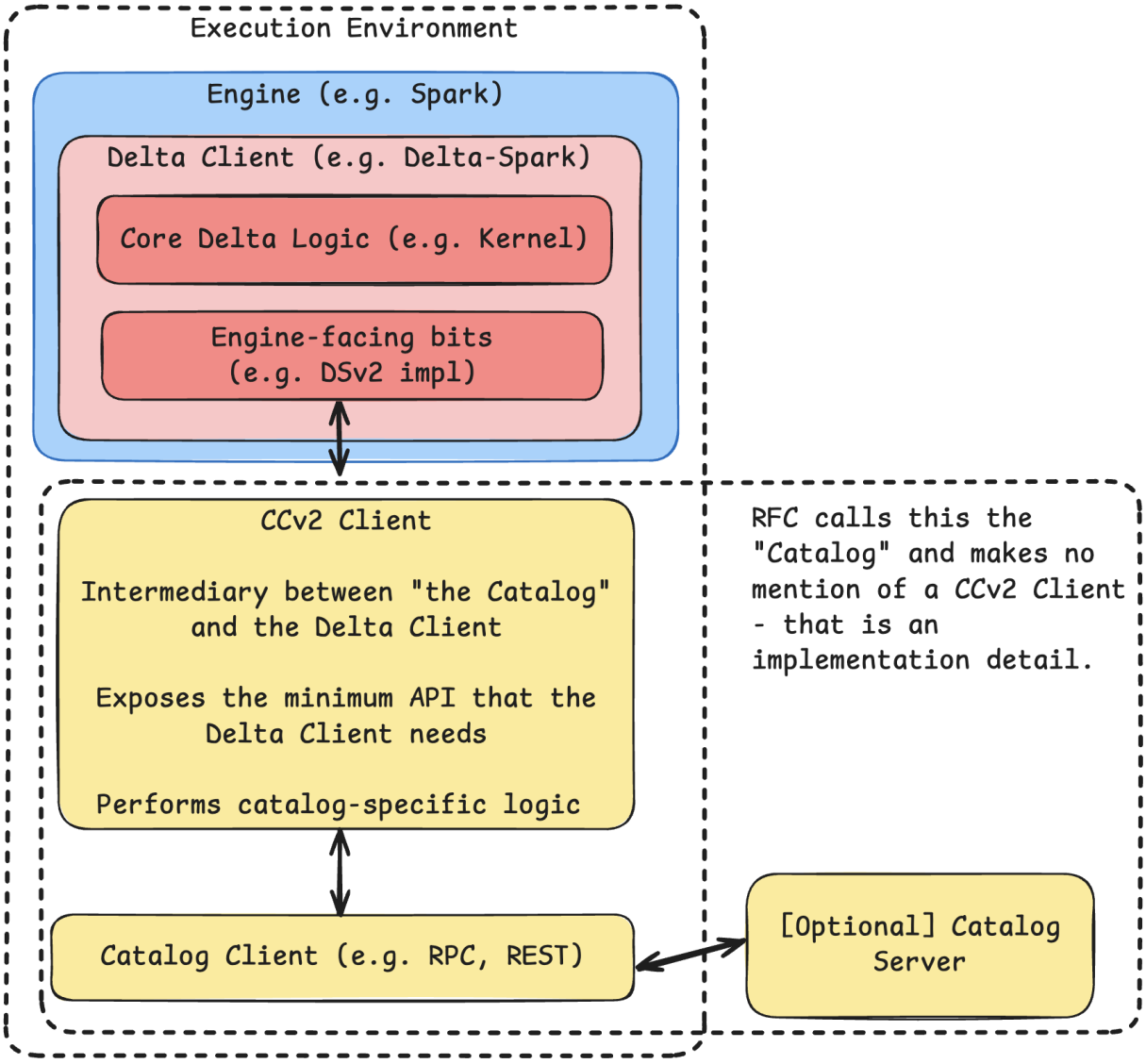
- CCv1 is difficult to integrate with catalogs because it (a) competed to "own" a given table and (b) had overlapping API requirements as demonstrated in building integrations with delta-spark, unity-catalog, delta-kernel, etc.
- CCv1 is designed for commit coordinators that were not necessarily catalogs, intending to cover commit service use cases like those using DynamoDB to coordinate writes to S3. Since then, S3 launched atomic PUT-if-absent so we no longer need to support that use case.
- Catalog-managed tables have become the norm, with most major vendors offering external access to catalog-managed tables. Doing that securely requires tight access controls, which are difficult to enforce with path-based access that bypasses the catalog.

CCv2 should also yield a performance benefit – as you'll learn below, we *can* trust the protocol/schema from the catalog and don't need to replay the delta log during query analysis.

Consequently, I propose that we pivot existing CCv1 protocol to become Catalog-based. This doc explores this new **Catalog-managed (CCv2) protocol**.

Terminology

Client / Engine / Catalog Terminology



- **[Query] Engine**: An engine like Spark or Trino or Polars that receives user queries and drives their execution. It relies on the Delta Client (see below) for Delta table format support.
- **Delta [Client]**: This is the code that implements support for reading and writing Delta tables, and which implements the CCv2 table feature. It likely consists of two key components:
    - **Core Delta Logic** (e.g. Kernel): Should never interface directly with the Catalog
    - **Engine-facing bits**: Communicates between the Engine, the Catalog

implementation, and the Core Delta Logic component

- **CCv2 Client**: The interaction layer between the Delta Client ⇔ Catalog. For example, takes the actions to be committed and executes catalog-specific logic to translate them to the catalog's preferred format (e.g. write out to a staged UUID file, or send as an inline commit)
    - For an example of a CCv2 client, see the Architecture Diagram below that shows the UC <> DSv2 <> Kernel integration. The CCv2 client is the `ResolvedMetadata` class
    - See two important sections of details [here] and [here]
- **Catalog Client**: The catalog-defined client-side component that provides APIs to to the CCv2 client and, optionally, communicates with a Catalog Server, likely via RPCs or REST APIs
- **Catalog Server:** The optional catalog backend service.
    - Optional because not all catalog implementations will have a catalog "server"
    - e.g. a DynamoDB-backed implement doesn't have a "catalog server"
    - e.g. a local HMS doesn't have a "server"
- **"Catalog implementation"**: CCv2 Client + Catalog Client + Catalog Server

Commit Terminology

CCv1 Terminology	CCv2 Terminology	Description
Staged commit	Proposed commit	A commit that a Delta client has proposed for version `N` of the table. It might not be the winner for that version. It could be "staged" and written to disk or stored "inline" in the catalog. It will either become "ratified" or "rejected"
	Sub-case: Staged commit	A proposed commit that is staged and written to disk at `_delta_log/_staged_commits/<version>.<uuid>.json`
	Sub-case: Inline commit	A proposed commit for which the catalog is the owner of the *content* rather than the *location*. i.e. its content was not written to the file system.
Unbackfilled commit	Ratified commit	A proposed commit that a catalog determined has "won" the commit at version `N`. It may or may not be published (i.e. backfilled). It may or may not even be tracked by the catalog at all – the catalog may have just atomically published it to the filesystem
Backfilled commit	Published commit	A ratified commit that has been backfilled into the

		`_delta_log/<version>.json`. Clients can directly discover published commits by listing, and the catalog may or may not still be tracking it. Note that publishing (aka backfilling) is an optional optimization that Catalogs may employ.
"Tracking"		Catalogs "track" ratified commits in that they "remember" which proposed commit won at that particular version. Once a ratified commit is published, the catalog no longer needs to "track" it. It can forget about it. It is now the filesystem's responsibility to inform Delta readers about it.
	Rejected commit	A proposed commit that did not win.

Requirements

Functional requirements

MUST:

- The CCv2 protocol must be 100% catalog-centric.
    - The catalog must determine whether a given commit attempt succeeded
- The CCv2 protocol must make very few catalog API assumptions. It must be compatible with:
    - a simple key-value-store catalog like DynamoDB
    - a simple filesystem-based catalog with PUT-if-absent
- The CCv2 protocol must allow Catalogs to be able to protect against rename race conditions
- The CCv2 protocol must allow Catalogs to (a) specify required table features upon table creation/upgrade and (b) set arbitrary catalog-specific table properties
- Each Delta table advertises whether it uses filesystem or CCv2 commits, and conforming clients refuse to perform filesystem-based commits when CCv2 is enabled.
- The CCv2 feature must support Creation of new tables, Upgrade/Downgrade (aka add/drop table feature) of existing tables, and Transfer to other catalogs
- Cloud storage does *not* need to be the source of truth for the content of commits. Catalogs may store the content of commits themselves.

SHOULD:

- The catalog should be the source of truth for schema and table-level metadata, *for at least the latest version of the table*. That is, it need not be the source of truth of table-level metadata for *historical* versions.
- Cloud storage should be the source of truth for the location of all but the most recent ratified commits, which are published to cloud storage "soon" after the commit finalizes .
    - This is primarily to reduce the number of commits the catalog has to track (millions in the worst case for a fast-moving table with 30 day history retention)
- The CCv2 protocol could allow Catalogs to detect discrepancies between the Catalog and the filesystem, i.e. detect invalid file-system commits

Non-functional requirements

- N/A – this doc is mainly about the CCv2 protocol, not low-level implementation details

Out-of-scope

- Exact Catalog APIs
- Single-commit change of ownership (from CatalogA to CatalogB). For now, users can just downgrade in one commit and upgrade in another

Proposal sketch

We don't want this proposal sketch to duplicate all the nitty gritty details of the RFC, found here. This proposal sketch gives an overview of some important areas.

CCv2 Overview

1. We introduce a new ReaderWriter (RW) Delta table feature named `catalogManaged` that defines the commit protocol that directly impacts Delta tables. This table feature requires (a) `inCommitTimestamp` to be enabled and (b) each commitInfo action to include a UUID.
    a. The consequence of a RW feature is that legacy clients, which do not understand this RW feature, cannot read nor write to the table.
2. The CCv2 protocol does not define the API and protocol for how the Delta Client and Catalog communicate – each Catalog implementation (CCv2 Client, Catalog REST/RPC Client, Catalog Server) decides this for itself.
3. Delta Clients will issue commits through the Catalog and will not attempt filesystem-based commits
    a. They will still write their parquet data to the filesystem directly. These are not commits. These files are not visible until committed through the catalog

(just like filesystem-based commits today).
    b. It is up to the Catalog to decide if it wants to detect *illegal* filesystem-based writes and what to do once it detects them.
4. Path-based access is not supported. Only catalog-aware + name-based access is supported.
    a. If you attempt path-based access, your warranty is void, and data integrity is not guaranteed
5. Delta tables do not need to store Catalog endpoint configuration or catalog-table metadata.
    a. It isn't the Delta Client's responsibility to tell the Engine which Catalog to contact. Instead, Engines hold this responsibility
6. The Catalog, not the filesystem, will be the source of truth about whether a given commit attempt succeeded or not
7. To briefly describe the read workflow:
    a. At the very least, there needs to be some GET COMMITS-type call that returns the catalog's ratified commits (location of staged commits, contents of inline commits). Delta-Spark, for example, should bundle this up with the GET TABLE call
        i. Some of these ratified commits may already be published – perhaps the Catalog just hasn't gotten the RPC yet to update its state
    b. The GET TABLE response should also include everything the Engine needs for query planning, such as the protocol + schema
    c. Optionally, the GET TABLE response may additionally include all necessary commit/checkpoint/minor-log-compaction file statuses so that the Delta Client does not need to LIST at all during the SCAN
    d. Else, to SCAN the table, Delta Clients must LIST the Delta Log and union the published commits with the ratified commits
        i. LogSegment = published commits + ratified staged commits + ratified inline commits
8. To briefly describe the write workflow:
    a. Delta Clients and Engines work together to write the parquet data
    b. Delta Clients send the actions to be committed to the CCv2 Client
    c. CCv2 Clients implement the catalog-specific logic, and propose commits by deciding to either send the location of a staged UUID commit file or just the content of an inline commit to the Catalog.
    d. Catalog ratifies the proposed commit, deciding who wins.
9. Catalog must track ratified commits until they are published (a.k.a. backfilled). Some ways to do this include
    a. Ratify by atomically publishing the commit to the file system directly

    b. Atomically recording that the location of the staged commit file corresponds to version `<V>`
    c. Atomically recording that the content of the inline commit corresponds to version `<V>`.
10. Publishing must be ordered
11. The table feature can be added or removed at any time

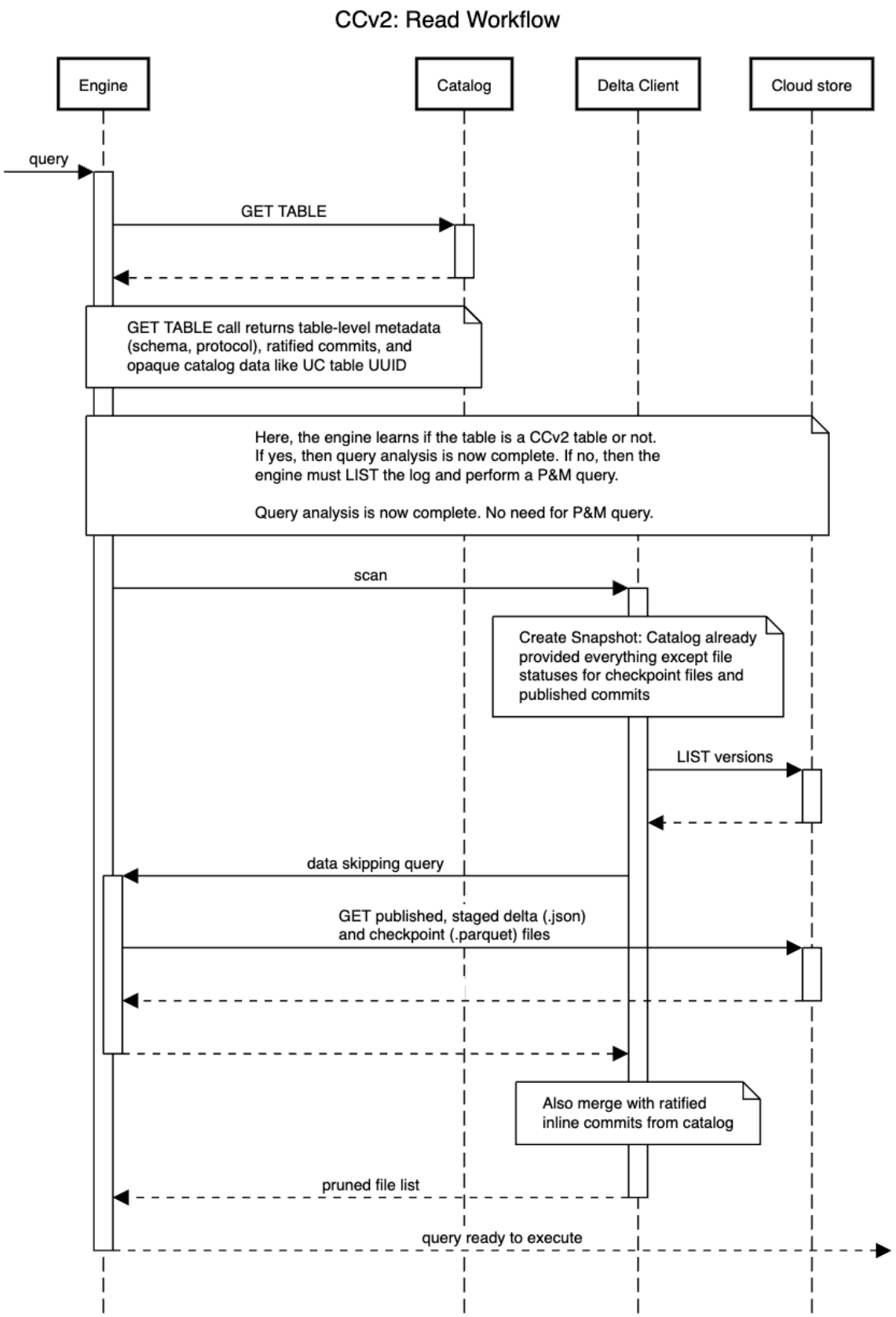
Engine query planning

Engines should make a GET TABLE type call to the Catalog as part of normal table resolution

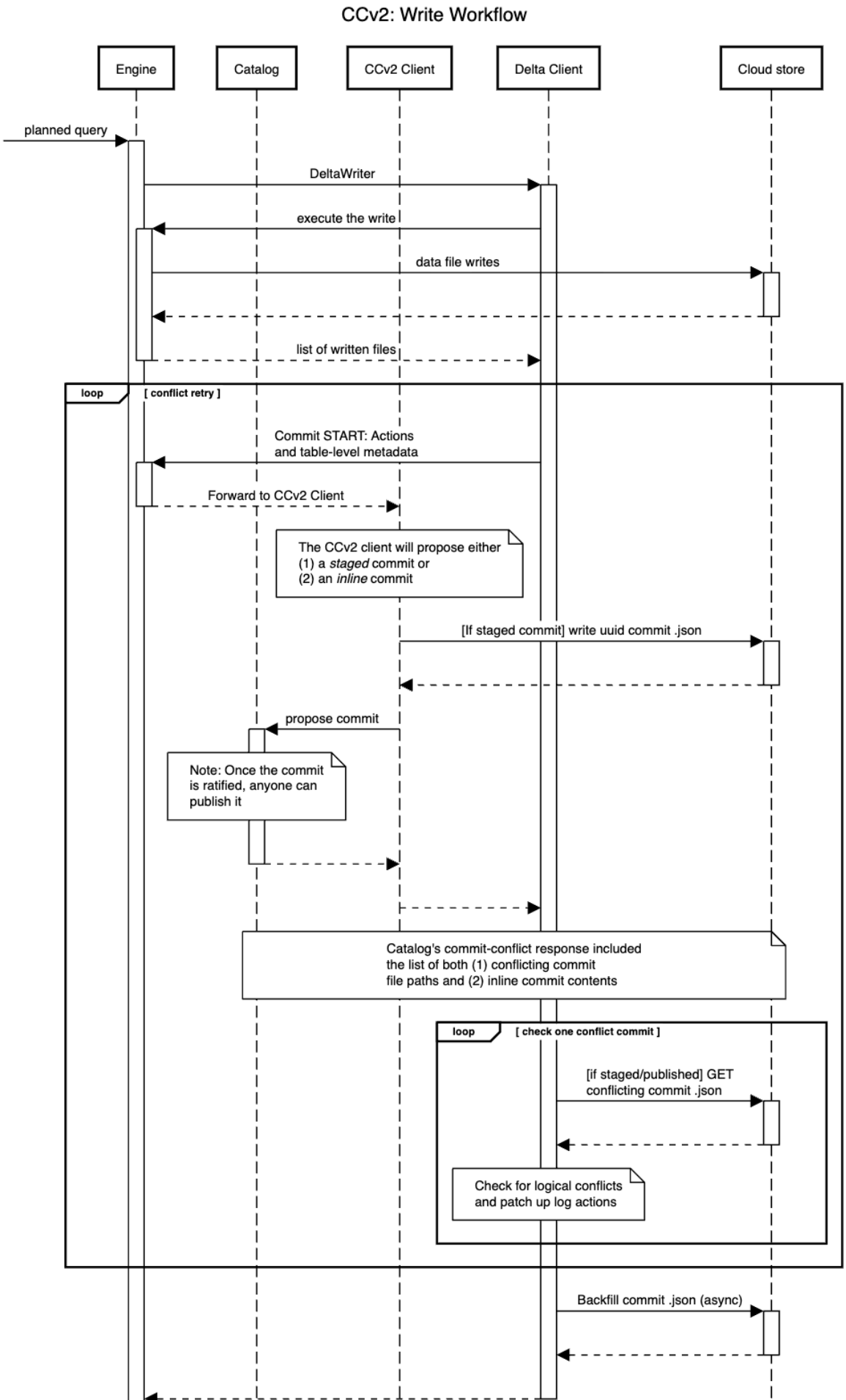
- This GET TABLE response includes the table's storage location and also gives the Engine all the information it needs for query planning. The Engine does not need to construct a Delta snapshot (i.e. perform log replay) for query planning
- Delta Clients do not invoke any Catalog APIs on the read path – the Engine gets all this information from the Catalog during table resolution and passes this on the the Delta Client
    - We do NOT need to pass around Catalog references/callbacks inside of our Core Delta Logic
- Note that time travel complicates this a little bit, since we only require Catalogs to return table-level metadata for the *latest* version

Part II: Detailed design

Read Workflow Diagram


CCv2: Read Workflow
Engine
Catalog
Delta Client
Cloud store
query
GET TABLE
GET TABLE call returns table-level metadata (schema, protocol), ratified commits, and opaque catalog data like UC table UUID
Here, the engine learns if the table is a CCv2 table or not. If yes, then query analysis is now complete. If no, then the engine must LIST the log and perform a P&M query.
Query analysis is now complete. No need for P&M query.
scan
Create Snapshot: Catalog already provided everything except file statuses for checkpoint files and published commits
LIST versions
data skipping query
GET published, staged delta (.json) and checkpoint (.parquet) files
Also merge with ratified inline commits from catalog
pruned file list
query ready to execute

Write Workflow Diagram


CCv2: Write Workflow
Engine
Catalog
CCv2 Client
Delta Client
Cloud store
planned query
DeltaWriter
execute the write
data file writes
list of written files
loop
[conflict retry]
Commit START: Actions
and table-level metadata
Forward to CCv2 Client
The CCv2 client will propose either
(1) a staged commit or
(2) an inline commit
[If staged commit] write uuid commit .json
propose commit
Note: Once the commit
is ratified, anyone can
publish it
Catalog's commit-conflict response included
the list of both (1) conflicting commit
file paths and (2) inline commit contents
loop
[check one conflict commit]
[if staged/published] GET
conflicting commit .json
Check for logical conflicts
and patch up log actions
Backfill commit .json (async)

What exactly is the CCv2 Client and how does it relate to the Catalog Client?

For the purpose of this design doc, the Catalog Client does one thing and one thing only: communicate with the Catalog Server (via, say, RPC or REST). It implements Catalog-specific APIs like LIST tables, POST to some endpoint, GET a table, etc.

The CCv2 client is the intermediary between the Delta Client and the Catalog Client. It implements client-side catalog logic, including:

- Receiving the set of actions the Delta Client wants to be committed and either (a) writing them to a staged UUID commit file or (b) sending them inline to the catalog server.
- Translating whatever the catalog client received from the catalog server into a format that the Delta Client can understand.
    - For example, it may take the list of FileStatuses of the ratified staged commits returned by the catalog and group them together into a LogSegment – if that's what the Delta Client wishes to receive.

Rename Protection

Let's first identify the problem. Suppose I have tables FOO (UUID_1) and BAR (UUID_2). Suppose my query to table FOO starts. Then FOO is renamed to BAZ, and BAR is renamed to FOO. Table FOO now has UUID_2, not UUID_1.

When my engine goes to write to table FOO, and if the Engine ⇔ Catalog communicate by `tableName` only, then there is no way for the engine nor catalog to know that this is not the original FOO I had intended to write to.

This is why we allow for our GET TABLE catalog call to return opaque catalog information, such as table UUID, that can be passed back to the catalog during the COMMIT call. This is what UC will do.

Detecting illegal filesystem commits during backfill

First, recall that `catalogManaged` depends on `inCommitTimestamp`, which requires that the commitInfo action is the first action in a commit. `catalogManaged` further requires that the commitInfo action has a `txnId` field which stores a UUID. Here is why:

Suppose that some Publisher (a Catalog or Delta client – we don't care which) publishes (via PUT-if-absent) a ratified commit into `N.json` and receives a

`FileAlreadyExistsException`.

Consider these cases

- Case A: `N.json` was already correctly backfilled, and the catalog either didn't receive that response or failed before it could update its internal state
- Case B: A filesystem writer illegally wrote `N.json`.

The Publisher can distinguish between them by issuing a HEAD request against `N.json`, parsing the commitInfo's `txnId`, and comparing that to the `txnId` of the ratified commit that was to be backfilled.

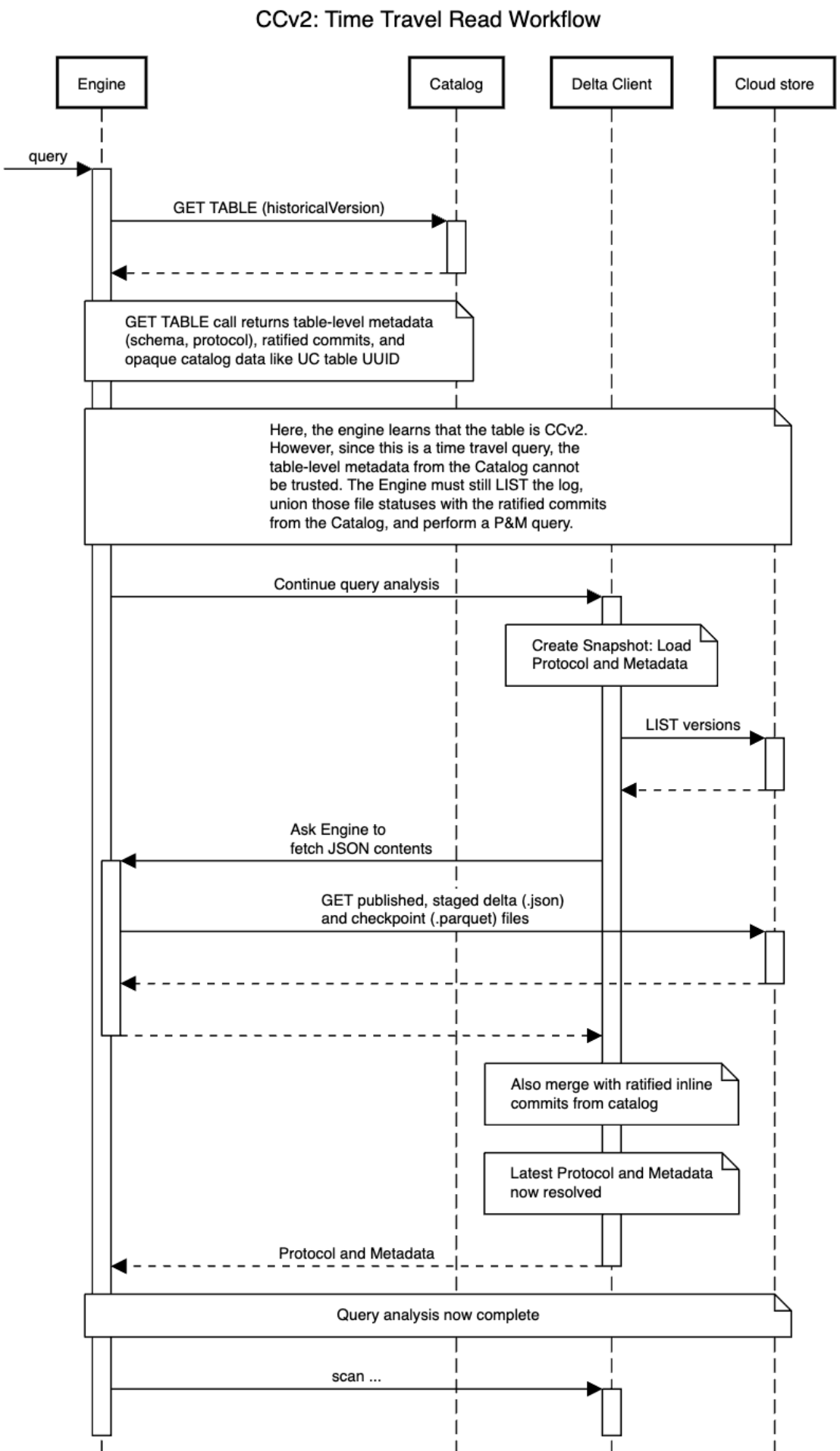
Time Travel

Our CCv2 Requirements say that Catalogs should be the source of truth of table-level metadata for the *latest* table version. For historical queries, the catalog has a few options, which the CCv2 RFC need not prescribe.

Note that we expect the Catalog to be contacted (GET_TABLE) for *every* query – latest or historical.

- **Option 1** : The Catalog provides table-level metadata (schema, protocol, etc.) only for the latest table version. It must return any ratified commits <= targetVersion
    - The Delta Client will have to do Log Replay using the file system + ratified commits
- **Option 2**: The Catalog provides table-level metadata for *any* historical version of the table by replaying the `_delta_log` server-side
- **Option 3**: The Catalog provides native support for historical table-level metadata

Time Travel Read Workflow Diagram


CCv2: Time Travel Read Workflow
Engine
Catalog
Delta Client
Cloud store
query
GET TABLE (historicalVersion)
GET TABLE call returns table-level metadata (schema, protocol), ratified commits, and opaque catalog data like UC table UUID
Here, the engine learns that the table is CCv2. However, since this is a time travel query, the table-level metadata from the Catalog cannot be trusted. The Engine must still LIST the log, union those file statuses with the ratified commits from the Catalog, and perform a P&M query.
Continue query analysis
Create Snapshot: Load Protocol and Metadata
LIST versions
Ask Engine to fetch JSON contents
GET published, staged delta (.json) and checkpoint (.parquet) files
Also merge with ratified inline commits from catalog
Latest Protocol and Metadata now resolved
Protocol and Metadata
Query analysis now complete
scan ...