Iceberg Materialized Views:

Lineage and State Representation

Introduction

In the last community sync on Materialized Views, community members agreed to split the information that is used to determine the materialized view staleness to two parts:

- **Lineage information:**
  - Tracked on the view side.
  - Stored as a map, referred to in this doc as *L*. The map contains an entry *(k, v)* for each view child (i.e., another view or table that the view depends on), where *k* is the child catalog identifier and *v* is an assigned ID to that child. See rest of the doc for the nature of the children (whether they are immediate children or transitive children).
- **State information:**
  - Tracked on the table side.
  - Stored as a map, referred to in this doc as *S*. The map contains an entry *(v, s)* for each deeply nested child of the view, where *v* is the ID of the child (drawn from the same pool of IDs used in the lineage map *L*) and *s* is the state information of *v*. If *v* refers to a table then state information is the refresh time snapshot ID, and if *v* refers to a view then state information is the (child's) view version ID at refresh time.

Purpose of Lineage Information

The lineage information allows engines that use a different dialect than the view's dialect to leverage the storage table without needing to parse the view text. By having child information readily available, these engines can avoid failures caused by parsing incompatible dialects.

Type of IDs

One open design question is about the type of the IDs used in both the lineage and state information. Such IDs can be sequence numbers (whose scope is within each view; they restart from 1 for each view), or the respective table/view UUID.

Below are the methods for setting IDs in both cases.

Sequence Numbers

- The list of children tracked by the lineage map *L* should encompass all the view's transitive children. Since sequence IDs scope is local to each view (i.e., they start form 1 for each view), the lineage map *L* cannot be spread across multiple child nested views.
- **At view creation or refresh time**, the entire view tree must be parsed by the engine to come up with the deep children sequence IDs. Such sequence IDs of deeply nested children are used to create the lineage map *L* on the view. Same sequence IDs are used to set the state information *S* on the storage table.
- **At view evaluation time**, IDs on both the lineage map *L* and the state map *S* are directly used to correlate the catalog identifier (of a child) with state information.

UUIDs

- The list of children on each view can be limited to the immediate children only since UUIDs are globally unique.
- **At view creation time**, for both regular (non-materialized) and materialized views, only the current view must be parsed. UUIDs of immediate children are used in the lineage map *L* of the respective view.
- **At view refresh time**, deep lineage is obtained by traversing the view tree. Deep lineage will still be a map *L_deep* from a catalog identifier to a UUID. UUIDs from the deep lineage map *L_deep* are used to set the state *S* on the storage table.
- **At view evaluation time**, similarly, deep lineage *L_deep* is calculated from the nested views, then UUIDs from *L_deep* and *S* are used to correlate the catalog identifiers with state information.

Advantages of using UUIDs

- UUIDs avoid having engines parse incompatible views (which is the main purpose of introducing lineage information in the first place). In the case of sequence IDs, an engine is expected to deeply parse the nested views at creation or refresh time, even if their dialect is not compatible with the main view, which is quite possible. In the case of UUIDs, each engine is responsible for setting lineage information of its own created views. Hence, cross dialect parsing is not required.
- UUIDs are more resilient to missing lineage information (in case one engine does not provide them) since they can always be deterministically reconstructed by parsing (as a fallback). IDs from this reconstruction will always be consistent with info stored in the state map. On the other hand, the state map in the case of sequence IDs is meaningless if the lineage is missing, and parsing is always required.
- Using sequence IDs requires the refresh module to reproduce the same IDs used by the create view module. Depending on this constraint is fragile. The other option is to delegate creating the sequence IDs exclusively to the refresh module, but that will

require the refresh module to modify the view information, which is an unnecessary coupling and can lead to less modular code.

- When using sequence IDs, the lineage information on a view becomes invalid once a child view changes which tables/views it references. That does not happen when using UUIDs.
- UUIDs are already an established concept in Iceberg. This enables lineage information to be useful in other applications other than materialized view staleness, such as general lineage calculation/tracking.
- Lineage using UUIDs is applied equally on materialized and non-materialized views allowing for a simpler view spec, and frictionless conversion between materialized and non-materialized views.