[OSS] Secure Views for dynamic policy enforcement

Authors: Prashant Singh Russell Spitzer Vishwa Lakkundi
Last updated: Jun 24, 2025

Reviewers & Contributors: laurent@dremio.com jb.onofre@dremio.com
royhasson@microsoft.com kevinliu5@microsoft.com

Prev OSS work : Iceberg Spec Extensions for Data Access Decision Exchange

Motivation

For data warehouse and lakehouse systems, robust access control is vital to maintaining security, privacy, and compliance. In multi-cloud, multi-engine environments, it serves as a key component of comprehensive data governance.

It is already possible to store access management policies in several catalogs and systems such as Glue and Ranger and implement enforcement of such policies in the several engines like Spark, Trino and others. **The OSS proposal here discusses an approach to extend Iceberg spec to enable exchanging Access Decision' rather than merely the access control policies**.

The main reason we use the Iceberg view is that different catalogs have different types of transformation policies, and developing a standardized spec that all parties can all agree upon will be very challenging and also limiting in functionalities.

In this view-based approach, catalogs can always dynamically construct a view object that can be used by an engine which fully describes all the access decisions. The goal is that **as long as engines naturally integrate with Iceberg view features that we propose below, they will be able to enforce access decisions from transformation policies without the need to make additional changes**.

Typical alternatives demand extensive, intrusive engine modifications for both **policy store** integration and policy evaluation. In contrast, view redirection streamlines this by centralizing policy management within the catalog, eliminating the need for such deep engine changes.

Some Companies such as Linkedin (ViewShift) and AWS, already use such redirection and views of enforcing access decisions in prod to achieve the same.

Goals

1. Enable a *uniform way* to enforce row and column governance policies across engines.
2. Enable these policy enforcement *with least amount of changes* to engines such (as Trino, Starrocks, Spark Connect)
3. Support the following policy spec:
   [OSS] Row and Column Based Access Control: Policy Definitions
4. Support row and column based access control for reads only, all writers need to have a privilege as TABLE_WRITE_DATA of Apache Polaris, which essentially means Enabling writing data to the table by receiving short-lived read+write storage credentials from the catalog (details).

Non Goals:

1. Establish trust between Polaris and Engines. We will have to solve this problem, separately as there needs to be a trust relationship between catalog and engine. We have some early thoughts on it as if it is like mTLS or by using secure token exchange.
2. Adding reading Iceberg view read support to Engine
3. Secure view execution requirements such as not allowing reordering of predicates within the engine leading to security vulnerabilities, hiding the plan of execution from the user etc are not part.
4. Supporting UDF's there is no standard way to create and store UDF in iceberg yet,Iceberg Community is working for SQL based UDF, one should wait for it and then the story on how it integrates with secure worlds such as column masks. This doc / proposal only supports what iceberg view spec supports as of today.
5. Non SQL clients

Background:

Engines way of identifying if it's view or table

Engines such as Trino during the analysis phase tries to understand if the identifier refers to a table or a view by issuing the catalog IRC loadView / loadTable (code) .

Engines way of storing security properties for a view:

Trino uses iceberg view metadata (spec) is stored in view properties (code)

Implementation Details : Passing secure view back during analysis phase:

POC: Scratch Pad for Trino POC

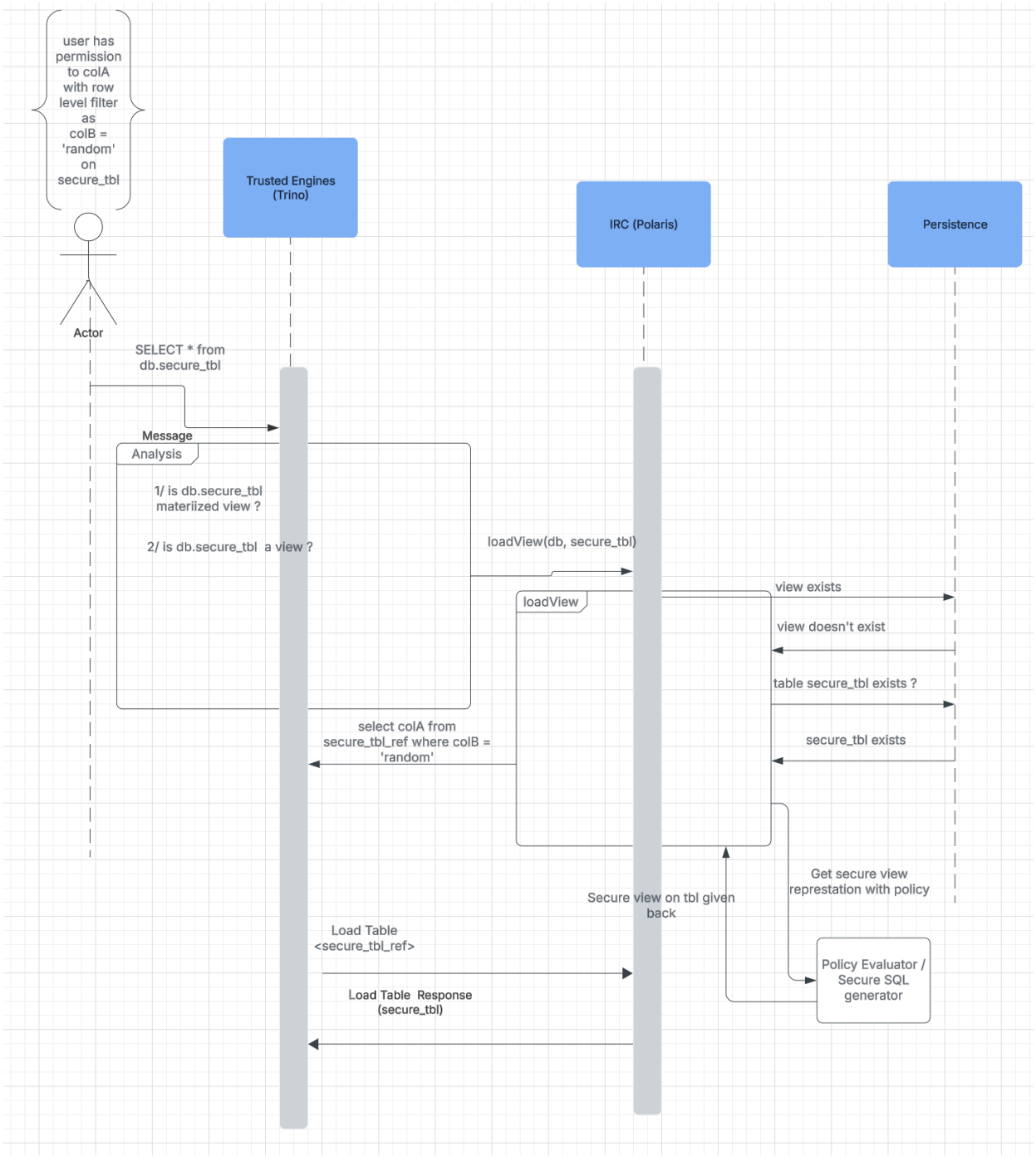
Overview:

The main idea is to use views as a decision of access control and enforce FGAC (Fine Grained Access Control) without making any changes to the engine. The following flow works seamlessly without any code changes in Trino and deals with the complexity of breaking recursive calls without any change to spec or engines.

A user can define an access control policy (Column level Security or RLS Row level Security), when during analysis phase, the engine tries to infer if its view or table and issues a loadTable and the table had Column and Row level policy on identifier it should get a 404 so the analyzer calls loadView on identifier. Polaris returns back SecureView (a view translating row level and column level access control to a SQL which engines understand). As per iceberg spec, the dialect is part of the view definition so a secure view will have dialects of engines which Polaris thinks can execute secure views.
Secure view generation, both metadata for secure view and SQL dialects supported is Polaris responsibility, as if Polaris acts as a Policy Evaluator and conveys the result back to the Engine as part of Secure View.

Requirements:

- Column level Security / Row level Security policy contains table column names and row filters or may be the column fieldIds or iceberg projections. (to ensure schema evolution can't exploit for security attacks)
- Engines should understand / support reading iceberg views
- Engines execute the view in a secure way as if it shouldn't allow things like predicate re:ordering which could lead to attacks like this [ref link].
- Polaris Policy Evaluator module supports generating the SQL in the target dialects (ANSI mostly) and also supports generating view metadata per engine to convey back to the engine that it's a secure view.
- Things like credential vending for the table remains intact as the underlying APIs such as loadTable and loadCredentials of IRC will be called in the normal flow.



Steps:

- Engines try to understand if the identifier is a view or a table.
- Engine issues a loadTable with identifier, Polaris checks if the table had RLS / CLS attached, if yes gives back 404
- Engine Analyzer forced to issue IRC loadView call for trying to find if the identifier is view

- Polaris determines that no such view exists but there exists a table of the same name which has policies defined.
- Existing IRC AuthN should be able to figure out what permission a user has against the table and what are the policies associated with it .
- Polaris issues a call to Policy engine Evaluator to evaluate the RLS and CLS policies associated with the table and get back the Iceberg View in all the engine dialects Polaris thinks the engine is trustable.
  Note: Tables referenced in the Secure view definitions are one shot table names that Polaris tracks let call it ***secure_tbl_ref***
- Engines post seeing the view definition sql should treat this Relation as View and proceed to its resolution.
- Engine tries to make a call to Polaris again to resolve the table which view refers to (***secure_tbl_ref***), Polaris does a lookup to find out the actual table name and returns its response as part of loadTable. This is required to break the recursion, as if we give back the engine the view containing the actual table name, it will again get a back view.
- View gets resolved and executed securely within the **trusted** engine.

FAQs: Secure Views for Dynamic Policy Enforcement

1. **What if an engine doesn't want to use views for Fine-Grained Access Control (FGAC)?**
   Engines can directly request the policy from the catalog. This policy can be provided during ***loadTable*** or through a separate ***loadPolicy*** API, similar to Polaris's approach. (Refer to: Policy Store Implementation in Apache Polaris), we can have a defined header, which will send back a token issued by the catalog, which when trusted engines call the table with it can indicate it wants view redirection or not.
   Essentially there are 4 cases
   - Trusted Engine which are smart and can enforce policy themselves (loadTable would pass, policy can be conveyed via loadTable or loadPolicy API)
   - Trusted Engine which don’t understand policy (loadTable would give 404, loadView would give a 200 with the policy evaluated view)
   - Non trusted Engine which can understand policy (loadTable and loadView all 404)
   - Non trusted Engine which can’t understand policy (loadTable and loadView all 404)
2. **Who generates these secure views?**
   Polaris generates the secure views. This generation can occur at runtime or through an offline process. The key is to ensure that when a table is requested, a view reflecting the most current policy is returned.
3. **What is the impact on listTables and listViews who qualify for re-direction?**
   - listViews : No effect, as there isn’t technically a view of that name

- listTables : some engines does an optimizations to do a listTable before resolving a view to avoid repeated calls to interpret if its a table or a view, i think it am fine with either listTable to show the table name, note in Polaris list tables it is an entirely different Privilege TABLE_LIST here, as for cases like federation we do want to know list of tables, but other perspective is will federation be a user with just READ privilege or a SUPER ?

4. **How does it affect caching ?**

   Table Metadata: For trusted engines, if they accept the contract that table references will always be hidden from the user, we will consistently return the same table reference. This allows engines to cache metadata names against identical table reference names.

   Result Set Caching: Result set caching is typically performed per query fragment, which yields deterministic results. We capture the key as the plan node signature and utilize the ETAG of the files. This approach should ideally not negatively impact cache hits, given that the entire query subtree and leaf nodes (like tables) are largely consistent.

5. **What features does this proposal support?**
   - **Column Hiding:** Controls which columns a user can see.
   - **Column Masking:** Applies a mask to a column before projection.
   - **Row Filtering:** Filters rows to determine visibility.

   The proposal outlines a phased implementation:

   - **Phase 1:** Policies with column projections and basic row filters using Iceberg expressions. (Refer [OSS] Row and Column Based Access Control: Policy Definitions)
   - **Phase 2:** Policies with column projections and basic row filters using Iceberg expressions that can reference context functions / variables. (Refer to: [OSS] Row and Column Based Access Control: Policy Definitions)
   - **Phase 3:** Policies incorporating column masks and row masks, contingent on Iceberg UDFs being available. (Refer to: [OSS] Row and Column Based Access Control: Policy Definitions)

6. **Why use Iceberg Expressions as row filters?**

   Iceberg expressions are portable. While their operator support is limited, most engines already have converters from engine-specific SQL to engine expressions and then to Iceberg expressions. Examples include Trino (here) and Spark (here).

7. **Is the view SQL text required to be dialect-specific?**

   No, the goal is for the view SQL to be ANSI compliant, allowing the same text to work with Spark and Trino. This involves creating a template like: ***Select <column projection list> from tbl where <row_filters>***. This essentially creates a Project and Filter node on top of the table relation, which engines would need to do regardless, even if the catalog returned the policy.

8. **How are dialect-specific SQL policies handled?**

The community is currently working on UDF support, (for ex [OSS] Row and Column Based Access Control: Policy Definitions) which when there, is expected to manage the complexities of dialect-specific functions. For instance, a UDF named **hash** might map to **sha2** in Spark dialect and **sha256** in Trino, or even better if some IR comes up fine too.

9. **What is the impact on iceberg functions like history/time travel ?**
   Short answer, it will not work, though things like icebergs snapshot table (here) exposes more metadata than intended for example snapshot summary, which defeats the purpose for FGAC, also time travel with history of column names might require a lot of thinking but might not work at all without rest spec change to load table to give it in the call that what snapshot we want to time travel to ?