Iceberg Materialized View Concept Discussion

Background and Motivation

A materialized view precomputes results of a query to be used as a logical table. When queried the materialized view serves the precomputed results reducing the query latency. The cost of query execution is pushed to the precomputation step and is amortized over the query executions.

The big open-source query engines Trino and Spark have either recently added (link) or are in the process of adding materialized views. Currently the materialized views are implemented as an iceberg view with an underlying storage table. The metadata required for view maintenance is stored as a property of the underlying storage table.

The iceberg table format is becoming an important building block in modern data lakes and lakehouses. In addition to open-source query-engines, support from commercial cloud data warehouses like Snowflake, Bigquery and Dremio is available or underway. Iceberg therefore plays a crucial role in enabling data federation between different data lakes and warehouses.

Current limitations

1. No formal specification

Currently materialized views are lacking an open, accessible definition of the format. This makes it difficult to implement iceberg materialized views for new query-engines and consequently hinders adoption.

2. No process for evolution

Without a formal specification it is difficult to manage the evolution of the format across different query-engines. There is no central place where requests can be brought forward. A specification can help with maintaining backward compatibility.

Goal

A common metadata format for materialized views enabling materialized views to be created, read and updated by different query engines.

Requirements for Materialized Views

Fundamentally the main functionalities required for materialized views are:

1. View definition (representation of relational algebra)
2. Storage of precomputed data
3. Lineage information (freshness)

It shows that MVs therefore have similarities with common views (1) and common tables (2). It follows that MVs can be realized by different designs, depending on how functionalities 1 and 2 are combined.

Design for Materialized Views

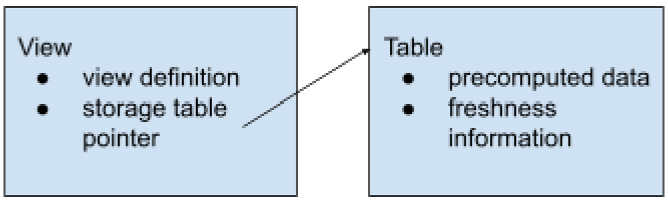
Comparison of different design options

	Design 1: Common view + attached storage table	Design 2: Table + attached common view	Design 3: Combined metadata for table and view
Description	Common view stores view definition and pointer to storage table. Storage table stores precomputed data and lineage information.	Table stores precomputed data, lineage information, and pointer to associated common view. Common view stores view definition.	New materialized view metadata format that stores view definition, precomputed data and lineage information
Pros	• Fallback to View, if MV is not supported: If iceberg materialized views are not supported by query engine or BI tool, the view definition can still be executed • Can be realized with existing iceberg tables and common views • Similar to Trino's design	• precomputed data can still be returned if MV is not supported by query engine or BI tool • Can be realized with existing iceberg tables and common views	• Can achieve high write performance without requiring that a second storage table has to be registered in the catalog

Cons	• If the storage table metadata wants to be stored in the REST catalog (better write performance), the view and the storage table have to be registered in the catalog.	• Different from Trino's design	• New metadata format has to be defined • Different from Trino's design

Current design

Description



Common view stores view definition and pointer to storage table. Storage table stores precomputed data and update information.

Pros

- Fallback to View, if MV is not supported: If iceberg materialized views are not supported by query engine or BI tool, the view definition can still be executed
- View can be linked to multiple storage tables, which query engine can choose from depending on the table partitionings
- Can be realized with existing iceberg tables and common views

Cons

- duplicate schema in view and table

Lineage information

To check if the materialized view is still up-to-date additional metadata has to be stored in the materialized view. Most importantly this includes the snapshots-id's of the tables referenced in the view definition. There are different ways to store this metadata.

Comparison of ways to store freshness information

	summary field of storage table snapshot	properties of storage table
Description	Freshness information is stored as additional fields in the summary field of a storage table snapshot.	Freshness information is stored as additional entries in the properties field of the storage table.
Pros	● tracks the history of the freshness information over all snapshots ● enables “timetravel”	● simple
Cons		● only current freshness information is stored

0. Question: How should the Materialized View Metadata be stored?

Decision for Option 1

Materialized views are composed of a query definition, precomputed data and lineage information. They can be realized by different designs depending how these different parts are composed. The following table shows the pros and cons of different MV designs.

Votes:

	Design 1: Common view + storage table registered in the catalog	Design 2: Common view + storage table stored as metadata.json file	Design 3: Common view + storage table stored as nested field of view	Design 4: New metadata object with combined metadata for table and view	Design 5: Common view + storage table registered as system table
Description	Common view stores view definition and pointer to storage table. Storage table stores precomputed data and lineage information. The storage table is registered in the catalog.	Common view stores view definition and pointer to storage table. Storage table stores precomputed data and lineage information. The storage table is stored in a metadata.json file and not registered in the catalog.	Common view stores view definition and pointer to storage table. Storage table stores precomputed data and lineage information. The entire metadata of the storage table is stored as a nested field inside of the view.	New materialized view metadata format that stores view definition, precomputed data and lineage information.	Common view stores view definition and pointer to storage table. Storage table stores precomputed data and lineage information. The storage table is only accessible as a system table of the view.
Pros	- Can be realized by extending iceberg tables and common views - Not required to use JSON	- Can be realized by extending iceberg tables and common views - One entry in catalog	- Can be realized by extending iceberg tables and common views - One entry in catalog	- Not required to use JSON files (better write performance) - One entry in catalog	- Can be realized by extending iceberg tables and common views

	files (better write performance)	- Similar to Trino’s design	- Not required to use JSON files (better write performance)	- Evolve the materialized view without evolving the view spec.	- one entry in catalog for view users
Cons	- Two entities have to be stored in the catalog (users could modify storage table) - Catalog has to be queried twice to get metadata - Different from Trino’s design	- Use of JSON files	- New REST Operations need to be defined - highly nested structure - Different from Trino’s design	- New metadata + REST format has to be defined - Different from Trino’s design	- Two entities have to be stored in the catalog - Catalog has to be queried twice to get metadata - Different from Trino’s design
Opinions				Jan: Just looking at the technical arguments, I prefer this approach (difference from trino + new format are not technical reasons)	Jack: added this option based on devlist discussions. There is an open question of this design of how to access the storage table in REST. If it is /namespaces/ns/tables/view.storage, then how do aspects like permission, name conflict, work.

Summary of discussion in devlist:
https://lists.apache.org/thread/tb3wcs7czjvjbq9y1qtr87g9s95ky5zh

Jack Ye (proposed design 4, but also acknowledged this will be a lot more work compared to other designs): I think we (at least me) started with this assumption of MV = view + storage table, mostly because this is how Trino implements MV, and how Hive tables store MV information today. But does it mean we should design it that way in Iceberg? Now I look back at how we did the view spec design, we could also say that we just add a representation field in the table spec to store view, and an Iceberg view is just a table with no data but with representations defined. But we did not do that. So it feels quite inconsistent to say we want to just add a few fields in the table and view spec to call it an Iceberg MV. If we have a new and independent Iceberg MV spec, then an Iceberg MV is under-the-hood a single object containing all MV information. It has its own name, snapshots, view representation, etc. I don't believe we will be blocked by Trino due to its MV SPIs currently requiring the existence of a storage table, as it will just be a different implementation from the existing one in Trino-Iceberg. In this direction, I don't think we need to have any further debate about pointers, metadata locations, storage table, etc. because everything will be new. But on the other side it is definitely associated with more work to maintain a new spec, and potentially big refactoring in the codebase to make sure operations today that work on table or view can now support MV as a different object. And it definitely has other problems that I have overlooked.

Daniel Weeks (support design 1, proposed alternative design 5): I think we should consider either allowing the storage table to be fully exposed/addressable via the catalog or allow access via namespacing like with metadata tables. E.g. <catalog>.<database>.<table>.<storage>, which would allow for full access to the underlying table. In many ways the materialized view is an extension/optimization of a view. not only do I think it makes sense to expose the storage, I think it is necessary and provides a lot of capability. You want to be able to have multiple engines potentially participate and have access to the underlying storage because they may have different abilities to refresh or consume that data. Additionally, it's important to have a way to audit and inspect the storage table and how it changes over time. I think it makes sense to have a single spec for both view and materialized view as there is a significant overlap in the definition and behaviors. In fact, a materialized view is a superset of view (depending on defined behaviors). I think it overcomplicates things to separate the two.

Micah Kornfield (seems acceptable to design 4, and see the value for reusing table and view components in other designs): I think we want this to the extent that we do not want to redefine the same concept with different representations/naming to the greatest degree possible. This is why borrowing the concepts from the view (e.g. multiple ways of expressing the same view logic in different dialects) and aspects of the materialized data (e.g. partitioning, ordering) feels most natural. I think you are saying maybe two modifications to the existing proposals in the document:

1. No separate storage table link, instead embed most of the metadata of the materialized table into the MV document (the exception seems to be snapshot history)
2. For snapshot history, have one unified history specific to the MV.

Walaa Eldin Moustafa (in favor of a view + storage table design, leaning towards design 5): For the end user, interfacing with the engine APIs (e.g., through SQL), materialized view APIs should be almost the same as regular view APIs (except for operations specific to materialized views like REFRESH command etc). Typically, the end user interacts with the (materialized) view object as a view, and the engine performs the abstraction over the storage table. For the engines interfacing with Iceberg, it sounds the correct abstraction at this layer is indeed view + storage table, and engines could have access to both objects to optimize queries. So in a sense, the engine will ultimately hide most of the storage detail from the end user (except for advanced users who want to explicitly access the storage table with a modifier like "db.view.storageTable" -- and they can only read it), while Iceberg will expose the storage details to the engine catalog to use it in scans if needed. So the storage table is hidden or exposed based on the context/the actual users. From Iceberg point of view (which interacts with the engines), the storage table is exposed. Note that this does not necessarily mean that the storage table is registered in the catalog with its own independent name (e.g., where we can drop the view but keep the storage table and access it from the catalog). Addressing the storage table using a virtual namespace like "db.view.storageTable" sounds like a good middle ground.

Manish Malhotra (support design 1): It is good to keep things simple, though not 100% sure, if the storage table should be registered to the metastore.

Amogh Jahagirdar (**support design 5, find approach 1 also acceptable**):

I think it's advantageous to leverage existing concepts we have to achieve materialized views.

We know that foundationally a materialized view is composed of 2 broad parts:
1. A representation for computation (e.g. SQL)
2. A materialization of the results of the computation (a table)

We also know at this point Iceberg Views can serve as the representation, and the Iceberg Table naturally serves as the materialization of the results. Leveraging the view metadata is advantageous for more obvious reasons. We need some shared metadata to store a SQL/IR computation, Iceberg Views facilitate that and I'm not convinced yet we need to really reinvent anything here for the materialized view case, beyond defining some additional properties. Leveraging the existing table primitives is where things are more interesting and surface aspects we'll get "for free" with this design. We'll probably want to keep track of history for materialized view changes, which tables already provide. Materialized views can also be partitioned, which tables already provide. Materialized views can reference indices and stats via Puffin, which will be very helpful when trying to perform more efficient incremental maintenance. The same maintenance procedures which run on tables can also be run on the underlying MV table. Now there are some aspects and more open questions we'll need to consider with this approach but I think solving these problems is more tractable than designing a new spec and having to implement that.
For example, this approach includes ideally ensuring only one database object (the view) is actually visible either through a notion of a hidden or system table. I also think it’s important that the view doesn’t point to the materialized storage physical location and is rather a logical reference to the table to simplify the update path for refreshes. Lastly, since a view can have multiple dialects what does this mean for a materialized view with multiple dialects since semantics are different? Do we store both materialized results as separate storage tables or do we store a single table with multiple branches? I've glossed over API design and detail but I've already written quite a bit, and if we conclude that the tradeoffs for composing existing primitives are better than the tradeoffs for a new spec, we can talk more about this in detail

Design 1

Overview

MVs (Materialized views) are realized as a combination of an iceberg common view with an underlying storage table. The definition of the materialized view is stored in the representation field of the common view. The precomputed data is stored in an iceberg table called storage table. The information required for refresh operations is stored as a property in the storage table. All changes to either the view or the storage table state create a new view metadata file and completely replace the old view metadata file using an atomic swap. Like Iceberg tables and views, this atomic swap is delegated to the metastore that tracks tables and views by name.

Metadata Location

An atomic swap of one view metadata file for another provides the basis for making atomic changes. Readers use the version of the view that was current when they loaded the view metadata and are not affected by changes until they refresh and pick up a new metadata location.

Writers distinguish between changing the view or the storage table state.

Writers changing the view state create view metadata files optimistically, assuming that the current metadata location will not be changed before the writer’s commit. Once a writer has created an update, it commits by swapping the view's metadata file pointer from the base location to the new location.

Writers changing the storage table state create table metadata files optimistically, assuming that the storage table pointer in the view will not be changed before the writer’s commit. The commit is performed in two steps. First, the Writer creates a new view metadata file optimistically and changes the storage table pointer to the new location. Second, the new view metadata file gets committed by swapping the view's metadata file pointer in the metastore from the base location to its new location. The commit is only successful when the second step succeeds.

Specification (DRAFT!)

The metadata of the materialized view consists of four parts. The view and the storage table metadata constitute one part each. Since not all information can be stored inside the view and storage table metadata, two additional parts are introduced in the `properties` field of the view and storage table metadata respectively.

Materialized view metadata stored in the common view properties

One part of the materialized view metadata is stored inside the `properties` field of the common view. The metadata is stored in JSON format under the key "materialized_view_metadata". The materialized view metadata stored in the view has the following schema.

v1	Field Name	Description
required	**`format-version`**	An integer version number for the materialized view format. Currently, this must be 1. Implementations must throw an exception if the materialized view's version is higher than the supported version.
required	**`storage-table-location`**	Path to the metadata file of the storage table.
optional	**`allow-stale-data`**	Boolean that defines the query engine behavior in case the base tables indicate the precomputed data isn't fresh. If set to FALSE, a refresh operation has to be performed before the query results are returned. If set to TRUE the data in the storage table gets returned without performing a refresh operation. If field is not set, defaults to FALSE.

Materialized view metadata stored in the storage table properties

Another part of the materialized view metadata is stored inside the `properties` field of the storage table. The metadata is stored in JSON format under the key "materialized_view_metadata". The materialized view metadata stored in the storage table has the following schema.

v1	Field Name	Description
required	**`format-version`**	An integer version number for the materialized view format. Currently, this must be 1. Implementations must throw an exception if the materialized view's version is higher than the supported version.
required	**`refreshes`**	A list of refresh operations.
required	**`current-refresh-id`**	Boolean that defines the query engine behavior in case the base tables indicate the precomputed data isn't fresh. If set to FALSE, a refresh operation has to be performed before the query results are returned. If set to TRUE the data in the storage table gets returned without performing a refresh operation. If field is not set, defaults to FALSE.

aRefreshes

Freshness information is stored as a list of `refresh operation` records. Each `refresh operation` has the following structure:

v1	Field Name	Description
required	**`refresh-id`**	ID of the refresh operation.
required	**`version-id`**	Version id of the materialized view when the refresh operation was performed.
required	**`base-tables`**	A List of `base-table` records.
optional	**`sequence-number`**	Sequence number of the snapshot that contains the refreshed data files.

Refreshes could be handled in different ways. For a normal execution the refresh list could consist of only one entry, which gets overwritted on every refresh operation. If "timetravel" is enabled for the materialized view, a new `refresh operation` record gets inserted into the list on every refresh. Together with the `sequence-number` field, this could be used to track the evolution of data files over the refresh history.

Base table

A `base table` record can have different forms based on the common field "type". The other fields don't necessarily have to be the same.

Iceberg-Metastore

v1	Field Name	Description
required	**`type`**	type="iceberg-metastore"
required	**`identifier`**	Identifier in the SQL expression.
required	**`snapshot-reference`**	Snapshot id of the base table when the refresh operation was performed.
optional	**`properties`**	A string to string map of base table properties. Could be used to specify a different metastore.

Design 2

Overview

MVs (Materialized views) are realized as a combination of an iceberg table with an associated view. The definition of the materialized view is stored in the associated view. The information required for refresh operations is stored as a property in the `summary` field of each table snapshot. All changes to either the table or the associated view create a new table metadata file and completely replace the old table metadata file using an atomic swap. Like Iceberg tables and views, this atomic swap is delegated to the metastore that tracks tables and views by name.

Metadata Location

An atomic swap of one table metadata file for another provides the basis for making atomic changes. Readers use the version of the materialized view that was current when they loaded the table metadata and are not affected by changes until they refresh and pick up a new metadata location.

Writers distinguish between changing the table or the associated view state.

Writers changing the table state create table metadata files optimistically, assuming that the current metadata location will not be changed before the writer's commit. Once a writer has created an update, it commits by swapping the table's metadata file pointer from the base location to the new location.

Writers changing the associated view state create view metadata files optimistically, assuming that the `associated-view-location` field of the table will not be changed before the writer's commit. The commit is performed in two steps. First, the Writer creates a new table metadata file optimistically and changes the `associated-view-location` field to the new associated view metadata location. Second, the new table metadata file gets committed by swapping the table's metadata file pointer in the metastore from the base location to its new location. The commit is only considered successful when the second step succeeds.

Specification (DRAFT!)

The metadata of the materialized view is comprised of four parts. The table and the associated view metadata constitute one part each. Since not all information can be stored inside the table and the associated view metadata, two additional parts are introduced. One part is stored in the `properties` field of the table metadata. And a part for the freshness information is stored in the `summary` field of each table snapshot.

Materialized view metadata stored in the table properties

One part of the materialized view metadata is stored inside the `properties` field of the table. The metadata is stored in JSON format under the key "materialized_view_metadata". The materialized view metadata stored in the table has the following schema.

v1	Field Name	Description
required	**`format-version`**	An integer version number for the materialized view format. Currently, this must be 1. Implementations must throw an exception if the materialized view's version is higher than the supported version.
required	**`associated-view-location`**	Path to the current metadata file of the associated view.

Materialized view metadata stored in the table snapshot properties

Another part of the materialized view metadata is stored inside the `summary` field of each table snapshot. The following fields are added in additionally:

v1	Field Name	Description
required	**`version-id`**	Version id of the materialized view when the refresh operation was performed.
required	**`base-tables`**	A List of `base-table` records.

Base table

v1	Field Name	Description
required	**`identifier`**	Identifier in the SQL expression.
required	**`snapshot-id`**	Snapshot id of the base table when the refresh operation was performed.