Persistence in Apache Polaris

Motivation

Polaris currently relies on EclipseLink, which is very hard to configure for end users and not convenient in k8s deployments[1]. EclipseLink's support for NoSQL is rather limited to a single relevant database (MongoDB) and transactional behavior, which is rather "standard" for RDBMS, is effectively missing[2]. It also seems that EclipseLink[3] is not very actively maintained, considering the recent commits (also here).

Some use cases, especially bigger multi-tenant/multi-catalog use cases, would benefit from a highly ***horizontally*** scalable database, realistically only a few NoSQL databases. All relevant NoSQL databases[4] support only single-row-level CAS (compare-and-swap) operations, have no multi-row or even multi-database-table transactional behavior.

Bigger use cases are also likely to require replication of catalog data (mainly for Disaster Recovery). Databases that perform distributed transactions are probably going to show lower R/W performance in those scenarios (e.g. Mongo transactions). Reducing requirements on the database technology in this regard (CAS being the common denominator) is valuable to allow broader choices for catalog deployments.

Polaris does not currently store any information, yet, of the managed entities like the contents of table/view-metadata or even manifest (list) information, but reads this information every time from the backing object store, which incurs a significant overhead (at least in public clouds).

Maintaining at least some metadata in Polaris' database is critical for performance, because it is just faster to read from a database than to read even the likely huge metadata[5].

Use cases & features - current and near future

- Allows using nearly all databases as a persistence layer for Polaris, both NoSQL and RDBMS
- Allows horizontally scalable backend databases
- Root pointer store[6] for Iceberg entities (tables and views)
- Management of Iceberg namespaces
- Managing authN + authZ information for the current Polaris implementation

Near future use cases

- Managing authN + authZ information for different authN/Z plugins
- Managing contents of Iceberg metadata (what's in metadata JSON)
- Unique storage locations per table/view

Proposal for a scalable and extensible persistence model

Goals for a new scalable and extensible persistence model:

[1] https://github.com/apache/polaris/tree/main/helm/polaris "kubectl create secret ... --from-file=persistence.xml"
[2] MongoDB has some transactional support, but it's rather limited.
[3] EclipseLink was chosen over Hibernate due to the ASF incompatible license of Hibernate
[4] Examples: Google BigTable, Apache Cassandra, MongoDB, RocksDB
[5] It is sadly still required to store Iceberg table/view metadata in the object store.
[6] A root pointer store manages the location of the metadata location in an object store

- Extensible & pluggable Polaris (“application”) entity object model. It should be a pure “coding exercise” to add a new or enhance an existing object type.
- Efficient, general storage of any serializable type[7] serving the needs for rapidly evolving Polaris and supporting 3rd party extensions/plugins.
- No backend database schema requirements, hence no schema evolution tracking requirements tied to Polaris or its plugins/extensions. Users should not be forced to change the database (schema).
- Atomic multi-entity changes / support changing multiple tables/views/namespaces in a single atomic operation.
- Leverage caching, design to be very cache friendly.
- Persistence model must fit the “least common denominator” for popular databases:
    - Single-row CAS/conditions
    - Optimized for point-queries
    - No database schema required
- Row size “limited” to be “database friendly” (aka not huge like x00MB)[8]
- Efficient (space & performance) indexes
- Allow horizontal scaling of Polaris (backend and database)
- Support persisting “catalog information/metadata” and also internal information like configuration settings and e.g. pluggable content authorization data.
- Suitable for rolling upgrades and downgrades

Object cache

An efficient cache heavily contributes to an overall performance of the whole system.

The persistence model should leverage immutable persisted objects as much as possible, which eliminates the need for distributed cache invalidation (for immutable objects) and potential race conditions[9] caused by mutable cached objects.

Application defined object types

Object types evolve naturally with the application. Adding new attributes to object types and adding entirely new object types is a natural process in software development. While it is convenient for debugging purposes to have a normalized schema in the backend database, it requires a lot of specialized object-mapping and a strict ordering of schema changes. It also requires potentially complex schema management, which is rather complicated in a (rather) stateless architecture.

Storing object values as “blobs” in the database and relying on a standardized object mapping using Jackson is much easier to reason about and eventually more efficient and performant and less prone to bugs.

Considering that the current state of Polaris is rather just the beginning and new functionality will be added regularly and the intent is to support plug-ins/extensions, likely provided by 3rd parties, effectively mandates that the backend database is rather generic / schema-less. The least common denominator is a “blob” with an object-type indicator.

Object storage model

The proposed persistence model does not need a database schema for its tables and therefore works with all databases. The schema for “objects” is rather simple and does not need (semi) automatic schema evolution when adding or changing application-level object types.

[7] Jackson using Smile, which is a space efficient way to serialize JSON data using a binary encoding.
[8] Consider DynamoDB’s hard 400kB row limit
[9] Race conditions wrt (distributed) cache invalidations are described in this interesting blog post

Atomic multi-entity updates

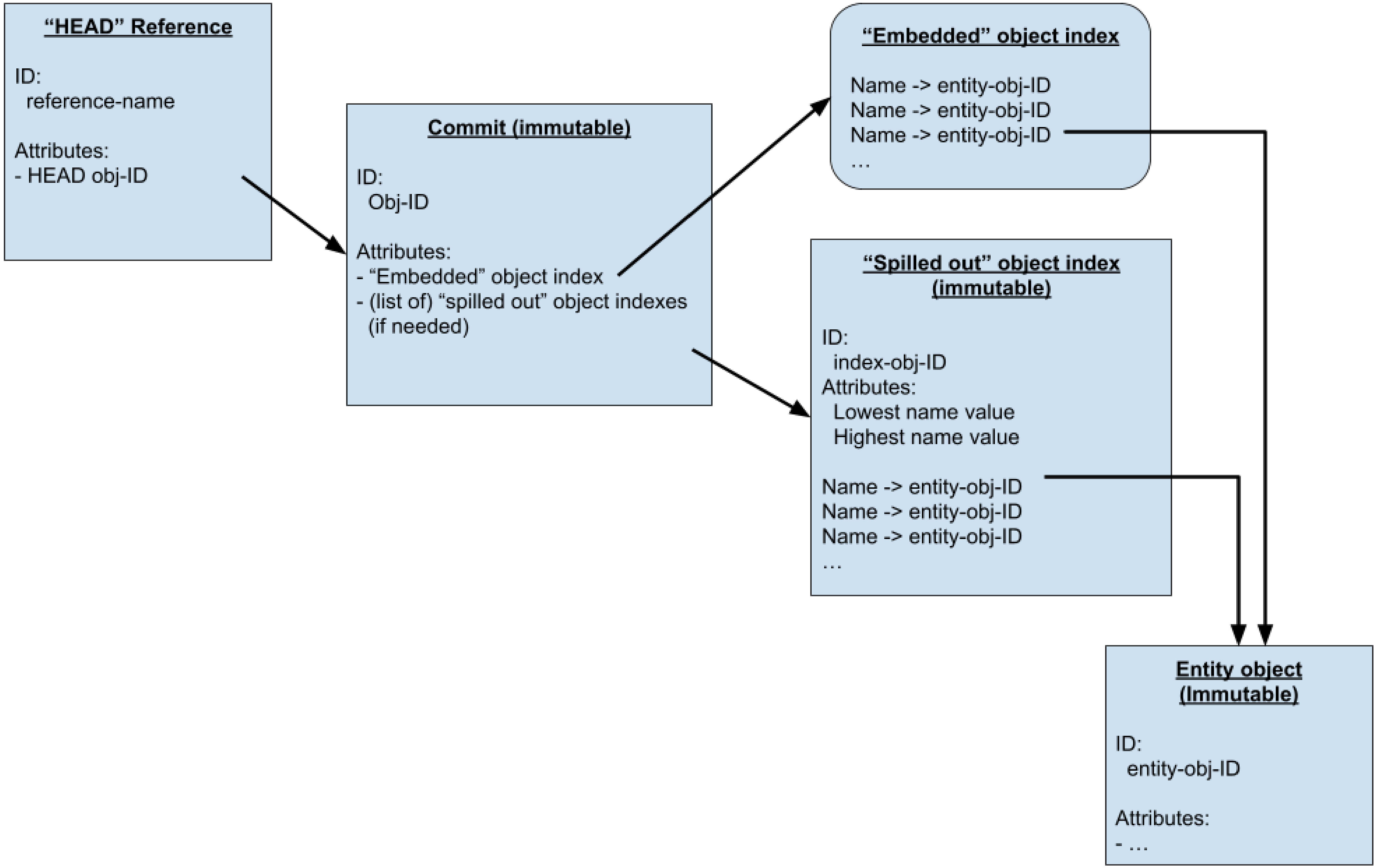
The biggest problem is to support atomic (and consistent) updates to multiple entities (tables, views, namespaces, etc) in a horizontally scalable database requiring only one single-row CAS operation.

This requirement can be achieved by persisting the changes individually as *new* rows and referencing the individual entity-states from a single lookup-object, which has references to related objects organized in a *spillable* index structure to allow nearly endless growth of the “indexed universe”.

The ID of the object that represents the current state (“HEAD”) is stored in a separate, small reference object. Atomic (CAS) operations are used to update the reference object and change the pointer to the current lookup-object.

Committing operations first load the single lookup-object, then apply the changes to it and finally optimistically persist it.

Object Model Schema



Performing a Commit

1. Fetch the “HEAD” reference
2. Fetch the “current HEAD object” with the referenced ID
3. Apply all entity (tables, views, namespaces, etc) changes
    a. Lookup object IDs from the index
    b. Fetch existing entity objects
    c. Perform requirements checks
    d. Fail commit if a requirements check fails
    e. Apply updates to entity object
    f. Persist updated entity object using a new entity-obj-ID
    g. Update object-index for the entity to point to the new entity-obj-ID.
4. Persist updated commit with a new commit-obj-ID
5. Try CAS on the “HEAD” reference
    a. If the CAS succeeds, the commit succeeded -> DONE
    b. If the CAS fails, restart at step #1[10]

Since objects are immutable (only written, never updated), those can be cached w/o the need of a distributed cache invalidation mechanism.

If necessary, the implementation can reduce the computational effort of step #3 by memoizing the results (map of entity-update to already persisted entity object). Memoized results however must only be reused, if the state of the entity (entity-obj-ID) did not change since the last iteration of the commit attempt.

The “HEAD reference CAS” operations are likely as performant as UPDATEs in an RDBMS. However, it is beneficial at least for some database implementations to keep the total row-size of that reference object small, which is also beneficial when reading it (network traffic, (de)serialization effort).

Internal Data Model

Multi-tenancy / multi-catalog

Both the tenant-ID and catalog-ID are in this proposal part of the primary/row keys. This allows adding multi-tenancy later to Polaris w/o having to change the persistence layer.

Object Types

The proposed Polaris persistence model is agnostic to the actually persisted data and its format. It provides the primitives to work with persisted data, but does not have any knowledge of the persisted data itself.

Although, the persistence model maintains a set of known object types, using dynamic discovery mechanisms like Java’s ServiceLoader. Each object type defines a few attributes/capabilities:

- Object type name - a short string, ideally registered with the Apache Polaris project and therefore being reserved
- Mutability - whether objects of this type are mutable. This affects object caching.
- Cache expiry computation functionality for mutable object types (including negative caching)

The object type is part of each object’s persistence primary key, which allows *derived* objects. For example, considering a future requirement where Polaris becomes (persistence wise) more than an Iceberg root pointer

[10] Leveraging exponential backoff, w/ randomization

store. The Object ID of a table's state holding the metadata pointer can be reused for another object type holding related information like the contents of Iceberg's metadata JSON.

Objects

All object types are persisted in a single table. The primary key of each row is a composite of:

- Tenant ID
- Catalog ID
- Object type
- Object ID

The following attributes are present:

- Serialized object value ("blob")
- Version token (see below)
- Created at (timestamp / microseconds), useful for maintenance

Reference

The pointer to the current HEAD (see above) is maintained in a separate table, using a different primary key for each row as a composite of:

- Tenant ID
- Catalog ID
- Reference name

The following attributes are present:

- HEAD object ID
- Created at (timestamp / microseconds), useful for maintenance
- (Limited) list of recent HEAD object IDs

The list of recent HEAD object IDs can be useful in disaster recovery scenarios, when a distributed database is asynchronously replicated to another data center, and globally consistent replication is not achievable. Having some recent HEAD object IDs allows users to *manually* reset the state to another recent previous version that is consistent.

While there is only one state of all Iceberg entities (tables/views/namespaces), it is necessary to reference Polaris internal state like catalog configuration, configuration of and for plugins.

Object Indexes

Indexes are used to resolve fully-qualified table/view/namespace identifiers to object IDs. Those indexes are lexicographically ordered for multiple reasons:

- Ease paging support
- Efficient point / range queries
- Prefix-compression of index keys (omit duplicate identifier prefix strings[11])

While there can be a huge total number (couple 10s or 100s of thousands) of entities (tables, views, namespaces), the amount of frequently changed entities is much smaller.

Recently changed entities are accessible from the object-index "embedded" in the above-mentioned HEAD commit object.

If the "embedded" object-index becomes bigger than some configurable serialized size-threshold, all but the most recent changes are spilled out into separate index objects in the database. The HEAD commit object then also references the spilled-out object-index-object(s).

[11] It is inefficient to store for example the same namespace identifier for every table/view.

A single “index key” can exist in both the embedded and spilled-out (referenced) index objects, in which case the state of the embedded index takes precedence. This means that changes to spilled-out index objects are rather rare and caching those makes lookups extremely efficient and fast.

Both embedded and spilled-out index objects contain a lexicographically ordered list (by entity-key). This enables iterating over entities limited to key prefix, considering “from key” and/or “to key” conditions.

System Managed Entities

Tenants and catalogs need to be managed by Polaris as well, including functionality to create, list/query, fetch, update and delete both catalogs and tenants. This is possible using the above proposed functionalities.

Paging Support

Listing tables/views/namespaces must support paging. Implementations can generate “next page tokens” that reference the same state and with this an immutable view of the listed contents. Leverages the object indexes discussed above.

Handling no longer needed objects

The above proposal mostly writes only new objects but never deletes those. This is a prerequisite to prevent race conditions which would occur if “currently used” objects are needed.

An object in the database is no longer needed, if all of these conditions are met:

- The object is not reachable from the object graph referenced from the “current HEAD”
- The object is “old enough” and therefore not referenced by any concurrent operation (for example commits, listings). “Old enough” means the longest reasonable time to list entities / finish or abort a commit-retry loop

A background process (or maintenance job, see Special case: coordinated tasks below) must regularly (every couple of days, configurable) scan the database for objects that are unreferenced and known to be no longer needed. Those objects can then be safely deleted.

To keep objects that are still being potentially used, it is necessary to retain commits that are not “old enough” to become eligible for removal.

Special case: consistent, updateable objects

There may be some use cases that mandate to update the object directly, if a certain condition matches. The condition match can be generalized by using a “version token”. Writing new updateable objects or updating those would have to happen using CAS operations to ensure consistency and atomicity.

Special case: coordinated tasks

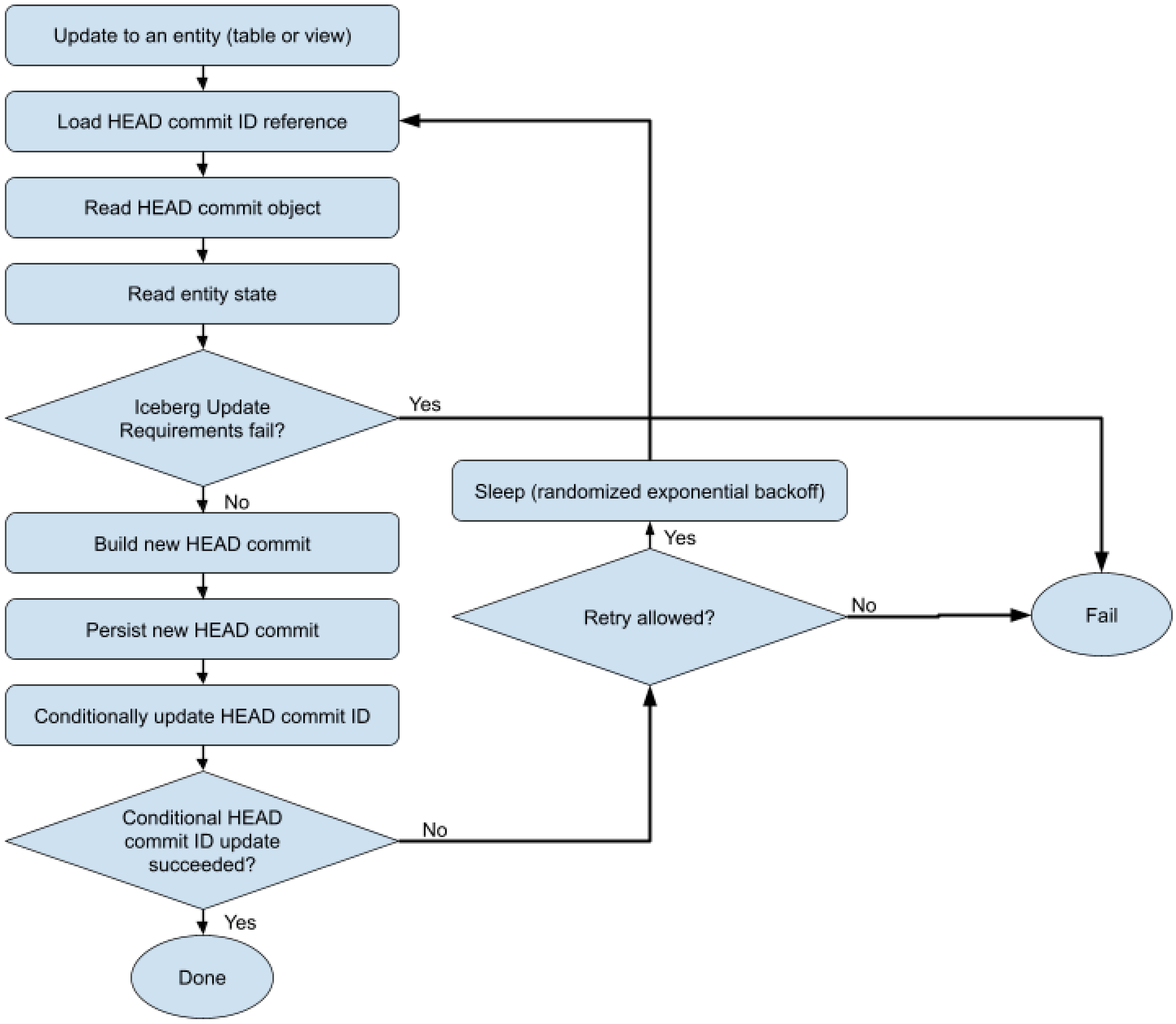
Operations that are rather expensive (time or resource requirements) or have concurrency constraints (e.g. at max one running instance) need some coordination. This can be implemented purely using updateable objects, effectively leveraging the database’s CAS capabilities to coordinate instances. This approach is however only suitable for operations that do not happen very frequently. Examples:

- Coordinating the maintenance job as mentioned in Handling no longer needed objects.

- Coordinating metadata-JSON loads/imports from object stores, when Iceberg metadata is being maintained by Polaris

Contention Discussion

The proposed model has one point of contention, namely the conditional update of the HEAD reference per catalog, which has to be done for every update to any entity (table/view/namespace). And this conditional update is literally the only point of contention in Iceberg REST workloads. The conditional update fails, if any change to another entity happened in the meantime. A retry has to happen in case of a failed conditional update, which has to read the updated HEAD commit and re-check for logical conflicts. If there are no logical conflicts, the entity change can be applied against the new HEAD commit and the update being retried. The flow of this operation is illustrated below.


Update to an entity (table or view)
Load HEAD commit ID reference
Read HEAD commit object
Read entity state
Iceberg Update Requirements fail?
Yes
No
Build new HEAD commit
Persist new HEAD commit
Conditionally update HEAD commit ID
Conditional HEAD commit ID update succeeded?
No
Yes
Done
Sleep (randomized exponential backoff)
Yes
Retry allowed?
No
Fail

Possible Optimization - No Concurrent Retry-Loops per Process

The actual point of "physical" contention happens on the (state of the) commit object and the conditional update (CAS). While it is initially not necessary, and likely overengineered for an initial implementation, there is room for optimization. The process/loop can be serialized inside a Polaris instance process, processing all concurrent changes processed by the Polaris instance process, reducing load from the database. The implementation however becomes more complex.

Possible, Wholistic Optimization - Fine Granular Updates

This approach is not specific to Polaris and discussed below.

Object ID Alternatives

Since Polaris is a distributed, horizontally scalable and (rather) stateless system, the generation of new IDs must be free of contention, for example in a RDBMS database sequence. There are several approaches that can be used to generate IDs. Snowflake IDs seem to be the best compromise, see below.

- Natural keys are IDs that are derived from logical, user provided values like fully qualified table names. Those keys are rather long and require another discriminator to distinguish different versions to form a composite ID to meet the requirements above.
- Hash (SHA-256 and similar) over the object value. The hash would be used as the ID of the object in the database, *assuming* that there are no hash collisions. Ensuring that there are no hash collisions however requires a CAS operation for each write of any object.
- Snowflake IDs (link link link) require 64 bits and consist of three values: timestamp, node ID, sequence number. Each process instance “leases” a node ID, ensuring that no two running processes/generators use the same node ID. Timestamp uses millisecond precision, covering roughly 69 years[12]. The sequence number is there to allow multiple IDs per millisecond. One bit is reserved - the meaning of that bit is not defined (yet?). The size (in bits) of these three values and the beginning of the epoch for the timestamp can be defined, but must stay constant.
- TimeUUIDs (variant 2, version 1 UUIDs) follow a similar concept as Snowflake-IDs, but require 128 bit (2 x 64bit), offering bigger value ranges.

In Polaris is it safe to assume that no two clients write the exact same piece of data, which in turn means that using a hash based mechanism to detect duplicates is rather fruitless effort.

Generating IDs locally (within an operating system process) is a pretty quick operation compared to remote calls, although it requires synchronization mechanisms within the process. Being able to use locally generated IDs requires a mechanism to “lease” node-IDs. The process to lease node-IDs however is not performance critical, because it is performed at process startup.

Coordination of actively leased node-IDs can be performed in the backend database using the following attributes.

- Node ID (primary key)
- Leased until (timestamp) - new value should be “now() + some time”. Active processes must renew the lease before it expires.
- Lease token (random string token) - used for CAS operations

Polaris specific snowflake ID parameters

- Epoch: Jan 1st 2025 00:00:00.000 GMT
- Node ID: 10 bits (max 1024 nodes)

[12] It is probably okay to assume that after this timespan there are either 128 bit integer values everywhere and memory/size requirements are less of an issue than today.

- Timestamp: 41 bits (~69 years)
- Sequence: 12 bits (4096 IDs per millisecond per node)

Allocating a node ID

To allocate a new node ID, the following steps are needed:

1. Choose a random node ID
2. Load the row for the node ID
3. If the row does not exist, attempt to insert it (CAS)
4. If the row exists and leased-until is in the past, attempt to update it (CAS)
5. If the row exists and leased-until is not in the past, go to step 1

The objects used to maintain node ID leases can be persisted in the same way as other objects, leveraging updateable objects. The object IDs for this use case however need to use a special timestamp, which could be “timestamp = 0”, and a fixed “sequence = 0”.

In general, it feels beneficial to reserve the first 1000 timestamp values since epoch for this and potential other special cases.

Renewing a lease

A node ID is leased for a specific amount of time, which means that the leased-until attribute must be regularly updated. The update should happen with a CAS and set a new lease-token, comparing the old lease-token.

Managed Iceberg Metadata

The implementation should manage Iceberg metadata, at least the information in table/view-metadata. A proposal for the “Polaris object model” for Iceberg metadata is still to be defined. It should however not duplicate “big” pieces of information and must not store “huge blobs” (aka: full table-metadata serializations with hundreds/thousands of snapshots).

Polaris could automatically perform operations like “expire snapshots” on the fly, eliminating the need to run those maintenance tasks manually.

Current Iceberg proposals that involve access to manifest (list) information should already be considered in a proposal for the Polaris object type model for Iceberg metadata.

Loading/importing Iceberg Metadata

Polaris is currently mostly “only” a root pointer store, holding the location of the Iceberg metadata-JSON. Once Polaris is able to manage Iceberg metadata, there will be entities that have only a location but no metadata in Polaris’ database and entities that have a location *and* metadata stored in Polaris.

The state of each table is both reflected in Polaris and persisted (rather as a copy/as an export) in the object store. There must be a process to load metadata of an existing state of a table into Polaris.

Iceberg REST - Fine Granular Updates

(This chapter requires rather fundamental changes to at least Iceberg’s REST Catalog implementation and at least additions to the Iceberg REST protocol.

Iceberg REST table/view updates from clients contain information and IDs that have been fully computed by the client, which have no knowledge about other concurrent updates to the same entity. This can lead to duplicate values for schema IDs, field IDs, etc and more importantly conflicting snapshot additions, which is a problem for streaming use cases.

Such "logical" conflicts (conflicts that are not caused by the persistence layer) are by far the most expensive conflicts, because those need to be handled on the client side and require quite some round trips including object store accesses.

With fine(r) grained table/view updates and delegating ID generation to the catalog, the probability of running into "logical" conflicts will be massively reduced, resulting in better overall performance and a better overall experience.

One of the most common scenarios is the *append-files* operation. Letting the catalog resolve those "logical" conflicts is the better approach (reasons mentioned above).

A common source of conflicts are maintenance operations like *expire-snapshots* or *compaction*, which also run for quite some time.

Reference: Atomicity in various NoSQL databases

Database (Alphabetically ordered)	**Capabilities**	**Notes**
BigTable	Conditions on a single row	
Cassandra	Conditions on a single row	Batch DML is not atomic (across keys).
DynamoDB	Conditions on a single row	
FoundationDB	Transactions	Very difficult to use[13].
MongoDB	CAS-like API (insertOne, findOneAndUpdate, …)	Bulk/"many) operations are not atomic.
RocksDB	n/a	Single-process DB. Conditions can be implemented in the calling code.

Reference: Iceberg Metadata

All IDs used in Iceberg metadata are generated by clients. It would be safer to enforce that all IDs must be generated by Polaris. ID generation in Iceberg should be a concern of the Iceberg `SessionCatalog` and related implementations.

[13] FoundationDB is very hard to install and maintain in CI and especially on developer machines, because binary packages are mandatory, client/server versions must exactly match.

Tables

- Metadata
    - Format version
    - Location
    - UUID (client assigned)
    - Property bag
    - Last sequence number (client assigned)
    - Last updated timestamp (client assigned)
    - Last column ID (client assigned)
    - Schemas
        - ID (client assigned)
        - …
    - Current schema ID (client assigned)
    - Partition specs
        - ID (client assigned)
        - …
    - Default partition spec ID (client assigned)
    - Last partition ID (client assigned)
    - Sort orders
        - ID (client assigned)
        - …
    - Default sort order ID (client assigned)
    - Current snapshot ID (client assigned)
    - Snapshots
        - ID (client assigned)
        - Sequence number (client assigned)
        - Parent snapshot ID (client assigned)
        - Timestamp (client assigned)
        - Summary (property bag)
        - Manifest list (client assigned)
        - Schema ID (client assigned)
    - Snapshot Log
        - Snapshot ID (client assigned)
        - Timestamp (client assigned)
    - History
        - Path (client assigned)
        - Timestamp (client assigned)
    - Statistics files
        - Snapshot ID (client assigned)
        - Path (client assigned)
        - Blob metadata (list)
            - Type
            - Snapshot ID (client assigned)
            - Sequence number (client assigned)
            - Field IDs
            - Property bag
    - Partition statistics files
        - Snapshot ID (client assigned)
        - Path (client assigned)
    - Refs
        - Snapshot ID (client assigned)

        - Type
        - Min-snapshots-to-keep
        - Max-snapshot-age
        - Max-ref-age
- Manifest list
    - Manifest file references
        - Path (client assigned)
        - Partition Spec ID (client assigned)
        - Added snapshot ID (client assigned)
        - Sequence number (client assigned)
        - Min sequence number (client assigned)
        - (file/row stats)
        - Partition field summary
- Manifest file
    - Data/delete file references
        - Snapshot ID (client assigned)
        - Sequence number (client assigned)
        - File sequence number (client assigned)
        - Path (client assigned)
        - Format
        - Partition value
        - Per-field upper/lower bounds (can be very big!)
        - …
- Data/delete files
    - …

Views

- Metadata
    - Location
    - Property bag
    - Schemas
        - (see above)
    - Current-Version-ID
    - Versions
        - ID (client assigned)
        - Summary (property bag)
        - Default catalog
        - Default namespace
        - Representations
            - Key
            - SQL
            - Dialect
    - View History
        - Version-ID
        - Timestamp