Pluggable Key-Value Store for Upserts

09/2023

Authors: random.shuffle(Aravind Suresh , Pratik Tibrewal) | Uber
GitHub Issue: #11658

Summary

As we know, Pinot's upsert functionality requires an in-memory map that tracks the primary keys to the corresponding record locations. This is used to find if a record with that primary key already exists and if so, merges that with the incoming record. For tables with a large number of primary keys, this leads to huge memory consumption because this map is stored in the heap memory.

In certain use-cases at Uber, we came across tables that required a longer retention period and a strict level of correctness, so we explored alternatives on replacing this in-memory map with disk-backed maps.

The current implementation of Pinot Upserts (see ConcurrentMapPartitionUpsertManager) is heavily coupled with the in-memory map (Java's ConcurrentHashMap). This reduces the flexibility for Pinot adopters to replace this Map with their own implementation of this map. This write-up talks about how we can make this Map pluggable and our initial experiments around this.

Proposed Implementation

The following table highlights the usages of ConcurrentHashMap's methods inside "ConcurrentMapPartitionUpsertMetadataManager".

Method	**Where?**
Compute	While adding or replacing a segment, we sometimes point to old / new segment accordingly While adding record, we differentiate between existing / new primary keys using this
ComputeIfPresent	Removing key from the map if a segment is getting removed While updating record, we use it to merge incoming and existing records

Remove	Removing old primary keys based on upsert metadata TTL
ForEach	Iterating over all primary keys to remove the old ones during TTL
Size	To publish metrics and logging

An interface “UpsertKeyValueStore” with the above methods can be extracted out (or we can reuse Java's ConcurrentMap interface with some default implementations for other methods).

The usage of the keyValueStore will be behind an upsert config : ***upsertKeyValueStoreClass***. If not mentioned, we will use the present ConcurrentHashMap itself.

```
"upsertConfig": {
    "mode": "PARTIAL",
    “upsertKeyValueStoreClass”:”org.apache.pinot.plugin.MyUpsertKeyValueStore”
}
```

Implementation Plan

1. Introduce “UpsertKeyValueStore” interface or reuse “ConcurrentMap” interface – **open to review by the community**
2. A minor refactor of “ConcurrentMapPartitionUpsertMetadataManager” to bind an implementation of “UpsertKeyValueStore” to “ConcurrentHashMap”
3. Introduce an upsertConfig to instantiate the Map object based on the input or default to present ConcurrentHashMap

FAQs

1. Are we proposing to change the default map strategy from ConcurrentHashMap to some other key-value db?

    No. At present we want to change the hard-coded concrete implementation of ConcurrentHashMap in the upsert-flow to a pluggable key-value store interface. The default implementation of it will be ConcurrentHashMap like present.

2. Why not create your own PartitionUpsertManager and do all this internally? The PartitionUpsertManager is also pluggable.

    There are a lot of changes that keep on going inside the PartitionUpsertManager class. We don’t want to have the overhead of maintaining our own manager with the open source

PartitionUpsertManager class from time-to-time. Our use-case focuses mainly on using different Key-Value store and we still want to keep using the new features without any concerns.

Appendix

This section talks about a few experiments we did in Uber recently with different techniques of storing (PrimaryKey -> RecordLocation). This has no implications on any OS workflows.

For one of our tables in Uber, we've noticed that not all primary keys get updated at the same rate. To be precise, there were cases where >95% of the primary keys wouldn't be actively used after the initial few days (4-5 days) of creation. So, we tried having a "HybridMap" that wraps two underlying maps:

1. A "primary" fast map with a bound on number of keys (defaults to ConcurrentHashMap)
2. A "secondary" slow map, but can handle large number of keys
3. An in-memory bloom filter for all the keys that are present in the "secondary" map – this enables us to avoid secondary.Get call for non existent keys
4. An LRU based movement of keys from primary to secondary maps

The key behaviours of the "HybridMap" are documented below:

1. Put(key, value):
    a. We would "put" it to the "primary" map alone.
2. Get(key):
    a. If it's present in the "primary" map, we'd return the value.
    b. If it's not present in the "primary" map, we check the "secondary" map and then return the value. We also insert this into the "primary" map here to support recency.
    c. Note that step-b would happen for **all the new records**, so we can introduce an in-memory bloom filter over the secondary map here to check if we need to really read from the "secondary" map as those reads would incur higher latency.
3. Remove(key):
    a. If it's present in the "primary" map, we remove it.
    b. We also check if it's present in the secondary map (via bloom filter), if so we attempt the removal.
    c. Return coalesce(oldValueFromPrimary, oldValueFromSecondary).
4. Size():
    a. If the underlying "secondary" map has an O(1) size function, then we invoke it. Else we wrap the "secondary" map with an approx distinct counter (HyperLogLog++).
    b. Return the size of the "secondary" and "primary" maps.
    c. We are okay with approx here as the existing usages are mainly for logging / metrics. We can also introduce approxSize() methods in the interface for clarity.

Overflow of Keys

Overflow refers to the insertion of "older" keys into the secondary map and the corresponding eviction of those keys from the primary map. Note that the term older could refer to a FIFO / LRU mechanism of overflow but we focus on pluggable implementations for this overflow policy.

This overflow can happen in a scheduled thread that periodically identifies keys that need to be overflowed and moves them. The exact time complexity of this depends on the implementation:

1. LRU based on comparison values would be O(n log k), where n is the number of keys in the primary map, k is the number of keys getting overflowed
2. Approx LRU can be implemented in O(n) using a theta sketch to compute approx percentiles over the comparison values in O(1)

Overall, the idea is we can implement this overflow policy such that the amortised overhead per operation (get / put / remove) on the HybridMap is O(1) / O(log k).

We had 3 different implementations of "SlowMap":

1. Chronicle's disk based map
2. MapDB's disk based map
3. MapDB's implementation of HybridMap (which does overflow automatically by itself)
4. RocksDB based map

Out of this, we tuned and tested "#3" for 2 of our tables and these are the results for the comparison of ingestion delay between the Baseline (the existing large in-memory map) and the HybridMap.

Option	**Metric**	**Baseline (ConcurrentHashMap)**	**HybridMap**
MapDB's HybridMap + Bloom Filter	p90	181 ms	195 ms
MapDB's HybridMap + Bloom Filter	p75	121 ms	125 ms
MapDB's HybridMap + Bloom Filter	p50	87 ms	89 ms

We'll conduct more experiments in the future (for different implementations of SlowMap and for different access patterns) and then furnish the results here.

Again, the goal here is to make the **Upsert PrimaryKey Map pluggable** so that it can be replaced with an implementation that suits the Pinot adopters. We will anyways **default to the existing ConcurrentHashMap implementation**.

If the community agrees, we can definitely provide the HybridMap as an OS functionality in future.