Upsert table primary key TTL support

Note: it's recommended to read the full upsert design doc to understand overall upsert requirements and design

This document is shared externally
Status: WIP

Authors

- Qiaochu Liu qiaochu@uber.com

Approvers

- Yupeng Fu yupeng@uber.com (Approved)
- Jackie Jiang jackie.jxt@gmail.com
- Ujwala Tulshigiri ujwala@uber.com (Approved)
- Ting Chen tingchen@uber.com (Approved)

Motivation

Apache Pinot provides native support of Upsert since v0.6.0, it allows users to modify existing records, and successfully onboard many use cases. However, some challenges and feature gaps were identified during customer onboarding.

One issue we observed in Pinot upsert cluster is high usage of heap memory. This is because the upsert metadata, reflecting mapping from primary keys to record locations, is stored in heap of each host. For use cases with high cardinality of primary keys, the heap usage of these upsert tables usually becomes the bottleneck of the hardware resource.

For example, one common use case for Pinot upsert is dedup use cases, it has a large number of unique combinations of primary keys. It's hard to store large amounts of primary keys in memory. Another example can be, in an upsert table, records with specific primary keys will get updates frequently during a time window, and after the time window, these records won't get updated any more. For example, an Uber trip state might get updated when it is assigned, picked up, and dropped off. And the state will never be updated once the trip is completed.

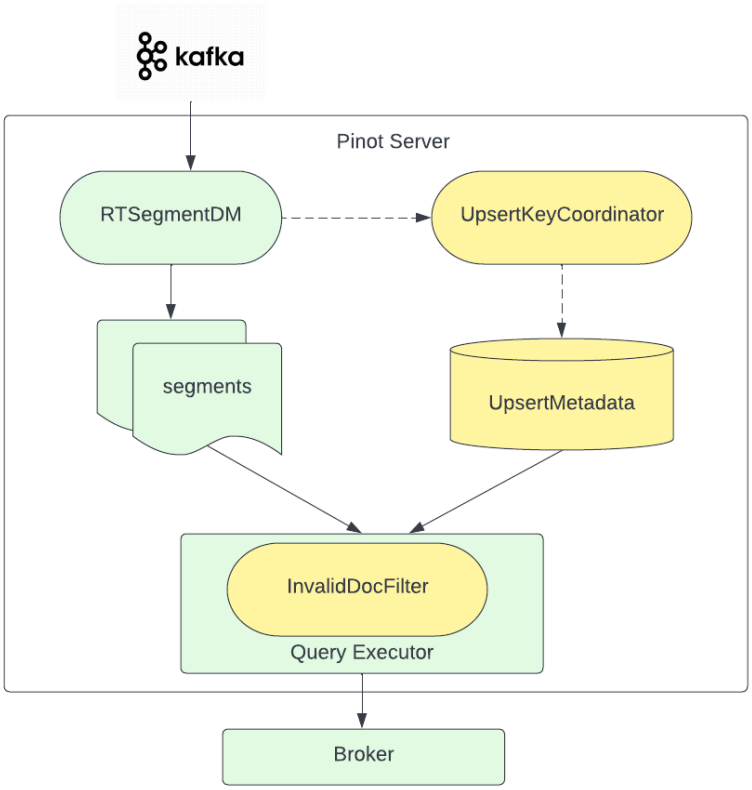
In these use cases, each primary key has a lifecycle and will be deactivated after the time window. Currently these primary keys won't expire until the retention days. We shall introduce TTL (time-to-live) for Pinot primary keys. Primary keys will expire after the TTL, and we can remove inactive keys from upsert metadata to save heap space.

Background

Pinot upsert revisit

The pinot upsert architecture is shown in the diagram below. For an upsert use case, the source kafka topic needs to be pre-partitioned by primary keys, so the records with the same primary key can be distributed

to the same partition. Upsert happens in the local key coordinator, updating valid records with the same primary keys.

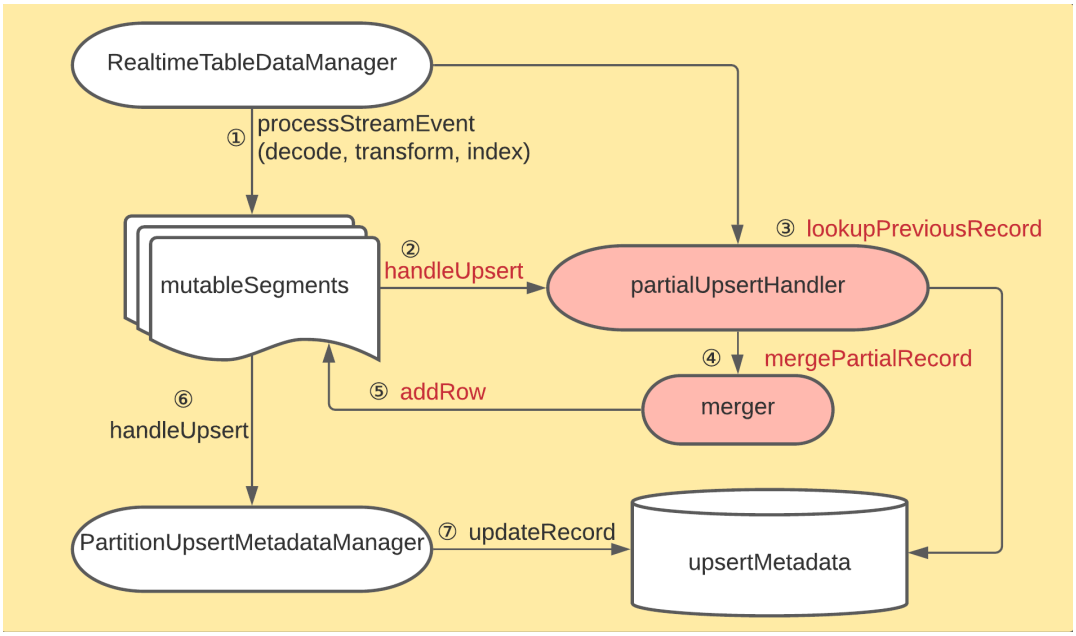


Pinot upsert architecture has the following major components:

- **PartitionUpsertMetadataManager**, also known as local key coordinator, manages the upsert/insert in each partition. PartitionUpsertMetadataManager maintains the upsert metadata for each partition, including primary key to record location.
- **PrimaryKeyIndexes**, also known as upsertMetadata, is the mapping from primary keys to record location. RecordLocation contains information about segment_id, doc_id, and comparison column values. When a message with existing primary key values comes, we update the record location of this primary key if the new record has a higher comparison column value.
- **ValidDocIndexes**, also known as SkipDocFilter, is a filter bitmap that contains valid records in a segment. When a message with existing primary key values comes, we use the comparison column value to decide whether it's a valid record or not. When processing queries, Pinot servers skip invalid records by filtering by validDocIndexes.

Partial upsert revisit

Partial upsert is built on top of pinot full upsert. During ingestion, the partial upsert handler processes the incoming message, handles merge and derives the full update for the row in the mutable segment. The pinot partial upsert architecture is shown in the diagram below.



Besides components in full upsert architecture, partial upsert has two special components..

- **PartialUpsertHandler** is a local handler that identifies the partial upsert merge strategies for each field to initialize a list of mergers, looks up the last derived full update record, and handles merges.
- **Mergers** handle the merge of the previous full record and the new consumed record with different strategies, including updates, ignore, increment, append strategies, etc.

Challenges

During onboarding upsert use cases, we observed few challenges and scenarios that can be optimized.

High memory usage of Upsert metadata

A common upsert use case is live sessions. There are scenarios that users need to update records with specific primary keys frequently during a time window, and after the time window, these records won't get updated any more. For example, an Uber trip state might get updated when it is assigned, picked up, and dropped off. And the state will never be updated once the trip is completed. Currently these use cases rely on Flink Sessions to maintain the time windows for active primary keys. Each primary key has a lifecycle and will be deactivated after the time window.

We store upsert metadata(PrimaryKeyIndexes) in the heap, which is a map data structure to look up recordLocation by PrimaryKeys. The heap usage of these upsert tables usually becomes the bottleneck of the hardware resource. Early this year, we introduced a tool to estimate heap usage for Pinot upsert tables. There are some other efforts to reduce heap usage and avoid out-of-memory issues.

- Primary keys size compression, primary keys can be stored as hash values and compressed.
- Cardinality reduction, cardinality is the number of unique records of primary keys stored in heap.

Slow segment rebuilt for upsert tables

During segment reload, or server restart, in-memory states will be lost, and segments need to be rebuilt.

We observed that server reload and restart for upsert tables took significantly longer time than non-upsert tables.

Architecture

Based on the problems identified, below are some upsert optimizations.

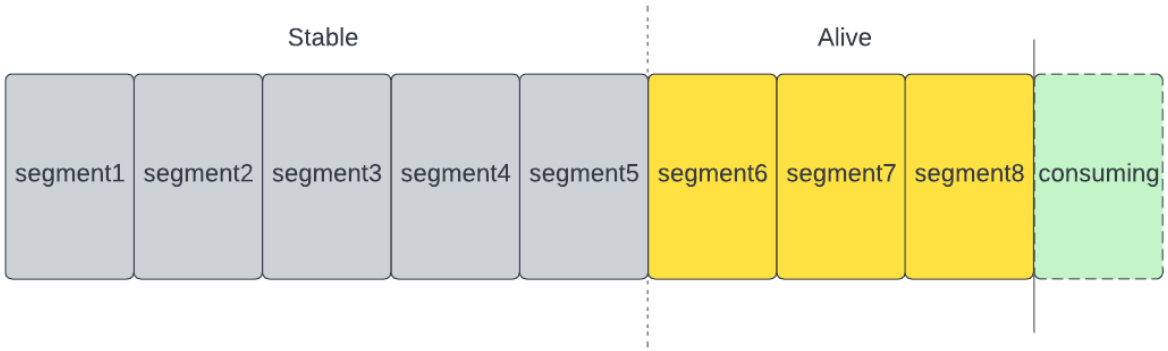
TTL support for primary keys

Currently for upsert tables, the primary key values within retention will be stored in memory. Because of this pattern, we cannot support upsert tables that have long retention days. For use cases where primary keys will be alive for short periods of time, we shall introduce TTL(time-to-live) for Pinot primary keys. For TTL-enabled use cases, the primaryKeyIndexes only contain primary entries within TTL. Primary keys will expire after the TTL, and we can remove them from the heap to save heap space. When reloading or restarting the servers, the validDocIdsSnapshots will be used to rebuild segments.

There are two limitations of TTL support,

- TTL should be longer than consuming segment sealing time.
- TTL is not compatible with upsert data correction backfill.

With the TTL introduced, now the data stored in the pinot upsert table can be splitted into “stable” and “alive”. It can be illustrated in the following figures.



We can discuss the behavior for consuming new messages and reloading the existing segments.

Consuming behavior with TTL

There are two types metadata we need to maintain for Pinot upsert,

- Primary key Indexes per partition (PKI)
- Valid Docs Indexes per segment

When a new message comes, Pinot will try to lookup the primary key in PKI to find out the location of the previous valid record if it exists. It can be described by the following code.

```
def ingestionWithUpdate(PrimaryKeyIndex upsertMetadata):

 while msg := kafkaConsumer.poll() and msg.time > currentTime - ttl do:
     if msg.primaryKey in upsertMetadata then:
        if msg.time > upsertMetadata.get(msg.primaryKey).time then:
          upsertMetadata.remove(msg.primaryKey)
          upsertMetadata.put(msg.primaryKey, IndexTuple(segmentName,docId,msg.time))
     else:
        upsertMetadata.put(msg.primaryKey, IndexTuple(segmentName,docId,msg.time))
```

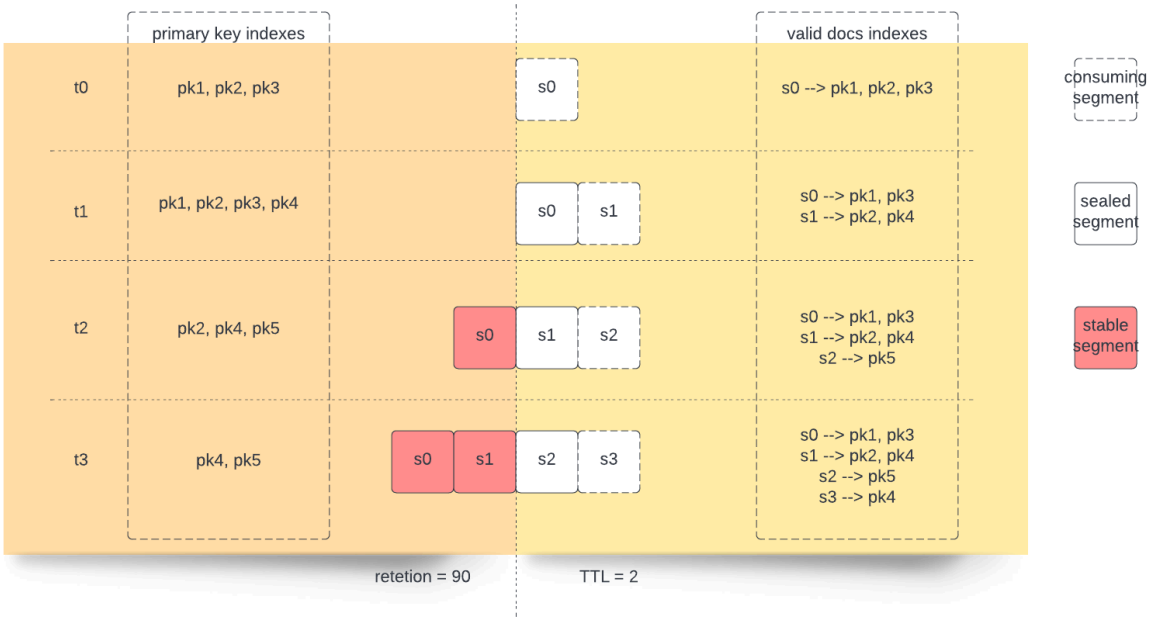
With TTL support, the consuming behavior can be categorized into the following 3 scenarios.

Since we cannot always guarantee all the out-of-TTL primary keys can be removed from primary key indexes immediately. We might have the following three scenarios.

	Behavior (current message is incoming)
Scenario1. Previous record is **out of TTL**, and removed from PK indexes	• The previous record **won't be stored** in the primary key indexes. • The previous doc is valid for the stable segment. • The incoming message will be added in the primaryKey indexes. • The new doc will be valid for the new segments. • The previous doc **won't be invalidated**. • When querying the table, we will get **two records**. • We don't need to periodically update validDocIds snapshot
Scenario2. Previous record is in TTL, haven't been removed from PK indexes	• The previous record is stored in the primary key indexes. • The previous doc is valid for the alive segments. • The incoming message will update the primaryKey indexes. • The previous doc will be invalid for the alive segments. • The new doc will be valid for the new segments. • When querying the table, we will get **one record**. • We need to periodically update validDocIds snapshot
Scenario3. **The old event is out of TTL but still in the primary key indexes**	• The previous record is **stored** in the primary key indexes. • The previous doc is valid for the stable segments. • The incoming message will be added in the primaryKey indexes. • The new doc will be valid for the new segments. • The previous doc can **be invalidated but we shouldn't do that.** • Additional check: when inserting a new record, even the pk can be found in PK indexes but out-of-TTL, we should treat it as a new record. (we need all-replica to have the same behavior, otherwise the server might crash). • When querying the table, we will get **two records**. • We don't need to periodically update validDocIds snapshot

The scenarios 1 and scenarios 2 can be explained in the following figure.

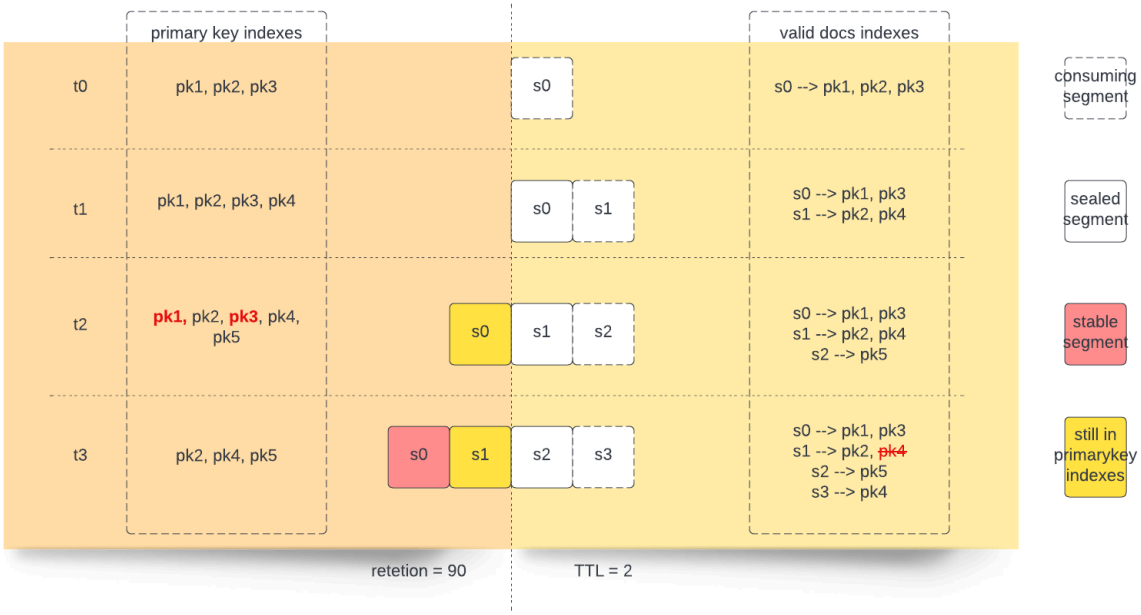
Ts and new segment	Pk1 valid location	Pk2 valid location	pk3 valid location	Pk4 valid location	Pk5 valid location
T0, S0 (pk1, pk2, pk3)	s0	s0	s0	-	-
T1, S1 (pk2, pk4)	s0	s1	s0	s1	-
T2, S2 (pk5)	s0	s1	s0	s1	s2
T3, S3 (pk4)	s0	s1	s0	s1, s3	s2



As we can see from the above result, we will get 2 records for pk4.

For scenario3, we can explain it with the following figures.

Ts and new segment	Pk1 valid location	Pk2 valid location	pk3 valid location	Pk4 valid location	Pk5 valid location
T0, S0 (pk1, pk2, pk3)	s0	s0	s0	-	-
T1, S1 (pk2, pk4)	s0	s1	s0	s1	-
T2, S2 (pk5)	s0	s1	s0	s1	s2
T3, S3 (pk4)	s0	s1	s0	~~s1,~~ s3	s2



The above cases show when the out-of-TTL primary key hasn't been removed from primary key indexes. They can still be looked up and invalidated in the previous segments. E.g. When querying pk4, we will only get 1 record.

However this might cause servers to crash. We should guarantee that all replicas have the same behavior, and keep the pattern consistent by adding the following additional check: when inserting a new record, even the pk can be found in PK indexes but out-of-TTL, we should treat it as a new record. When adding this rule, the scenario3 will become the same as scenario 1 and 2.

Note that valid doc indexes are inferred from primary keys indexes. With TTL support, the primary key indexes containing partial primary keys won't be sufficient. We will need to keep validDocIdsSnapshot for segments out of TTL. Their primary keys are not stored in the primary key indexes anymore and validDocIndexes cannot be inferred based on primaryKeyIndexes.

ValidDocIndexes Snapshots

We shall introduce **validDocIdsSnapshot** to recover in-memory metadata quickly. When sealing a segment, validDocIdsSnapshot will be saved periodically. When segment reloads, the sealed segments will use the validDocIdsSnapshot to recover the ValidDocIndexs instead of inferred from the primaryKeyIndexes.

It's very common that the process can be restarted in the Pinot servers. The in-memory upsert metadata, primaryKeyIndexes and validDocIndexes, will be lost and need to be rebuilt. However, after ttl enabled on primary keys, we only kept active primary keys within TTL in the upsert metadata, and we lost the records location of inactive primary keys.

The solution for this problem is to make validDocId snapshots for out of retention TTL segments. The snapshot is stored as roaringBitmap, can be maintained per segment, and when loading the segment again, only try to add the records that are valid to the metadata map. Two challenges for persisting validDocIdsSnapshot are as follows,

- When to persist the snapshot to avoid inconsistent results.
- Where to persist the snapshot so that we are able to access the snapshot.

Implementation

ValidDocIds Snapshot for recovering in-memory states

A solution for recovering metadata is to make validDocId snapshots for out of retention TTL segments. The snapshot is stored as **roaringBitmap**, can be maintained per segment, and when loading the segment again, only try to add the records that are valid to the metadata map.

The validDocIds recovery process is shown below.

When to persist snapshot

We listed all the scenarios that validDocIndexes might change. E.g. Persisting snapshots during segment replacement might cause inconsistency validDocIndexes, and it was addressed by this OS commit.

When to persist snapshots are shown as follows.

	snapshot enabled	disabled
Add segment	Load validDocIds from snapshot	Delete snapshot
Replace segment	-	-
offload	Persist snapshot	-

During segment replacement, it shouldn't use the old segment's snapshot to recover validDocIds for the new segment. This is because the new segment doesn't necessarily have the same docs as the old segment. Even for consuming segments, we might re-order the docs.

The below figure shows which segments we need to update validDocIndexes.

	In PrimaryKeyIndexes (can recover from primaryKeyIndexes)	validDocIndexes can change	timeRetentions	Need update snapshot
Sealed Segments in TTL	yes	yes	ttl	yes
It should be removed - but it's still in TTL	yes	yes	Ttl + x	yes
Stable segments. Not in pk indexes.	no	no	Retention - ttl	no

Where to persist snapshot

Option1. store it in a hdfs url, maintain extra metadata

The current snapshots are persisted in the local host. And it might be lost during the server replacement. We planned to add the snapshot in deepstore and add the snapshot hdfs url in the segment metadata


```
{
  "segment.crc": "-1",
  "segment.creation.time": "1658278014231",
  "segment.end.time": "-1",
  "segment.flush.threshold.size": "2500000",
  "segment.flush.threshold.time": null,
  "segment.index.version": null,
  "segment.name": "rta_ads_metrics_events__0__0__20220720T0046Z",
  "segment.realtime.download.url": null,
  "segment.upsert.snapshot.download.url": “hdfs:///upinot/etc/segment_timestamp_00000”,
  "Segment.upsert.snapshot.last modifid.time": "1658278014231",
  "segment.realtime.endOffset": null,
  "segment.realtime.numReplicas": "2",
  "segment.realtime.startOffset": "37967961",
  "segment.realtime.status": "IN_PROGRESS",
  "segment.start.time": "-1",
  "segment.table.name": "rta_ads_metrics_events",
  "segment.time.unit": "null",
  "segment.total.docs": "-1",
  "segment.type": "REALTIME"
}
```


Option2 Store validDocIds in the segment tarball

We can also store the validDocIds in the tarball using the `SegmentCreator`

However, validDocIds won't be available when creating the segment. We can only pull it from the server when the segment is loaded.

Pros and cons

	pros	cons
Store separate hdfs url	Lightweight (snapshot in bitmap format is smaller than tarball)	(1) Have to manage a different HDFS dir, hard to manage (2) Enabling/disabling the snapshot on the fly will not be easy. (3) Sometimes HDFS urls can be lost.
Store snapshot in the tarball	Easy to manage	Might be expensive to tar and upload

Pruning out-of-TTL primary keys

When removing expired entries from upsert metadata, the expired keys should be removed from primary key indexes. There are two approaches.

Remove synchronously

Everytime when committing a new segment, we can check the out-of-TTL primary keys and remove them from primary key indexes. It can be described in the following figure.

Remove asynchronously

Full table scan. The upsert metadata gets pruned periodically by scanning the map, and comparing the timestamp of recordLocation with the current timestamp.

```
def pruneMetadata():
	for primaryKey, recordLocation in upsertMetadata:
		if recordLocation.ts < currentTs - ttl:
			upsertMetadata.remove(primaryKey)
```

Two maps rotation. We can use two maps to store TTL*2 upsert metadata to reduce the scan time. One map stores records with timestamp [currentTS - 2*ttl, currentTS - ttl], while the other map stores records with timestamp [currentTS - ttl, currentTS].

When pruning records, we can fully remove all records in the map maintained earlier records, then swap two maps.

```
def ingest(primaryKey, recordLocation):
	if recordLocation.ts > currentTs - ttl:
		if primaryKey in upsertMetadataOld:
			upsertMetadataOld.remove(primaryKey)
			upsertMetadataNew.put(primaryKey, recordLocation)
		elif primaryKey in upsertMetadataNew:
			if recordLocation.ts > upsertMetadataNew.get(primaryKey).getTs():
				upsertMetadataNew.put(primaryKey, recordLocation)
		else:
			upsertMetadataNew.put(primaryKey, recordLocation)

def pruneMetadata():
	upsertMetadataOld.clear()
	swap(upsertMetadataOld, upsertMetadataNew)
```

Recovering PrimaryKeyIndexes

For sealed segments, we rebuild the PrimaryKeyIndex by scanning the DocId, PK and TS of all the sealed segments. With these type columns, we are able to fully reconstruct the PrimaryKeyIndex map.

```
def ingestionWithUpdate(PrimaryKeyIndex upsertMetadata):

 while msg := kafkaConsumer.poll() and msg.time > currentTime - ttl do:
     if msg.primaryKey in upsertMetadata then:
```

```
        if msg.time > upsertMetadata.get(msg.primaryKey).time then:
          upsertMetadata.remove(msg.primaryKey)
          upsertMetadata.put(msg.primaryKey, IndexTuple(segmentName,docId,msg.time))
     else:
        upsertMetadata.put(msg.primaryKey, IndexTuple(segmentName,docId,msg.time))
```

Recovering ValidDocIndexes

The original approach to recover validDocIndex is inferred from primaryKeyIndexes. However, for TTL-enabled use cases, the primaryKeyIndexes only contain primary entries within TTL. One approach is to recover the full primaryKeyIndexes, but it will be expensive and time consuming. To solve this requirement, we proposed validDocIdsSnapshot to recover validDocIndexes. When sealing a segment, validDocIdsSnapshot will be saved periodically. When segment reloads, the sealed segments will use the validDocIdsSnapshot to recover the ValidDocIndexs instead of inferred from the primaryKeyIndexes.

```
def reloadWithoutTTL(segment, records, primaryKeyIndexes):
       for record in records:
              recordLocation = primaryKeyIndexes.get(record.primaryKey)
              if recordLocation != null:
                     ## existing record, same segment
                     if segment == recordLocation.getSegment():
                            if compareTime > 0:
                                   segment.validDocId.replace(record)
                     ## existing record, same segment being removed
                     elif segmentName == currentSegmentName:
                            if compareTime > 0:
                                   segment.validDocId.add(record)
                     ## existing record, different segment
                     else:
                            if compareTime > 0:
                                   segment.validDocId.add(record)
              ## new record
              else:
                     validDocId.add(record)


def reloadWithTTL(segment, records, validIdSnapshots):
       segment.validDocId = validIdSnapshot
```

When replacing the old snapshot, we replicate the old snapshot, generate a new snapshot then replace the old snapshot to avoid data loss.

TTL config in upsertConfig

```
"upsertConfig": {
 "mode": "FULL",
 "TtlUnit": "DAYS",
 "TtlValue": "1",
 "isSkipLateEvents": true
}
```

Sometimes users cannot guarantee the order of the message, so we use `isSkipLateEvent` to define the behavior whether or not to process events that have already passed TTL.

Limitations

TTLs with data bootstrap backfill

Upsert table backfill support was released in 2022 H1, this feature can be used for first time backfill for long retention tables, which improves slow backfill from Kafka due to the throughput limit. For some use cases, TTL support and bootstrap are both required.

However, we assume each primary key is only active for a short period of time, and it will be removed from upsert metadata after TTL. With backfill for upsert tables, inactive primary keys will be consumed again and might cause duplicate records.

Bootstrap backfill requires to disabled TTL and restart servers.

TTLs with data correction backfill

Another common use case of backfill is data correction backfill. It enabled users to backfill data for a given period of table, mostly to fix potential data corruption or missing data.

To avoid duplicate data while running the fixed time backfill, we need to recover the upsert data structure by scanning all historical data. When recovering the historical data, the upsert metadata will be increased significantly. Since capacity estimation might be based on the primary keys TTL, recovering the historical upsert metadata might cause server crashes. An approach to solve the problem is scaling up the cluster and disable TTL during backfill, the steps are as follows.

- Disable TTL.
- Add more machines in the cluster to avoid server crash by recovering historical upsert metadata. Rebalance the table.
- Start backfill, scan all data to recover historical metadata.
- Once backfill finished, enable ttl and prune old metadata.
- Return hosts from the tenant and rebalance.

However this approach has the following constraints.

- Scanning all historical data to recover upsert metadata will be slow.
- **When expanding the cluster and rebalancing the table, snapshots stored on local segments will be lost.**

Reference

- Pinot upsert revisit Design
- Pinot partial upsert Design
- Pinot upsert optimization discussion
- When to persist snapshot discussion when to persist snapshot