# Segment Merge Service

Pinot-Minion Task

## Motivation

Inside Pinot cluster, there are large number of small segments with very few documents inside. These kind of segments are usually generated from real-time use case with low consuming rate or hourly / daily pushed offline use case with very few records.

To aggregate the same amount of documents,  processing small segments are much more costly than processing large segments. Besides that, store small segments will consume more memory and disk space both from inefficient dictionary encoding and extra metadata storage.

Segment Merge Service is designed to merge multiple small segments into large segments and make segments serve the cluster better on both performance and storage aspects.

Segment merge jobs could be expensive on both CPU and memory, so Pinot-Minion cluster is ideal for serving this task.

## Requirements

### Multiple Features

We should support the following features for segment merge:

1. Concatenate: simply concatenate all records
2. Roll-up: roll-up on time column
3. Sort: sort on a specified column
4. Startree: enable startree index

### Data Consistency

During the segment merge, the data should be consistent, in the sense that: no doubled results, no missing results, the results should be returned either from merged segments or original segments, but not both.

# Run Periodically

The segment merge job should run periodically, picking up small segments and merge them into appropriately-sized segments.

# Job Triggering

## Merge Strategy

To interact correctly with partition based routing or other routing mechanisms where segments must follow some convention (e.g. column1 values must have the same hash key), we might need multiple merge strategies, which only select the segments that are mergeable.

If we need some special naming convention for the merged segment, the logic could also be implemented inside the merge strategy to decide the merged segment name.

## Triggering Conditions

We should support multiple triggering conditions for segment merge:

1. Merge all: merge all existing small segments
2. Number of segments: merge small segments only if the number of segments exceed a threshold
3. Number of total documents: merge small segments only if the number of total documents exceed a threshold

## Multilevel Merge

For roll-up case, we might want multilevel merge. For example, the original segments' time unit is SECONDS, the first level merge roll it up to HOURS, the second level merge roll it up to DAYS, etc.

## Fault Tolerant

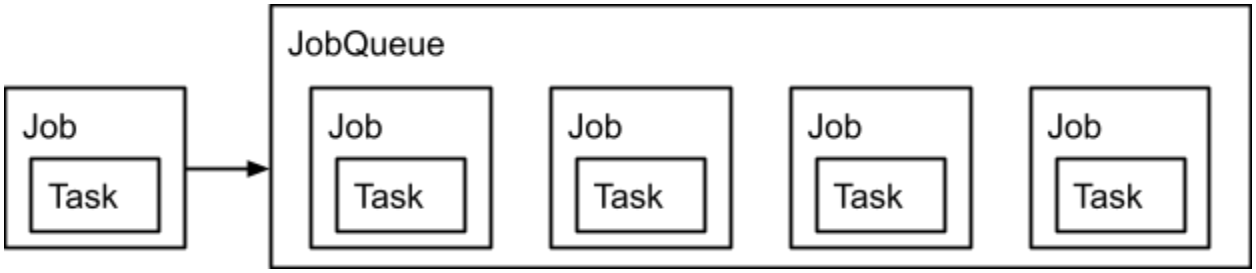
In case of failures or slow jobs, we should make sure:

1. We only merge each segment at most once, if the segment is already in a running merge job, ignore it in the new merge job.
2. If one job failed (after retry times defined), do not submit the identical job again.

# Solution

We propose the following design for the solution.

## Map to Pinot-Minion




For segment merge service, a Segment Merge JobQueue should be good as a starting point.

4

The arguments for the job should be as followings:

1. Target table name
2. List of segment download urls
3. Segment merge config (merge type, roll-up config, sort column, startree config, merged segment name etc.)
4. Merged segment upload address

## Segment Merge Config

We will put a new field “segmentMergeConfig” into the table config.

```
{
   "mergeType": "CONCATENATE" / "ROLL_UP",
   "mergeStrategy": “TIME_BASED” / "PARTITION_BASED",
   "rollUpConfig":
   {
      "preAggregateType":
      {
         "min_metric": "MIN",
         "max_metric": "MAX",
         "hll_metric": "HLL_8"
      },
      "settings":
      [
         {
            "rollUpFrom": "SECONDS",
            "rollUpTo": "DAYS",
            "minNumSegments": 20,
            "minNumTotalDocs": 100000
         },
         {
            "rollUpFrom": "DAYS",
            "rollUpTo": "MONTHS",
            "minNumSegments": 30,
            "minNumTotalDocs": 500000
         }
      ]
   },
   "enableStarTree": true,
   "starTreeConfig": null,
   "minNumSegments": 10,
   "minNumTotalDocs": 100000,
   "oldSegmentCleanup": true
}
```

The segment merge config contains the following arguments:

1. mergeType: CONCATENATE or ROLLUP

2. mergeStrategy (optional): merge strategy to determine segments to merge and merged segment name (by default, “TIME_BASED”)
3. rollUpConfig (for ROLL_UP merge type, optional):
    a. preAggregateType (optional): map from metric column to pre-aggregate type (by default, “SUM”)
    b. settings: list of roll-up settings (for multilevel merge)
        i. rollUpFrom: time unit to roll-up from
        ii. rollUpTo: time unit to roll-up to
        iii. minNumSegments (optional): override the global setting
        iv. minNumTotalDocs (optional): override the global setting
4. enableStarTree (optional): whether to enable startree index (by default, false)
5. starTreeConfig (optional): startree config
6. minNumSegments (optional): number of segments threshold to trigger merge (by default, merge all unmerged segments)
7. minNumTotalDocs (optional): number of total x threshold to trigger merge (by default, merge all unmerged segments)
8. oldSegmentCleanup: whether to clean up the old segment in the segment merge task generator. (by default, it’s set to true)

Besides these arguments, we will pick up the “sortedColumn” from “tableIndexConfig” and sort the merged segment accordingly.

## Segment Metadata

Add the following new fields into the merged segment metadata for segment merge control:

1. segment.merge.cover: [list of segments (original segments as well as all lower level merged segments) the merged segment covered]
2. segment.merge.rollup.unit: roll-up time unit

Field “segment.merge.cover” will be used for the following purpose:

1. Clean up the segments that already merged into the new segment
2. Help broker decide where to route the query

3. Prevent client from refreshing the segments that already merged into the new segment because that will cause doubled results.

Field “segment.merge.rollup.unit” will be used for multilevel roll-up use case. For original segments, we will use “segment.time.unit” as the roll-up unit.

## Time Based Merged Segment Name

We need a naming convention for merged segments to prevent conflict. Because segment merge only works on time-series table, we will use the range of the time column to construct the merged segment name: merged_<tableName>_<startTime>_<endTime>_<timestamp>. We need a timestamp because multiple merged segments might have the same time column range.

## Broker Enhanced Routing

After merged segment got pushed into the cluster, but before segments already merged into the new segment got removed, both old segments and merged segments co-locate inside the cluster, which might cause doubled results. To solve this, we need to enhance the broker routing algorithm to pick only one copy of the data (merged segments preferred).

To achieve that, whenever we route a query, we check all segments metadata for the target table, remove the segments inside the “segment.merge.cover” list. All the lasting segments are the final target segments.

## Old Segments Clean Up

We need to clean up the old segments when the merged segment gets loaded into the cluster. We can use the “segment.merge.cover” field in the merged segment metadata to help clean up all old segment.

There are two options for cleaning up the old segments:

1. Get a separate thread to periodically check the segments metadata and delete old segments.
2. Implement the old segments clean up inside Segment Merge Task Generator. Before generating the tasks, first clean up the old segments.

Comparing to the first option, the second option can save more system resources and automatically check for only tables with segment merge enabled.

# Segment Merge Task Generator

SegmentMergeTaskGenerator will implement the PinotMinionTaskGenerator interface and be registered into the Task Manager (refer to Pinot-Minion Service design doc).

SegmentMergeTaskGenerator is responsible for:

1. Clean up old segments: check the “segment.merge.cover” segment metadata inside the merged segments (need to check if merged segment already shows up in External View to make sure it’s loaded on the server side), delete all the segments covered by the merged segments.
2. Generate new tasks: check the remaining segments and task status, generate segment merge tasks if the triggering conditions defined in segment merge config are met.

One thing we need to bear in mind is that each segment can only be merged into one segment to prevent doubled results.

## Clean Up Old Segments

To clean up old segments, we need to fetch all segments metadata and External View for the given table, and do a 2 passes scan.

First pass: for each merged segment which showed up in External View, get it’s “segment.merge.cover” list from metadata, and put all covered segments into a set.

Second pass: Delete all segments that exist in the covered segments set.

## Generate New Tasks:

To generate new tasks, we need to use all remaining segments metadata (after cleaning up old segments) and all Segment Merge task status and config.

For NOT_STARTED / IN_PROGRESS / COMPLETED tasks, we need to skip merging the segments that already exist inside the task to ensure each segment will only be merged once. For FAILED tasks (after configurable retries), try not to submit an identical job again (optional).

Merge Strategy should be pluggable to select segments that are mergeable, and decide the merged segment name. By default, we use TIME_BASED merge strategy where all segments are mergeable.

Merge Strategy Interface

```
interface MergeStrategy {
  // Use segments metadata and triggering condition to generate a list of merge tasks
  // The first element inside the pair is merged segment name, and the second
  // element is the segments to be merged
  public List<Pair<String, List<String>>> generateMergeTasks();
}
```

# Segment Merge Task Executor

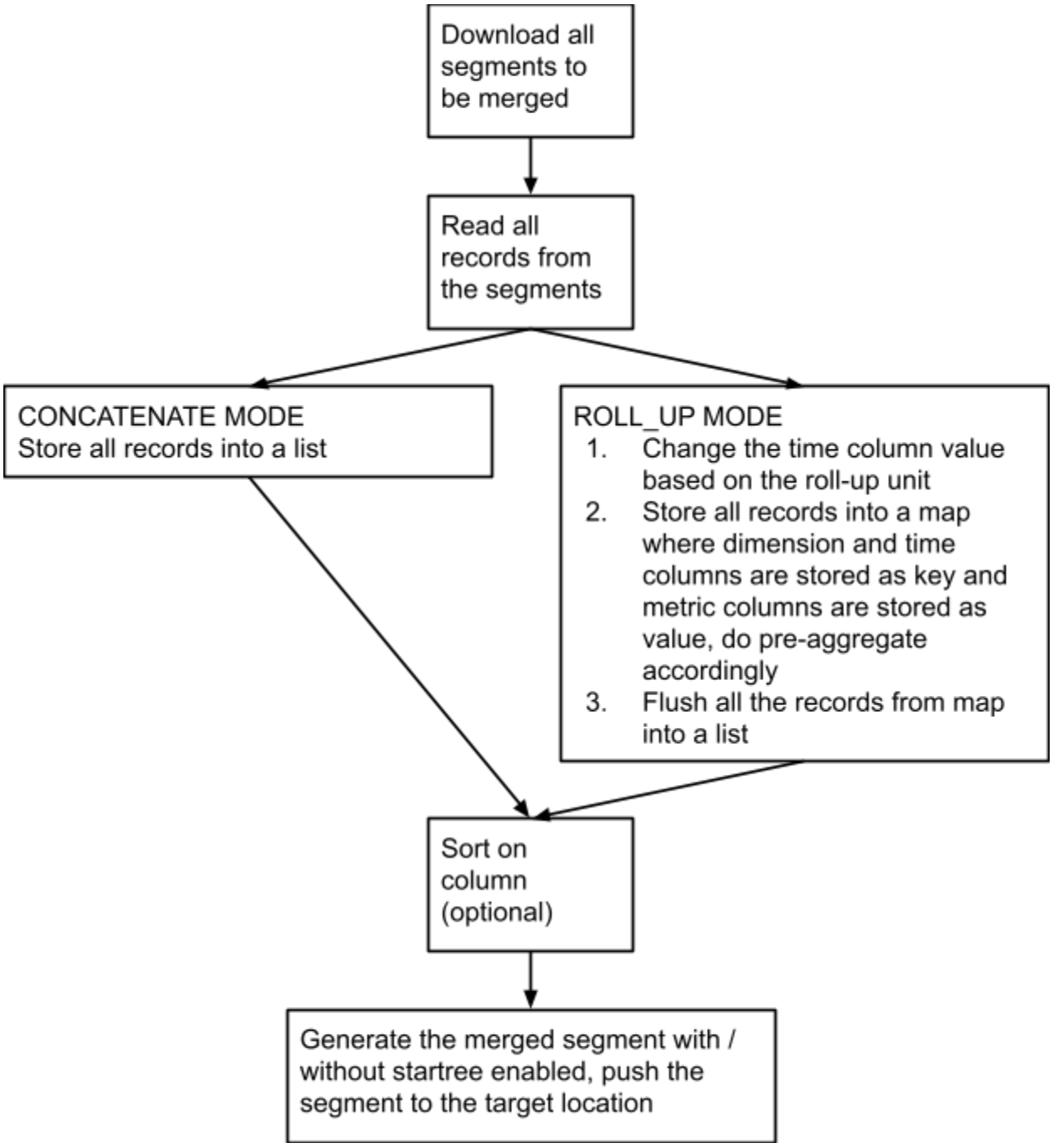
As the starting point, for simplicity, we will read all records from segments to be merged, and put them in memory before generate the merged segments.

## Arguments

The segment merge job need to take the following arguments:

1. Target table name
2. List of segment download urls
3. Merged segment name
4. Segment push address
5. Merge type
6. Roll-up config (optional)
7. Sort column (optional)
8. Enable startree index (optional)
9. Startree index config (optional)

Steps




## Restrictions

Segment merge only works on APPEND use case, because for REFRESH use case, segment might be replaced and the merged segments will be inconsistent with original segments.

Do not support backfill old segment (replace a segment with a new one with the same name) without downtime. To backfill old segments, we need to first delete the merged segment, and upload all segments used to generate the merged segment.

Segment merge only works on OFFLINE use case and LLC use case, because HLC segments are different among different servers and merge them doesn't make a lot of sense.

Target table should be time-series table, and roll-up only works on time column.

To make roll-up work properly, all time columns should be time since epoch.