Minion Segment Conversion Framework

Motivation

Pinot Minion service is designed to serve a lot of different costly operations and reduce the workload on other components.

The most common use case for Minion service is to convert segments (e.g. index creation, segment merge & rollup, data purge etc). Since segment conversion tasks share a lot of similar logics, we can make this as a general framework where a user can easily plug-in their segment processing logic without worrying about writing complex steps.

Requirements

Given a list of segments and a configuration, we need to handle the following types of segment conversion.

1. Row transformation
2. Row aggregation on x columns
3. Sort on sorted column
4. Index creation (startree, inverted index)
5. Support N -> M segment merge (or split)

Proposed Design

Current Status

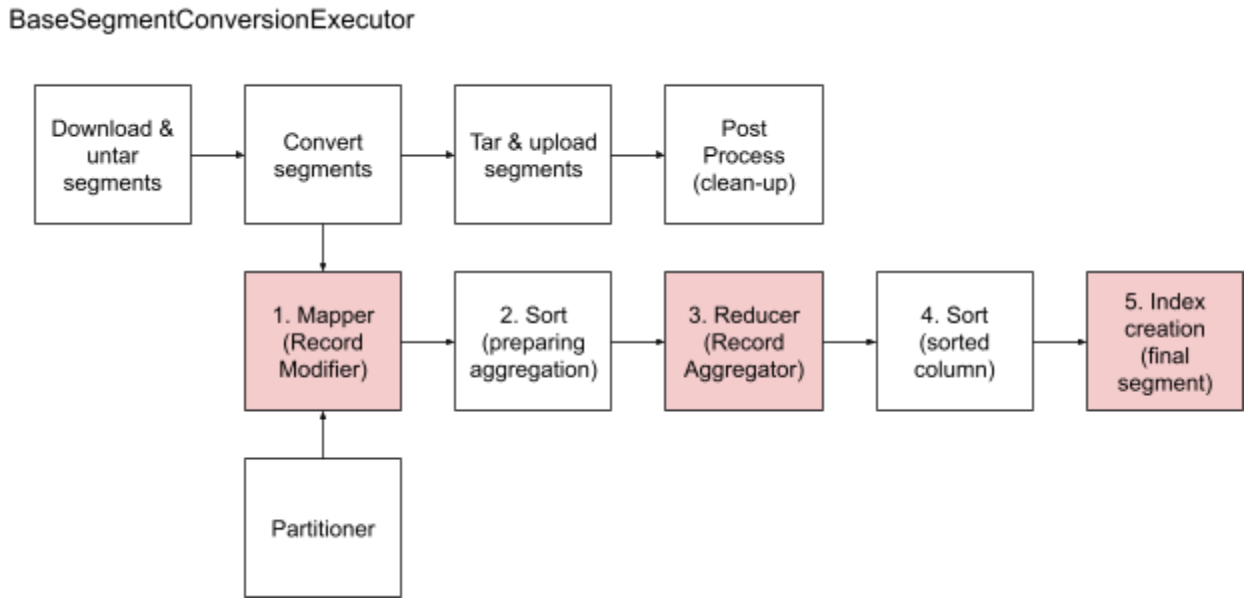
The current BaseSegmentConversionExecutor implementation is written for **1 -> 1 segment conversion** with **abstract convert()** method.

```
public abstract class BaseSegmentConversionExecutor extends BaseTaskExecutor {
  // put the logic for converting segment
  abstract convert(config, inputIndexDir, outputDir);

  // put the logic for modifying SegmentZKMetadata
  abstract getSegmentZKMetadataCustomMapModifier();

  public void executeTask(config) {
    // 1. download & untar segments
    // 2. convert segment
    // 3. tar & upload segment
```

```
        // 4. clean up
      }
    }
```

Proposed Design

We need to improve the current BaseSegmentConversionExecutor to support N-> M segment conversion. Also, we will subdivide **“convert()”** into 5 different stages. Different from distributed map-reduce job like in Hadoop, the following segment converting execution happens in a single node.



1. Mapper: all the records will be transformed based on the given record transformer. For each record, we will use partitioner to decide the partition.
2. Preparation for Aggregation: the input segments for Stage 3 will be sorted based on the provided aggregation column.
3. Reducer: records will be grouped by the columns specified in the sort order in Stage 2. Input for each RecordAggregator.aggregate() function is a list of rows from the same group. Input rows will be aggregated into one row based on the given record aggregator implementation.
4. Sorted Column: input for Stage 5 will be sorted based on the provided sort column
5. Index Creation: inverted index/ startree will be generated based on the index config.

Additional Notes

- Each red colored stage will generate **m** files on disk.
- Stage 1 (Mapper) is required while (2,3), 4, 5 are pluggable (2,3 should go together). We enforce Stage 1 in case where we need to merge the segment (Sorting in stage 2 requires the global view of the multiple segments). We can add the optimization later to avoid unnecessary segment creation when we process 1 -> 1 segment transformation.

- A user need to implement RecordPartitioner, RecordTransformer, RecordAggregator and specify sort orders for Stage 2, 4 in order to achieve the final segment.

Implementation Details

Sorter

```
public interface SegmentSorter {
   int[] getSortedDocIds(final List<String> sortOrder);
}

public PinotSegmentReader implements RecordReader {
   void setSortedDocIds(int[] sortedDocIds);
}
```

RecordTransformer / RecordPartitioner/ MapperRecordReader

```
interface RecordTransformer {
   GenericRow transformRecord(GenericRow row);
}

interface RecordPartitioner {
   int getPartitionForRecord(GenericRow row, int numPartition);
}

public MapperRecordReader implements RecordReader {
   MultiplePinotSegmentReader _recordReader;
   RecordTransformer _recordTransformer;
   RecordPartitioner _recordPartitioner;
   int _numPartition;
   int _partition;
   public GenericRow next(GenericRow reuse) {
      while (true){
         reuse = _recordReader.next(reuse);
         // Read the partition x only
         if (_recordPartitioner.getPartitionForRecord(reuse, _numPartition) == _partition) {
            break;
         }
      }
      reuse = _recordTransformer.transformRecord(reuse);
      return reuse;
   }
}
```

RecordAggregator / ReducerRecordReader

```
interface RecordAggregator {
   GenericRow aggregateRecord(List<GenericRow> rows);
```

```
}

public ReducerRecordReader implements RecordReader {
   PinotSegmentRecordReader _recordReader;
   RecordAggregator _recordAggregator;

   public GenericRow next(GenericRow reuse) {
      while(haveSameColumnValues(prevRow, currentRow)) {
         // add the row to the list
      }
      return _recordAggregator.aggregate(rows);
   }
}
```

SegmentConverter (Putting Everything Together)

A user needs to implement “RecordTransformer” and “RecordAggregator” and specify the sort order for both stage 2, 4.

```
public SegmentConverter {
   List<File> inputFiles;
   Config config;

   public List<File> convertSegment() {
      List<File> resultFiles = new ArrayList<>();
      for (int currentPartition = 0; currentPartition < numPartition; currentPartition++) {
         // 1. Mapper stage
         MapperRecordReader mapReader =
               new MapperRecordReader(inputFiles, recordTransformer, currentPartition);
        File mapperOutputSegment = buildSegment(mapReader);

         // 2. Sort
         Sorter sorter = new Sorter(mapperOutputSegment);
         int[] getSortedDocIds = sorter.getSortedDocIds(sortOrderForAggregation);

         // 3. Reduce
         RecuderRecordReader reduceReader =
            new RecuderRecordReader(mapperOutputSegment, recordAggregator);
         reducerReader.setSortedDocIds(sortedDocIds);
         File reduceOutputSegment = buildSegment(config, reduceReader);

         // 4. Sort
         Sorter sorter = new Sorter(reduceOutputSegment);
         int[] getSortedDocIds = sorter.getSortedDocIds(sortedColum);

         // 5. Final Index Creation
         PinotSegmentReader reader = new PinotSegmentReader(reduceOutputSegment)
         reader.setSortedDocIds(sortedDocIds);
```

```
            File finalSegment = buildSegmentWithIndexConfig(config, reader);

            resultFiles.add(finalSegment);
        }
        return resultFiles;
    }
}
```

Example Minion Segment Converting Job

In this section, we introduce the way to represent different possible segment converting jobs with the proposed framework.

Add Sorted column / Creating Inverted / Startree Index

- Stage 1 - Mapper
    - SegmentTransformer.transformRecord(): return the original row
- Stage 4 - Sort (optional): sort on sorted column
- Stage 5 - Index creation (optional): create inverted index, startree based on table config

Segment Purge

- Stage 1 - Mapper
    - SegmentTransformer.transformRecord(): return null for rows to be purged
- Stage 4 - Sort (optional): sort on sorted column
- Stage 5 - Index creation (optional): create inverted index, startree based on table config

Segment Concatenate

- Stage 1 - Mapper
    - SegmentTransformer.transformRecord(): return the original row
- Stage 4 - Sort (optional): sort on sorted column
- Stage 5 - Index creation (optional): create inverted index, startree based on table config

Segment Rollup (No Time Granularity Change)

- Stage 1 - Mapper
    - SegmentTransformer.transformRecord(): return the original row
- Stage 2 - Sort
    - sortOrder: all dimensions + time column
- Stage 3 - Aggregation
    - SegmentAggregator.aggregateRecords(): for each metric column, apply configured aggregation function

- Stage 4 - Sort (optional): sort on sorted column
- Stage 5 - Index creation (optional): create inverted index, startree based on table config

Segment Rollup (With Time Granularity Change)

- Stage 1 - Mapper
    - SegmentTransformer.transformRecord(): transform time column using dateTimeTransformer
- Stage 2 - Sort
    - sortOrder: all dimensions + time column
- Stage 3 - Aggregation
    - SegmentAggregator.aggregateRecords(): for each metric column, apply configured aggregation function
- Stage 4 - Sort (optional): sort on sorted column
- Stage 5 - Index creation (optional): create inverted index, startree based on table config

Possible Improvements

- Sorting performance can be improved if used different data format for intermediate files (e.g. Startree generation uses row based storage format)
- Aggregation stage can use a lot of heap memory if the group size is large (we need to keep all the records for each group in memory).
- We can remove unnecessary intermediate file creation for certain situations.
- Support for changing schema (adding derived columns, removing columns)