Segment Compaction for Upsert Enabled Tables in Apache Pinot

By: Robert Zych

August 4th, 2023 • 4 min read

I'm happy to share that my 1st feature contribution to the Apache Pinot project (Segment compaction for upsert enabled real-time tables) was merged recently! In this post, I will briefly discuss the problem segment compaction addresses, how to configure it, and what it looks like in action. If you're unfamiliar with Pinot's Upsert features, I recommend reviewing Full Upserts in Pinot to get started and Stream Ingestion with Upsert for more information.

Context and Configuration

As Pinot's Upsert stores all versions of the record ingested into immutable segments on disk, older records unnecessarily consume valuable storage space when they're no longer used in query results. Pinot's Segment Compaction reclaims this valuable storage space by introducing a periodic process that replaces the completed segments with compacted segments which only contain the latest version of the records. I recommend reviewing the Minion documentation if you're unfamiliar with Pinot's ability to run periodic processes.

With task scheduling enabled and an available Minion, you can configure segment compaction by adding the following to your table's config.

```
"task": {
  "taskTypeConfigsMap": {
    "UpsertCompactionTask": {
      "schedule": "0 */5 * ? * *",
      "bufferTimePeriod": "7d",
      "invalidRecordsThresholdPercent": "30",
      "invalidRecordsThresholdCount": "100000"
    }
  }
}
```

All the configs above (excluding schedule) determine which completed segments are selected for compaction.

bufferTimePeriod is the amount of time that has elapsed since the segment was consuming. In the example above, this has been set to “7d” which means that any segment that was completed over 7 days ago may be eligible for compaction. However, if you want to ensure that segments are compacted without any additional delay this config can be set to “0d”.

invalidRecordsThresholdPercent is a limit to the amount of older records allowed in the completed segment represented as a percentage of the total number of records in the segment (i.e. old records / total records). In the example above, this has been set to “30” which means that if more than 30% of the records in the completed segment are old, then the segment may be selected for compaction. As segment compaction is an expensive operation, it is not recommended to set this config (or invalidRecordsThresholdCount) too close to 1. This config is optional on the condition that invalidRecordsThresholdCount has been set and can be used in conjunction with invalidRecordsThresholdCount.

invalidRecordsThresholdCount is also a limit similar to invalidRecordsThresholdPercent, but allows you to express the threshold as a record count. In the example above, this has been set to “100000” which means that if the segment contains more than 100K records then it may be selected for compaction.

Example Use Case

I’ve created a data set that includes 24M records. The data set contains 240K unique keys that have each been duplicated 100 times.

 alt

After ingesting the data set there are 6 segments (5 completed segments + 1 consuming segment) with a total estimated size of 22.8MB. Submitting the query “set skipUpsert=true; select count(*) from transcript_upsert” before compaction produces the following query result.

 alt

After the compaction tasks are complete, the Minion Task Manager UI reports the following.

 alt

Segment compaction generates a task for each segment to be compacted. 5 tasks were generated in this case because 90% of the records (3.6–4.5M records) are old in all 5 of the completed segments, therefore exceeding the configured thresholds. If a completed segment only contains old records, it is deleted immediately and a task isn’t generated to compact it.

 alt

Submitting the query again we now see that count matches the set of 240K unique keys.

 alt

Once compaction has completed and the original segments have been replaced with their compacted counterparts we see that the total number of segments remained the same, but the total estimated size dropped to only 2.77MB! Since compaction can yield

very small segments, one improvement would be to merge smaller segments into larger ones as this would improve query latency.

Conclusion

In this brief overview of Segment Compaction I covered the problem it addresses, how you can configure it, and demonstrated its ability to reclaim storage space. I'd like to thank Ankit Sultana, Seunghyun Lee, and especially Jackie Jiang for their feedback and support throughout the design and development stages. If you have any questions or feedback, I'm available on the Apache Pinot Slack.