package com.twitter.search.earlybird.archive;

import java.io.IOException;

import java.util.ArrayList;

import java.util.Calendar;

import java.util.Collections;

import java.util.Comparator;

import java.util.Date;

import java.util.List;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Preconditions;

import com.google.common.base.Predicate;

import com.google.common.collect.Lists;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.search.common.schema.thriftjava.ThriftIndexingEvent;

import com.twitter.search.common.util.io.MergingSortedRecordReader;

import com.twitter.search.common.util.io.recordreader.RecordReader;

import com.twitter.search.earlybird.config.TierConfig;

import com.twitter.search.earlybird.document.DocumentFactory;

import com.twitter.search.earlybird.document.ThriftIndexingEventDocumentFactory;

import com.twitter.search.earlybird.document.TweetDocument;

/\*\*

\* Responsible for taking a number of daily status batches and partitioning them into time slices

\* which will be used to build segments.

\*

\* We try to put at most N number of tweets into a time slice.

\*/

public class ArchiveTimeSlicer {

private static final Logger LOG = LoggerFactory.getLogger(ArchiveTimeSlicer.class);

private static final Comparator<TweetDocument> ASCENDING =

(o1, o2) -> Long.compare(o1.getTweetID(), o2.getTweetID());

private static final Comparator<TweetDocument> DESCENDING =

(o1, o2) -> Long.compare(o2.getTweetID(), o1.getTweetID());

// Represents a number of daily batches which will go into a segment.

public static final class ArchiveTimeSlice {

private Date startDate;

private Date endDate;

private int statusCount;

private final DailyStatusBatches directory;

private final ArchiveEarlybirdIndexConfig earlybirdIndexConfig;

// This list is always ordered from oldest day, to the newest day.

// For the on-disk archive, we reverse the days in getTweetReaders().

private final List<DailyStatusBatch> batches = Lists.newArrayList();

private ArchiveTimeSlice(DailyStatusBatches directory,

ArchiveEarlybirdIndexConfig earlybirdIndexConfig) {

this.directory = directory;

this.earlybirdIndexConfig = earlybirdIndexConfig;

}

public Date getEndDate() {

return endDate;

}

public int getStatusCount() {

return statusCount;

}

public int getNumHashPartitions() {

return batches.isEmpty() ? 0 : batches.get(0).getNumHashPartitions();

}

/\*\*

\* Returns a reader for reading tweets from this timeslice.

\*

\* @param archiveSegment The segment to which the timeslice belongs.

\* @param documentFactory The ThriftIndexingEvent to TweetDocument converter.

\* @param filter A filter that determines what dates should be read.

\*/

public RecordReader<TweetDocument> getStatusReader(

ArchiveSegment archiveSegment,

DocumentFactory<ThriftIndexingEvent> documentFactory,

Predicate<Date> filter) throws IOException {

// We no longer support ThriftStatus based document factories.

Preconditions.checkState(documentFactory instanceof ThriftIndexingEventDocumentFactory);

final int hashPartitionID = archiveSegment.getHashPartitionID();

List<RecordReader<TweetDocument>> readers = new ArrayList<>(batches.size());

List<DailyStatusBatch> orderedForReading = orderBatchesForReading(batches);

LOG.info("Creating new status reader for hashPartition: "

+ hashPartitionID + " timeslice: " + getDescription());

for (DailyStatusBatch batch : orderedForReading) {

if (filter.apply(batch.getDate())) {

LOG.info("Adding reader for " + batch.getDate() + " " + getDescription());

PartitionedBatch partitionedBatch = batch.getPartition(hashPartitionID);

// Don't even try to create a reader if the partition is empty.

// There does not seem to be any problem in production now, but HDFS FileSystem's javadoc

// does indicate that listStatus() is allowed to throw a FileNotFoundException if the

// partition does not exist. This check makes the code more robust against future

// HDFS FileSystem implementation changes.

if (partitionedBatch.getStatusCount() > 0) {

RecordReader<TweetDocument> tweetReaders = partitionedBatch.getTweetReaders(

archiveSegment,

directory.getStatusPathToUseForDay(batch.getDate()),

documentFactory);

readers.add(tweetReaders);

}

} else {

LOG.info("Filtered reader for " + batch.getDate() + " " + getDescription());

}

}

LOG.info("Creating reader for timeslice: " + getDescription()

+ " with " + readers.size() + " readers");

return new MergingSortedRecordReader<TweetDocument>(getMergingComparator(), readers);

}

private List<DailyStatusBatch> orderBatchesForReading(List<DailyStatusBatch> orderedBatches) {

// For the index formats using stock lucene, we want the most recent days to be indexed first.

// In the twitter in-memory optimized indexes, older tweets will be added first, and

// optimization will reverse the documents to make most recent tweets be first.

return this.earlybirdIndexConfig.isUsingLIFODocumentOrdering()

? orderedBatches : Lists.reverse(orderedBatches);

}

private Comparator<TweetDocument> getMergingComparator() {

// We always want to retrieve larger tweet ids first.

// LIFO means that the smaller ids get inserted first --> ASCENDING order.

// FIFO would mean that we want to first insert the larger ids --> DESCENDING order.

return this.earlybirdIndexConfig.isUsingLIFODocumentOrdering()

? ASCENDING : DESCENDING;

}

/\*\*

\* Returns the smallest indexed tweet ID in this timeslice for the given partition.

\*

\* @param hashPartitionID The partition.

\*/

public long getMinStatusID(int hashPartitionID) {

if (batches.isEmpty()) {

return 0;

}

for (int i = 0; i < batches.size(); i++) {

long minStatusID = batches.get(i).getPartition(hashPartitionID).getMinStatusID();

if (minStatusID != DailyStatusBatch.EMPTY\_BATCH\_STATUS\_ID) {

return minStatusID;

}

}

return 0;

}

/\*\*

\* Returns the highest indexed tweet ID in this timeslice for the given partition.

\*

\* @param hashPartitionID The partition.

\*/

public long getMaxStatusID(int hashPartitionID) {

if (batches.isEmpty()) {

return Long.MAX\_VALUE;

}

for (int i = batches.size() - 1; i >= 0; i--) {

long maxStatusID = batches.get(i).getPartition(hashPartitionID).getMaxStatusID();

if (maxStatusID != DailyStatusBatch.EMPTY\_BATCH\_STATUS\_ID) {

return maxStatusID;

}

}

return Long.MAX\_VALUE;

}

/\*\*

\* Returns a string with some information for this timeslice.

\*/

public String getDescription() {

StringBuilder builder = new StringBuilder();

builder.append("TimeSlice[start date=");

builder.append(DailyStatusBatches.DATE\_FORMAT.format(startDate));

builder.append(", end date=");

builder.append(DailyStatusBatches.DATE\_FORMAT.format(endDate));

builder.append(", status count=");

builder.append(statusCount);

builder.append(", days count=");

builder.append(batches.size());

builder.append("]");

return builder.toString();

}

}

private final int maxSegmentSize;

private final DailyStatusBatches dailyStatusBatches;

private final Date tierStartDate;

private final Date tierEndDate;

private final ArchiveEarlybirdIndexConfig earlybirdIndexConfig;

private List<ArchiveTimeSlice> lastCachedTimeslices = null;

public ArchiveTimeSlicer(int maxSegmentSize,

DailyStatusBatches dailyStatusBatches,

ArchiveEarlybirdIndexConfig earlybirdIndexConfig) {

this(maxSegmentSize, dailyStatusBatches, TierConfig.DEFAULT\_TIER\_START\_DATE,

TierConfig.DEFAULT\_TIER\_END\_DATE, earlybirdIndexConfig);

}

public ArchiveTimeSlicer(int maxSegmentSize,

DailyStatusBatches dailyStatusBatches,

Date tierStartDate,

Date tierEndDate,

ArchiveEarlybirdIndexConfig earlybirdIndexConfig) {

this.maxSegmentSize = maxSegmentSize;

this.dailyStatusBatches = dailyStatusBatches;

this.tierStartDate = tierStartDate;

this.tierEndDate = tierEndDate;

this.earlybirdIndexConfig = earlybirdIndexConfig;

}

private boolean cacheIsValid() throws IOException {

return lastCachedTimeslices != null

&& !lastCachedTimeslices.isEmpty()

&& cacheIsValid(lastCachedTimeslices.get(lastCachedTimeslices.size() - 1).endDate);

}

private boolean cacheIsValid(Date lastDate) throws IOException {

if (lastCachedTimeslices == null || lastCachedTimeslices.isEmpty()) {

return false;

}

// Check if we have a daily batch newer than the last batch used for the newest timeslice.

Calendar cal = Calendar.getInstance();

cal.setTime(lastDate);

cal.add(Calendar.DATE, 1);

Date nextDate = cal.getTime();

boolean foundBatch = dailyStatusBatches.hasValidBatchForDay(nextDate);

LOG.info("Checking cache: Looked for valid batch for day {}. Found: {}",

DailyStatusBatches.DATE\_FORMAT.format(nextDate), foundBatch);

return !foundBatch;

}

private boolean timesliceIsFull(ArchiveTimeSlice timeSlice, DailyStatusBatch batch) {

return timeSlice.statusCount + batch.getMaxPerPartitionStatusCount() > maxSegmentSize;

}

private void doTimeSlicing() throws IOException {

dailyStatusBatches.refresh();

lastCachedTimeslices = Lists.newArrayList();

ArchiveTimeSlice currentTimeSlice = null;

// Iterate over each day and add it to the current timeslice, until it gets full.

for (DailyStatusBatch batch : dailyStatusBatches.getStatusBatches()) {

if (!batch.isValid()) {

LOG.warn("Skipping hole: " + batch.getDate());

continue;

}

if (currentTimeSlice == null || timesliceIsFull(currentTimeSlice, batch)) {

if (currentTimeSlice != null) {

LOG.info("Filled timeslice: " + currentTimeSlice.getDescription());

}

currentTimeSlice = new ArchiveTimeSlice(dailyStatusBatches, earlybirdIndexConfig);

currentTimeSlice.startDate = batch.getDate();

lastCachedTimeslices.add(currentTimeSlice);

}

currentTimeSlice.endDate = batch.getDate();

currentTimeSlice.statusCount += batch.getMaxPerPartitionStatusCount();

currentTimeSlice.batches.add(batch);

}

LOG.info("Last timeslice: {}", currentTimeSlice.getDescription());

LOG.info("Done with time slicing. Number of timeslices: {}",

lastCachedTimeslices.size());

}

/\*\*

\* Returns all timeslices for this earlybird.

\*/

public List<ArchiveTimeSlice> getTimeSlices() throws IOException {

if (cacheIsValid()) {

return lastCachedTimeslices;

}

LOG.info("Cache is outdated. Loading new daily batches now...");

doTimeSlicing();

return lastCachedTimeslices != null ? Collections.unmodifiableList(lastCachedTimeslices) : null;

}

/\*\*

\* Return the timeslices that overlap the tier start/end date ranges if they are specified

\*/

public List<ArchiveTimeSlice> getTimeSlicesInTierRange() throws IOException {

List<ArchiveTimeSlice> timeSlices = getTimeSlices();

if (tierStartDate == TierConfig.DEFAULT\_TIER\_START\_DATE

&& tierEndDate == TierConfig.DEFAULT\_TIER\_END\_DATE) {

return timeSlices;

}

List<ArchiveTimeSlice> filteredTimeSlice = Lists.newArrayList();

for (ArchiveTimeSlice timeSlice : timeSlices) {

if (timeSlice.startDate.before(tierEndDate) && !timeSlice.endDate.before(tierStartDate)) {

filteredTimeSlice.add(timeSlice);

}

}

return filteredTimeSlice;

}

@VisibleForTesting

protected DailyStatusBatches getDailyStatusBatches() {

return dailyStatusBatches;

}

}