package com.twitter.search.earlybird.partition.freshstartup;

import java.io.IOException;

import java.time.Duration;

import java.util.ArrayList;

import java.util.Optional;

import com.google.common.base.Preconditions;

import com.google.common.base.Stopwatch;

import com.google.common.collect.ImmutableList;

import com.google.common.collect.ImmutableMap;

import org.apache.kafka.clients.consumer.ConsumerRecord;

import org.apache.kafka.clients.consumer.ConsumerRecords;

import org.apache.kafka.clients.consumer.KafkaConsumer;

import org.apache.kafka.clients.consumer.OffsetAndTimestamp;

import org.apache.kafka.common.TopicPartition;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.search.common.indexing.thriftjava.ThriftVersionedEvents;

import com.twitter.search.earlybird.factory.EarlybirdKafkaConsumersFactory;

import com.twitter.search.earlybird.partition.IndexingResultCounts;

import com.twitter.search.earlybird.partition.SegmentInfo;

import com.twitter.search.earlybird.partition.SegmentManager;

import com.twitter.search.earlybird.partition.SegmentWriter;

/\*\*

\* Responsible for indexing the tweets and updates that need to be applied to a single segment

\* before it gets optimized and then optimizing the segment (except if it's the last one).

\*

\* After that, no more tweets are added to the segment and the rest of the updates are added

\* in PostOptimizationUpdatesIndexer.

\*/

class PreOptimizationSegmentIndexer {

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

private SegmentBuildInfo segmentBuildInfo;

private final ArrayList<SegmentBuildInfo> segmentBuildInfos;

private SegmentManager segmentManager;

private final TopicPartition tweetTopic;

private final TopicPartition updateTopic;

private final EarlybirdKafkaConsumersFactory earlybirdKafkaConsumersFactory;

private final long lateTweetBuffer;

public PreOptimizationSegmentIndexer(

SegmentBuildInfo segmentBuildInfo,

ArrayList<SegmentBuildInfo> segmentBuildInfos,

SegmentManager segmentManager,

TopicPartition tweetTopic,

TopicPartition updateTopic,

EarlybirdKafkaConsumersFactory earlybirdKafkaConsumersFactory,

long lateTweetBuffer) {

this.segmentBuildInfo = segmentBuildInfo;

this.segmentBuildInfos = segmentBuildInfos;

this.segmentManager = segmentManager;

this.tweetTopic = tweetTopic;

this.updateTopic = updateTopic;

this.earlybirdKafkaConsumersFactory = earlybirdKafkaConsumersFactory;

this.lateTweetBuffer = lateTweetBuffer;

}

SegmentInfo runIndexing() throws IOException {

LOG.info(String.format("Starting segment building for segment %d. "

+ "Tweet offset range [ %,d, %,d ]",

segmentBuildInfo.getIndex(),

segmentBuildInfo.getTweetStartOffset(),

segmentBuildInfo.getTweetEndOffset()));

Optional<Long> firstTweetIdInNextSegment = Optional.empty();

int index = segmentBuildInfo.getIndex();

if (index + 1 < segmentBuildInfos.size()) {

firstTweetIdInNextSegment = Optional.of(

segmentBuildInfos.get(index + 1).getStartTweetId());

}

// Index tweets.

SegmentTweetsIndexingResult tweetIndexingResult = indexSegmentTweetsFromStream(

tweetTopic,

String.format("tweet\_consumer\_for\_segment\_%d", segmentBuildInfo.getIndex()),

firstTweetIdInNextSegment

);

// Index updates.

KafkaOffsetPair updatesIndexingOffsets = findUpdateStreamOffsetRange(tweetIndexingResult);

String updatesConsumerClientId =

String.format("update\_consumer\_for\_segment\_%d", segmentBuildInfo.getIndex());

LOG.info(String.format("Consumer: %s :: Tweets start time: %d, end time: %d ==> "

+ "Updates start offset: %,d, end offset: %,d",

updatesConsumerClientId,

tweetIndexingResult.getMinRecordTimestampMs(),

tweetIndexingResult.getMaxRecordTimestampMs(),

updatesIndexingOffsets.getBeginOffset(),

updatesIndexingOffsets.getEndOffset()));

indexUpdatesFromStream(

updateTopic,

updatesConsumerClientId,

updatesIndexingOffsets.getBeginOffset(),

updatesIndexingOffsets.getEndOffset(),

tweetIndexingResult.getSegmentWriter()

);

if (segmentBuildInfo.isLastSegment()) {

/\*

\* We don't optimize the last segment for a few reasons:

\*

\* 1. We might have tweets coming next in the stream, which are supposed to end

\* up in this segment.

\*

\* 2. We might have updates coming next in the stream, which need to be applied to

\* this segment before it's optimized.

\*

\* So the segment is kept unoptimized and later we take care of setting up things

\* so that PartitionWriter and the tweet create/update handlers can start correctly.

\*/

LOG.info("Not optimizing the last segment ({})", segmentBuildInfo.getIndex());

} else {

Stopwatch optimizationStopwatch = Stopwatch.createStarted();

try {

LOG.info("Starting to optimize segment: {}", segmentBuildInfo.getIndex());

tweetIndexingResult.getSegmentWriter().getSegmentInfo()

.getIndexSegment().optimizeIndexes();

} finally {

LOG.info("Optimization of segment {} finished in {}.",

segmentBuildInfo.getIndex(), optimizationStopwatch);

}

}

segmentBuildInfo.setUpdateKafkaOffsetPair(updatesIndexingOffsets);

segmentBuildInfo.setMaxIndexedTweetId(tweetIndexingResult.getMaxIndexedTweetId());

segmentBuildInfo.setSegmentWriter(tweetIndexingResult.getSegmentWriter());

return tweetIndexingResult.getSegmentWriter().getSegmentInfo();

}

private SegmentTweetsIndexingResult indexSegmentTweetsFromStream(

TopicPartition topicPartition,

String consumerClientId,

Optional<Long> firstTweetIdInNextSegment) throws IOException {

long startOffset = segmentBuildInfo.getTweetStartOffset();

long endOffset = segmentBuildInfo.getTweetEndOffset();

long marginSize = lateTweetBuffer / 2;

boolean isFirstSegment = segmentBuildInfo.getIndex() == 0;

long startReadingAtOffset = startOffset;

if (!isFirstSegment) {

startReadingAtOffset -= marginSize;

} else {

LOG.info("Not moving start offset backwards for segment {}.", segmentBuildInfo.getIndex());

}

long endReadingAtOffset = endOffset;

if (firstTweetIdInNextSegment.isPresent()) {

endReadingAtOffset += marginSize;

} else {

LOG.info("Not moving end offset forwards for segment {}.", segmentBuildInfo.getIndex());

}

KafkaConsumer<Long, ThriftVersionedEvents> tweetsKafkaConsumer =

makeKafkaConsumerForIndexing(consumerClientId,

topicPartition, startReadingAtOffset);

boolean done = false;

long minIndexedTimestampMs = Long.MAX\_VALUE;

long maxIndexedTimestampMs = Long.MIN\_VALUE;

int indexedEvents = 0;

Stopwatch stopwatch = Stopwatch.createStarted();

LOG.info("Creating segment writer for timeslice ID {}.", segmentBuildInfo.getStartTweetId());

SegmentWriter segmentWriter = segmentManager.createSegmentWriter(

segmentBuildInfo.getStartTweetId());

/\*

\* We don't have a guarantee that tweets come in sorted order, so when we're building segment

\* X', we try to pick some tweets from the previous and next ranges we're going to index.

\*

\* We also ignore tweets in the beginning and the end of our tweets range, which are picked

\* by the previous or following segment.

\*

\* Segment X Segment X' Segment X''

\* -------------- o ----------------------------------------- o ---------------

\* [~~~~~] ^ [~~~~~] [~~~~~] | [~~~~~]

\* | | | | | |

\* front margin | front padding (size K) back padding | back margin

\* | |

\* segment boundary at offset B' (1) B''

\*

\* (1) This is at a predetermined tweet offset / tweet id.

\*

\* For segment X', we start to read tweets at offset B'-K and finish reading

\* tweets at offset B''+K. K is a constant.

\*

\* For middle segments X'

\* ======================

\* We move some tweets from the front margin and back margin into segment X'.

\* Some tweets from the front and back padding are ignored, as they are moved

\* into the previous and next segments.

\*

\* For the first segment

\* =====================

\* No front margin, no front padding. We just read from the beginning offset

\* and insert everything.

\*

\* For the last segment

\* ====================

\* No back margin, no back padding. We just read until the end.

\*/

SkippedPickedCounter frontMargin = new SkippedPickedCounter("front margin");

SkippedPickedCounter backMargin = new SkippedPickedCounter("back margin");

SkippedPickedCounter frontPadding = new SkippedPickedCounter("front padding");

SkippedPickedCounter backPadding = new SkippedPickedCounter("back padding");

SkippedPickedCounter regular = new SkippedPickedCounter("regular");

int totalRead = 0;

long maxIndexedTweetId = -1;

Stopwatch pollTimer = Stopwatch.createUnstarted();

Stopwatch indexTimer = Stopwatch.createUnstarted();

do {

// This can cause an exception, See P33896

pollTimer.start();

ConsumerRecords<Long, ThriftVersionedEvents> records =

tweetsKafkaConsumer.poll(Duration.ofSeconds(1));

pollTimer.stop();

indexTimer.start();

for (ConsumerRecord<Long, ThriftVersionedEvents> record : records) {

// Done reading?

if (record.offset() >= endReadingAtOffset) {

done = true;

}

ThriftVersionedEvents tve = record.value();

boolean indexTweet = false;

SkippedPickedCounter skippedPickedCounter;

if (record.offset() < segmentBuildInfo.getTweetStartOffset()) {

// Front margin.

skippedPickedCounter = frontMargin;

if (tve.getId() > segmentBuildInfo.getStartTweetId()) {

indexTweet = true;

}

} else if (record.offset() > segmentBuildInfo.getTweetEndOffset()) {

// Back margin.

skippedPickedCounter = backMargin;

if (firstTweetIdInNextSegment.isPresent()

&& tve.getId() < firstTweetIdInNextSegment.get()) {

indexTweet = true;

}

} else if (record.offset() < segmentBuildInfo.getTweetStartOffset() + marginSize) {

// Front padding.

skippedPickedCounter = frontPadding;

if (tve.getId() >= segmentBuildInfo.getStartTweetId()) {

indexTweet = true;

}

} else if (firstTweetIdInNextSegment.isPresent()

&& record.offset() > segmentBuildInfo.getTweetEndOffset() - marginSize) {

// Back padding.

skippedPickedCounter = backPadding;

if (tve.getId() < firstTweetIdInNextSegment.get()) {

indexTweet = true;

}

} else {

skippedPickedCounter = regular;

// These we just pick. A tweet that came very late can end up in the wrong

// segment, but it's better for it to be present in a segment than dropped.

indexTweet = true;

}

if (indexTweet) {

skippedPickedCounter.incrementPicked();

segmentWriter.indexThriftVersionedEvents(tve);

maxIndexedTweetId = Math.max(maxIndexedTweetId, tve.getId());

indexedEvents++;

// Note that records don't necessarily have increasing timestamps.

// Why? The timestamps whatever timestamp we picked when creating the record

// in ingesters and there are many ingesters.

minIndexedTimestampMs = Math.min(minIndexedTimestampMs, record.timestamp());

maxIndexedTimestampMs = Math.max(maxIndexedTimestampMs, record.timestamp());

} else {

skippedPickedCounter.incrementSkipped();

}

totalRead++;

if (record.offset() >= endReadingAtOffset) {

break;

}

}

indexTimer.stop();

} while (!done);

tweetsKafkaConsumer.close();

SegmentTweetsIndexingResult result = new SegmentTweetsIndexingResult(

minIndexedTimestampMs, maxIndexedTimestampMs, maxIndexedTweetId, segmentWriter);

LOG.info("Finished indexing {} tweets for {} in {}. Read {} tweets. Result: {}."

+ " Time polling: {}, Time indexing: {}.",

indexedEvents, consumerClientId, stopwatch, totalRead, result,

pollTimer, indexTimer);

// In normal conditions, expect to pick just a few in front and in the back.

LOG.info("SkippedPicked ({}) -- {}, {}, {}, {}, {}",

consumerClientId, frontMargin, frontPadding, backPadding, backMargin, regular);

return result;

}

/\*\*

\* After indexing all the tweets for a segment, index updates that need to be applied before

\* the segment is optimized.

\*

\* This is required because some updates (URL updates, cards and Named Entities) can only be

\* applied to an unoptimized segment. Luckily, all of these updates should arrive close to when

\* the Tweet is created.

\*/

private KafkaOffsetPair findUpdateStreamOffsetRange(

SegmentTweetsIndexingResult tweetsIndexingResult) {

KafkaConsumer<Long, ThriftVersionedEvents> offsetsConsumer =

earlybirdKafkaConsumersFactory.createKafkaConsumer(

"consumer\_for\_update\_offsets\_" + segmentBuildInfo.getIndex());

// Start one minute before the first indexed tweet. One minute is excessive, but

// we need to start a bit earlier in case the first tweet we indexed came in

// later than some of its updates.

long updatesStartOffset = offsetForTime(offsetsConsumer, updateTopic,

tweetsIndexingResult.getMinRecordTimestampMs() - Duration.ofMinutes(1).toMillis());

// Two cases:

//

// 1. If we're not indexing the last segment, end 10 minutes after the last tweet. So for

// example if we resolve an url in a tweet 3 minutes after the tweet is published,

// we'll apply that update before the segment is optimized. 10 minutes is a bit too

// much, but that doesn't matter a whole lot, since we're indexing about ~10 hours of

// updates.

//

// 2. If we're indexing the last segment, end a bit before the last indexed tweet. We might

// have incoming tweets that are a bit late. In fresh startup, we don't have a mechanism

// to store these tweets to be applied when the tweet arrives, as in TweetUpdateHandler,

// so just stop a bit earlier and let TweetCreateHandler and TweetUpdateHandler deal with

// that.

long millisAdjust;

if (segmentBuildInfo.getIndex() == segmentBuildInfos.size() - 1) {

millisAdjust = -Duration.ofMinutes(1).toMillis();

} else {

millisAdjust = Duration.ofMinutes(10).toMillis();

}

long updatesEndOffset = offsetForTime(offsetsConsumer, updateTopic,

tweetsIndexingResult.getMaxRecordTimestampMs() + millisAdjust);

offsetsConsumer.close();

return new KafkaOffsetPair(updatesStartOffset, updatesEndOffset);

}

/\*\*

\* Get the earliest offset with a timestamp >= $timestamp.

\*

\* The guarantee we get is that if we start reading from here on, we will get

\* every single message that came in with a timestamp >= $timestamp.

\*/

private long offsetForTime(KafkaConsumer<Long, ThriftVersionedEvents> kafkaConsumer,

TopicPartition partition,

long timestamp) {

Preconditions.checkNotNull(kafkaConsumer);

Preconditions.checkNotNull(partition);

OffsetAndTimestamp offsetAndTimestamp = kafkaConsumer

.offsetsForTimes(ImmutableMap.of(partition, timestamp))

.get(partition);

if (offsetAndTimestamp == null) {

return -1;

} else {

return offsetAndTimestamp.offset();

}

}

private void indexUpdatesFromStream(

TopicPartition topicPartition,

String consumerClientId,

long startOffset,

long endOffset,

SegmentWriter segmentWriter) throws IOException {

KafkaConsumer<Long, ThriftVersionedEvents> kafkaConsumer =

makeKafkaConsumerForIndexing(consumerClientId, topicPartition, startOffset);

// Index TVEs.

boolean done = false;

Stopwatch pollTimer = Stopwatch.createUnstarted();

Stopwatch indexTimer = Stopwatch.createUnstarted();

SkippedPickedCounter updatesSkippedPicked = new SkippedPickedCounter("streamed\_updates");

IndexingResultCounts indexingResultCounts = new IndexingResultCounts();

long segmentTimesliceId = segmentWriter.getSegmentInfo().getTimeSliceID();

Stopwatch totalTime = Stopwatch.createStarted();

do {

pollTimer.start();

ConsumerRecords<Long, ThriftVersionedEvents> records =

kafkaConsumer.poll(Duration.ofSeconds(1));

pollTimer.stop();

indexTimer.start();

for (ConsumerRecord<Long, ThriftVersionedEvents> record : records) {

if (record.value().getId() < segmentTimesliceId) {

// Doesn't apply to this segment, can be skipped instead of skipping it

// inside the more costly segmentWriter.indexThriftVersionedEvents call.

updatesSkippedPicked.incrementSkipped();

} else {

if (record.offset() >= endOffset) {

done = true;

}

updatesSkippedPicked.incrementPicked();

indexingResultCounts.countResult(

segmentWriter.indexThriftVersionedEvents(record.value()));

}

if (record.offset() >= endOffset) {

break;

}

}

indexTimer.stop();

} while (!done);

// Note that there'll be a decent amount of failed retryable updates. Since we index

// updates in a range that's a bit wider, they can't be applied here.

LOG.info("Client: {}, Finished indexing updates: {}. "

+ "Times -- total: {}. polling: {}, indexing: {}. Indexing result counts: {}",

consumerClientId, updatesSkippedPicked,

totalTime, pollTimer, indexTimer, indexingResultCounts);

}

/\*\*

\* Make a consumer that reads from a single partition, starting at some offset.

\*/

private KafkaConsumer<Long, ThriftVersionedEvents> makeKafkaConsumerForIndexing(

String consumerClientId,

TopicPartition topicPartition,

long offset) {

KafkaConsumer<Long, ThriftVersionedEvents> kafkaConsumer =

earlybirdKafkaConsumersFactory.createKafkaConsumer(consumerClientId);

kafkaConsumer.assign(ImmutableList.of(topicPartition));

kafkaConsumer.seek(topicPartition, offset);

LOG.info("Indexing TVEs. Kafka consumer: {}", consumerClientId);

return kafkaConsumer;

}

}