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

import java.io.IOException;

import java.time.Duration;

import java.util.ArrayList;

import java.util.HashSet;

import java.util.List;

import java.util.Map;

import java.util.Set;

import com.google.common.base.Stopwatch;

import com.google.common.base.Verify;

import com.google.common.collect.ImmutableList;

import com.google.common.collect.ImmutableMap;

import com.google.common.collect.Lists;

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.apache.kafka.common.errors.ApiException;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

import static com.twitter.search.common.util.LogFormatUtil.formatInt;

import com.twitter.search.common.util.GCUtil;

import com.twitter.common.util.Clock;

import com.twitter.search.common.util.LogFormatUtil;

import com.twitter.search.earlybird.common.NonPagingAssert;

import com.twitter.search.earlybird.common.config.EarlybirdConfig;

import com.twitter.search.earlybird.exception.CriticalExceptionHandler;

import com.twitter.search.earlybird.exception.EarlybirdStartupException;

import com.twitter.search.earlybird.exception.WrappedKafkaApiException;

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

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

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

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

import com.twitter.search.earlybird.util.ParallelUtil;

/\*\*

\* Bootstraps an index by indexing tweets and updates in parallel.

\*

\* DEVELOPMENT

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

\*

\* 1. In earlybird-search.yml, set the following values in the "production" section:

\* - max\_segment\_size to 200000

\* - late\_tweet\_buffer to 10000

\*

\* 2. In KafkaStartup, don't load the index, replace the .loadIndex call as instructed

\* in the file.

\*

\* 3. In the aurora configs, set serving\_timeslices to a low number (like 5) for staging.

\*/

public class FreshStartupHandler {

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

private static final NonPagingAssert BUILDING\_FEWER\_THAN\_SPECIFIED\_SEGMENTS =

new NonPagingAssert("building\_fewer\_than\_specified\_segments");

private final Clock clock;

private final TopicPartition tweetTopic;

private final TopicPartition updateTopic;

private final SegmentManager segmentManager;

private final int maxSegmentSize;

private final int lateTweetBuffer;

private final EarlybirdKafkaConsumersFactory earlybirdKafkaConsumersFactory;

private final CriticalExceptionHandler criticalExceptionHandler;

public FreshStartupHandler(

Clock clock,

EarlybirdKafkaConsumersFactory earlybirdKafkaConsumersFactory,

TopicPartition tweetTopic,

TopicPartition updateTopic,

SegmentManager segmentManager,

int maxSegmentSize,

int lateTweetBuffer,

CriticalExceptionHandler criticalExceptionHandler

) {

this.clock = clock;

this.earlybirdKafkaConsumersFactory = earlybirdKafkaConsumersFactory;

this.tweetTopic = tweetTopic;

this.updateTopic = updateTopic;

this.segmentManager = segmentManager;

this.maxSegmentSize = maxSegmentSize;

this.criticalExceptionHandler = criticalExceptionHandler;

this.lateTweetBuffer = lateTweetBuffer;

}

/\*\*

\* Don't index in parallel, just pass some time back that the EarlybirdKafkaConsumer

\* can start indexing from.

\*/

public EarlybirdIndex indexFromScratch() {

long indexTimePeriod = Duration.ofHours(

EarlybirdConfig.getInt("index\_from\_scratch\_hours", 12)

).toMillis();

return runIndexFromScratch(indexTimePeriod);

}

public EarlybirdIndex fastIndexFromScratchForDevelopment() {

LOG.info("Running fast index from scratch...");

return runIndexFromScratch(Duration.ofMinutes(10).toMillis());

}

private EarlybirdIndex runIndexFromScratch(long indexTimePeriodMs) {

KafkaConsumer<Long, ThriftVersionedEvents> consumerForFindingOffsets =

earlybirdKafkaConsumersFactory.createKafkaConsumer("consumer\_for\_offsets");

long timestamp = clock.nowMillis() - indexTimePeriodMs;

Map<TopicPartition, OffsetAndTimestamp> offsets;

try {

offsets = consumerForFindingOffsets

.offsetsForTimes(ImmutableMap.of(tweetTopic, timestamp, updateTopic, timestamp));

} catch (ApiException kafkaApiException) {

throw new WrappedKafkaApiException(kafkaApiException);

}

return new EarlybirdIndex(

Lists.newArrayList(),

offsets.get(tweetTopic).offset(),

offsets.get(updateTopic).offset());

}

/\*\*

\* Index Tweets and updates from scratch, without relying on a serialized index in HDFS.

\*

\* This function indexes the segments in parallel, limiting the number of segments that

\* are currently indexed, due to memory limitations. That's followed by another pass to index

\* some updates - see the implementation for more details.

\*

\* The index this function outputs contains N segments, where the first N-1 are optimized and

\* the last one is not.

\*/

public EarlybirdIndex parallelIndexFromScratch() throws Exception {

Stopwatch parallelIndexStopwatch = Stopwatch.createStarted();

LOG.info("Starting parallel fresh startup.");

LOG.info("Max segment size: {}", maxSegmentSize);

LOG.info("Late tweet buffer size: {}", lateTweetBuffer);

// Once we finish fresh startup and proceed to indexing from the streams, we'll immediately

// start a new segment, since the output of the fresh startup is full segments.

//

// That's why we index max\_segments-1 segments here instead of indexing max\_segments segments

// and discarding the first one later.

int numSegments = segmentManager.getMaxEnabledSegments() - 1;

LOG.info("Number of segments to build: {}", numSegments);

// Find end offsets.

KafkaOffsetPair tweetsOffsetRange = findOffsetRangeForTweetsKafkaTopic();

ArrayList<SegmentBuildInfo> segmentBuildInfos = makeSegmentBuildInfos(

numSegments, tweetsOffsetRange);

segmentManager.logState("Before starting fresh startup");

// Index tweets and events.

Stopwatch initialIndexStopwatch = Stopwatch.createStarted();

// We index at most `MAX\_PARALLEL\_INDEXED` (MPI) segments at the same time. If we need to

// produce 20 segments here, we'd need memory for MPI unoptimized and 20-MPI optimized segments.

//

// For back of envelope calculations you can assume optimized segments take ~6GB and unoptimized

// ones ~12GB.

final int MAX\_PARALLEL\_INDEXED = 8;

List<SegmentInfo> segmentInfos = ParallelUtil.parmap(

"fresh-startup",

MAX\_PARALLEL\_INDEXED,

segmentBuildInfo -> indexTweetsAndUpdatesForSegment(segmentBuildInfo, segmentBuildInfos),

segmentBuildInfos

);

LOG.info("Finished indexing tweets and updates in {}", initialIndexStopwatch);

PostOptimizationUpdatesIndexer postOptimizationUpdatesIndexer =

new PostOptimizationUpdatesIndexer(

segmentBuildInfos,

earlybirdKafkaConsumersFactory,

updateTopic);

postOptimizationUpdatesIndexer.indexRestOfUpdates();

// Finished indexing tweets and updates.

LOG.info("Segment build infos after we're done:");

for (SegmentBuildInfo segmentBuildInfo : segmentBuildInfos) {

segmentBuildInfo.logState();

}

segmentManager.logState("After finishing fresh startup");

LOG.info("Collected {} segment infos", segmentInfos.size());

LOG.info("Segment names:");

for (SegmentInfo segmentInfo : segmentInfos) {

LOG.info(segmentInfo.getSegmentName());

}

SegmentBuildInfo lastSegmentBuildInfo = segmentBuildInfos.get(segmentBuildInfos.size() - 1);

long finishedUpdatesAtOffset = lastSegmentBuildInfo.getUpdateKafkaOffsetPair().getEndOffset();

long maxIndexedTweetId = lastSegmentBuildInfo.getMaxIndexedTweetId();

LOG.info("Max indexed tweet id: {}", maxIndexedTweetId);

LOG.info("Parallel startup finished in {}", parallelIndexStopwatch);

// verifyConstructedIndex(segmentBuildInfos);

// Run a GC to free up some memory after the fresh startup.

GCUtil.runGC();

logMemoryStats();

return new EarlybirdIndex(

segmentInfos,

tweetsOffsetRange.getEndOffset() + 1,

finishedUpdatesAtOffset + 1,

maxIndexedTweetId

);

}

private void logMemoryStats() {

double toGB = 1024 \* 1024 \* 1024;

double totalMemoryGB = Runtime.getRuntime().totalMemory() / toGB;

double freeMemoryGB = Runtime.getRuntime().freeMemory() / toGB;

LOG.info("Memory stats: Total memory GB: {}, Free memory GB: {}",

totalMemoryGB, freeMemoryGB);

}

/\*\*

\* Prints statistics about the constructed index compared to all tweets in the

\* tweets stream.

\*

\* Only run this for testing and debugging purposes, never in prod environment.

\*/

private void verifyConstructedIndex(List<SegmentBuildInfo> segmentBuildInfos)

throws IOException {

LOG.info("Verifying constructed index...");

// Read every tweet from the offset range that we're constructing an index for.

KafkaConsumer<Long, ThriftVersionedEvents> tweetsKafkaConsumer =

earlybirdKafkaConsumersFactory.createKafkaConsumer("tweets\_verify");

try {

tweetsKafkaConsumer.assign(ImmutableList.of(tweetTopic));

tweetsKafkaConsumer.seek(tweetTopic, segmentBuildInfos.get(0).getTweetStartOffset());

} catch (ApiException apiException) {

throw new WrappedKafkaApiException(apiException);

}

long finalTweetOffset = segmentBuildInfos.get(segmentBuildInfos.size() - 1).getTweetEndOffset();

boolean done = false;

Set<Long> uniqueTweetIds = new HashSet<>();

long readTweetsCount = 0;

do {

for (ConsumerRecord<Long, ThriftVersionedEvents> record

: tweetsKafkaConsumer.poll(Duration.ofSeconds(1))) {

if (record.offset() > finalTweetOffset) {

done = true;

break;

}

readTweetsCount++;

uniqueTweetIds.add(record.value().getId());

}

} while (!done);

LOG.info("Total amount of read tweets: {}", formatInt(readTweetsCount));

// Might be less, due to duplicates.

LOG.info("Unique tweet ids : {}", LogFormatUtil.formatInt(uniqueTweetIds.size()));

int notFoundInIndex = 0;

for (Long tweetId : uniqueTweetIds) {

boolean found = false;

for (SegmentBuildInfo segmentBuildInfo : segmentBuildInfos) {

if (segmentBuildInfo.getSegmentWriter().hasTweet(tweetId)) {

found = true;

break;

}

}

if (!found) {

notFoundInIndex++;

}

}

LOG.info("Tweets not found in the index: {}", LogFormatUtil.formatInt(notFoundInIndex));

long totalIndexedTweets = 0;

for (SegmentBuildInfo segmentBuildInfo : segmentBuildInfos) {

SegmentInfo si = segmentBuildInfo.getSegmentWriter().getSegmentInfo();

totalIndexedTweets += si.getIndexStats().getStatusCount();

}

LOG.info("Total indexed tweets: {}", formatInt(totalIndexedTweets));

}

/\*\*

\* Find the end offsets for the tweets Kafka topic this partition is reading

\* from.

\*/

private KafkaOffsetPair findOffsetRangeForTweetsKafkaTopic() {

KafkaConsumer<Long, ThriftVersionedEvents> consumerForFindingOffsets =

earlybirdKafkaConsumersFactory.createKafkaConsumer("consumer\_for\_end\_offsets");

Map<TopicPartition, Long> endOffsets;

Map<TopicPartition, Long> beginningOffsets;

try {

endOffsets = consumerForFindingOffsets.endOffsets(ImmutableList.of(tweetTopic));

beginningOffsets = consumerForFindingOffsets.beginningOffsets(ImmutableList.of(tweetTopic));

} catch (ApiException kafkaApiException) {

throw new WrappedKafkaApiException(kafkaApiException);

} finally {

consumerForFindingOffsets.close();

}

long tweetsBeginningOffset = beginningOffsets.get(tweetTopic);

long tweetsEndOffset = endOffsets.get(tweetTopic);

LOG.info(String.format("Tweets beginning offset: %,d", tweetsBeginningOffset));

LOG.info(String.format("Tweets end offset: %,d", tweetsEndOffset));

LOG.info(String.format("Total amount of records in the stream: %,d",

tweetsEndOffset - tweetsBeginningOffset + 1));

return new KafkaOffsetPair(tweetsBeginningOffset, tweetsEndOffset);

}

/\*\*

\* For each segment, we know what offset it begins at. This function finds the tweet ids

\* for these offsets.

\*/

private void fillTweetIdsForSegmentStarts(List<SegmentBuildInfo> segmentBuildInfos)

throws EarlybirdStartupException {

KafkaConsumer<Long, ThriftVersionedEvents> consumerForTweetIds =

earlybirdKafkaConsumersFactory.createKafkaConsumer("consumer\_for\_tweet\_ids", 1);

consumerForTweetIds.assign(ImmutableList.of(tweetTopic));

// Find first tweet ids for each segment.

for (SegmentBuildInfo buildInfo : segmentBuildInfos) {

long tweetOffset = buildInfo.getTweetStartOffset();

ConsumerRecords<Long, ThriftVersionedEvents> records;

try {

consumerForTweetIds.seek(tweetTopic, tweetOffset);

records = consumerForTweetIds.poll(Duration.ofSeconds(1));

} catch (ApiException kafkaApiException) {

throw new WrappedKafkaApiException(kafkaApiException);

}

if (records.count() > 0) {

ConsumerRecord<Long, ThriftVersionedEvents> recordAtOffset = records.iterator().next();

if (recordAtOffset.offset() != tweetOffset) {

LOG.error(String.format("We were looking for offset %,d. Found a record at offset %,d",

tweetOffset, recordAtOffset.offset()));

}

buildInfo.setStartTweetId(recordAtOffset.value().getId());

} else {

throw new EarlybirdStartupException("Didn't get any tweets back for an offset");

}

}

// Check that something weird didn't happen where we end up with segment ids

// which are in non-incresing order.

// Goes from oldest to newest.

for (int i = 1; i < segmentBuildInfos.size(); i++) {

long startTweetId = segmentBuildInfos.get(i).getStartTweetId();

long prevStartTweetId = segmentBuildInfos.get(i - 1).getStartTweetId();

Verify.verify(prevStartTweetId < startTweetId);

}

}

/\*\*

\* Generate the offsets at which tweets begin and end for each segment that we want

\* to create.

\*/

private ArrayList<SegmentBuildInfo> makeSegmentBuildInfos(

int numSegments, KafkaOffsetPair tweetsOffsets) throws EarlybirdStartupException {

ArrayList<SegmentBuildInfo> segmentBuildInfos = new ArrayList<>();

// If we have 3 segments, the starting tweet offsets are:

// end-3N, end-2N, end-N

int segmentSize = maxSegmentSize - lateTweetBuffer;

LOG.info("Segment size: {}", segmentSize);

long tweetsInStream = tweetsOffsets.getEndOffset() - tweetsOffsets.getBeginOffset() + 1;

double numBuildableSegments = ((double) tweetsInStream) / segmentSize;

LOG.info("Number of segments we can build: {}", numBuildableSegments);

int numSegmentsToBuild = numSegments;

int numBuildableSegmentsInt = (int) numBuildableSegments;

if (numBuildableSegmentsInt < numSegmentsToBuild) {

// This can happen if we get a low amount of tweets such that the ~10 days of tweets stored in

// Kafka are not enough to build the specified number of segments.

LOG.warn("Building {} segments instead of the specified {} segments because there are not "

+ "enough tweets", numSegmentsToBuild, numSegments);

BUILDING\_FEWER\_THAN\_SPECIFIED\_SEGMENTS.assertFailed();

numSegmentsToBuild = numBuildableSegmentsInt;

}

for (int rewind = numSegmentsToBuild; rewind >= 1; rewind--) {

long tweetStartOffset = (tweetsOffsets.getEndOffset() + 1) - (rewind \* segmentSize);

long tweetEndOffset = tweetStartOffset + segmentSize - 1;

int index = segmentBuildInfos.size();

segmentBuildInfos.add(new SegmentBuildInfo(

tweetStartOffset,

tweetEndOffset,

index,

rewind == 1

));

}

Verify.verify(segmentBuildInfos.get(segmentBuildInfos.size() - 1)

.getTweetEndOffset() == tweetsOffsets.getEndOffset());

LOG.info("Filling start tweet ids ...");

fillTweetIdsForSegmentStarts(segmentBuildInfos);

return segmentBuildInfos;

}

private SegmentInfo indexTweetsAndUpdatesForSegment(

SegmentBuildInfo segmentBuildInfo,

ArrayList<SegmentBuildInfo> segmentBuildInfos) throws Exception {

PreOptimizationSegmentIndexer preOptimizationSegmentIndexer =

new PreOptimizationSegmentIndexer(

segmentBuildInfo,

segmentBuildInfos,

this.segmentManager,

this.tweetTopic,

this.updateTopic,

this.earlybirdKafkaConsumersFactory,

this.lateTweetBuffer

);

return preOptimizationSegmentIndexer.runIndexing();

}

}