package com.twitter.search.earlybird.partition;

import java.io.IOException;

import java.util.ArrayList;

import java.util.List;

import java.util.SortedMap;

import java.util.TreeMap;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

import com.twitter.search.common.metrics.SearchRateCounter;

import com.twitter.search.common.partitioning.snowflakeparser.SnowflakeIdParser;

/\*\*

\* This class handles incoming updates to Tweets in the index.

\*

\* Much of the logic deals with retries. It is very common to get an update before we have gotten

\* the Tweet that the update should be applied to. In this case, we queue the update for up to a

\* minute, so that we give the original Tweet the chance to be written to the index.

\*/

public class TweetUpdateHandler {

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

private static final Logger UPDATES\_ERRORS\_LOG =

LoggerFactory.getLogger(TweetUpdateHandler.class.getName() + ".UpdatesErrors");

private static final String STATS\_PREFIX = "tweet\_update\_handler\_";

private IndexingResultCounts indexingResultCounts;

private static final SearchRateCounter INCOMING\_EVENT =

SearchRateCounter.export(STATS\_PREFIX + "incoming\_event");

private static final SearchRateCounter QUEUED\_FOR\_RETRY =

SearchRateCounter.export(STATS\_PREFIX + "queued\_for\_retry");

private static final SearchRateCounter DROPPED\_OLD\_EVENT =

SearchRateCounter.export(STATS\_PREFIX + "dropped\_old\_event");

private static final SearchRateCounter DROPPED\_INCOMING\_EVENT =

SearchRateCounter.export(STATS\_PREFIX + "dropped\_incoming\_event");

private static final SearchRateCounter DROPPED\_CLEANUP\_EVENT =

SearchRateCounter.export(STATS\_PREFIX + "dropped\_cleanup\_event");

private static final SearchRateCounter DROPPED\_NOT\_RETRYABLE\_EVENT =

SearchRateCounter.export(STATS\_PREFIX + "dropped\_not\_retryable\_event");

private static final SearchRateCounter PICKED\_TO\_RETRY =

SearchRateCounter.export(STATS\_PREFIX + "picked\_to\_retry");

private static final SearchRateCounter INDEXED\_EVENT =

SearchRateCounter.export(STATS\_PREFIX + "indexed\_event");

private static final long RETRY\_TIME\_THRESHOLD\_MS = 60\_000; // one minute.

private final SortedMap<Long, List<ThriftVersionedEvents>> pendingUpdates = new TreeMap<>();

private final SegmentManager segmentManager;

/\*\*

\* At this time we cleaned all updates that are more than RETRY\_TIME\_THRESHOLD\_MS old.

\*/

private long lastCleanedUpdatesTime = 0;

/\*\*

\* The time of the most recent Tweet that we have applied an update for. We use this to

\* determine when we should give up on retrying an update, instead of using the system clock,

\* because we may be processing the stream from a long time ago if we are starting up or if

\* there is lag in the Kafka topics and we want to let each update get a fair shot at being

\* applied.

\*/

private long mostRecentUpdateTime = 0;

public TweetUpdateHandler(SegmentManager segmentManager) {

this.segmentManager = segmentManager;

this.indexingResultCounts = new IndexingResultCounts();

}

/\*\*

\* Index an update to a Tweet.

\*/

public void handleTweetUpdate(ThriftVersionedEvents tve, boolean isRetry) throws IOException {

if (!isRetry) {

INCOMING\_EVENT.increment();

}

long id = tve.getId();

mostRecentUpdateTime =

Math.max(SnowflakeIdParser.getTimestampFromTweetId(id), mostRecentUpdateTime);

cleanStaleUpdates();

ISegmentWriter writer = segmentManager.getSegmentWriterForID(id);

if (writer == null) {

if (segmentManager.getNumIndexedDocuments() == 0) {

// If we haven't indexed any tweets at all, then we shouldn't drop this update, because it

// might be applied to a Tweet we haven't indexed yet so queue it up for retry.

queueForRetry(id, tve);

} else {

DROPPED\_OLD\_EVENT.increment();

}

return;

}

SegmentWriter.Result result = writer.indexThriftVersionedEvents(tve);

indexingResultCounts.countResult(result);

if (result == ISegmentWriter.Result.FAILURE\_RETRYABLE) {

// If the tweet hasn't arrived yet.

queueForRetry(id, tve);

} else if (result == ISegmentWriter.Result.FAILURE\_NOT\_RETRYABLE) {

DROPPED\_NOT\_RETRYABLE\_EVENT.increment();

UPDATES\_ERRORS\_LOG.warn("Failed to apply update for tweetID {}: {}", id, tve);

} else if (result == ISegmentWriter.Result.SUCCESS) {

INDEXED\_EVENT.increment();

}

}

private void queueForRetry(long id, ThriftVersionedEvents tve) {

long ageMillis = mostRecentUpdateTime - SnowflakeIdParser.getTimestampFromTweetId(id);

if (ageMillis > RETRY\_TIME\_THRESHOLD\_MS) {

DROPPED\_INCOMING\_EVENT.increment();

UPDATES\_ERRORS\_LOG.warn(

"Giving up retrying update for tweetID {}: {} because the retry time has elapsed",

id, tve);

return;

}

pendingUpdates.computeIfAbsent(id, i -> new ArrayList<>()).add(tve);

QUEUED\_FOR\_RETRY.increment();

}

// Every time we have processed a minute's worth of updates, remove all pending updates that are

// more than a minute old, relative to the most recent Tweet we have seen.

private void cleanStaleUpdates() {

long oldUpdatesThreshold = mostRecentUpdateTime - RETRY\_TIME\_THRESHOLD\_MS;

if (lastCleanedUpdatesTime < oldUpdatesThreshold) {

SortedMap<Long, List<ThriftVersionedEvents>> droppedUpdates = pendingUpdates

.headMap(SnowflakeIdParser.generateValidStatusId(oldUpdatesThreshold, 0));

for (List<ThriftVersionedEvents> events : droppedUpdates.values()) {

for (ThriftVersionedEvents event : events) {

UPDATES\_ERRORS\_LOG.warn(

"Giving up retrying update for tweetID {}: {} because the retry time has elapsed",

event.getId(), event);

}

DROPPED\_CLEANUP\_EVENT.increment(events.size());

}

droppedUpdates.clear();

lastCleanedUpdatesTime = mostRecentUpdateTime;

}

}

/\*\*

\* After we successfully indexed tweetID, if we have any pending updates for that tweetID, try to

\* apply them again.

\*/

public void retryPendingUpdates(long tweetID) throws IOException {

if (pendingUpdates.containsKey(tweetID)) {

for (ThriftVersionedEvents update : pendingUpdates.remove(tweetID)) {

PICKED\_TO\_RETRY.increment();

handleTweetUpdate(update, true);

}

}

}

void logState() {

LOG.info("TweetUpdateHandler:");

LOG.info(String.format(" tweets sent for indexing: %,d",

indexingResultCounts.getIndexingCalls()));

LOG.info(String.format(" non-retriable failure: %,d",

indexingResultCounts.getFailureNotRetriable()));

LOG.info(String.format(" retriable failure: %,d",

indexingResultCounts.getFailureRetriable()));

LOG.info(String.format(" successfully indexed: %,d",

indexingResultCounts.getIndexingSuccess()));

LOG.info(String.format(" queued for retry: %,d", QUEUED\_FOR\_RETRY.getCount()));

LOG.info(String.format(" dropped old events: %,d", DROPPED\_OLD\_EVENT.getCount()));

LOG.info(String.format(" dropped incoming events: %,d", DROPPED\_INCOMING\_EVENT.getCount()));

LOG.info(String.format(" dropped cleanup events: %,d", DROPPED\_CLEANUP\_EVENT.getCount()));

LOG.info(String.format(" picked events to retry: %,d", PICKED\_TO\_RETRY.getCount()));

}

}