package com.twitter.search.earlybird.partition;

import java.io.Closeable;

import java.time.Duration;

import java.util.Map;

import java.util.concurrent.atomic.AtomicBoolean;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Preconditions;

import com.google.common.base.Stopwatch;

import com.google.common.collect.ImmutableList;

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

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

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 com.twitter.search.common.metrics.SearchCounter;

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

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

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

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

import com.twitter.search.earlybird.EarlybirdStatus;

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

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

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

import com.twitter.search.earlybird.thrift.EarlybirdStatusCode;

/\*\*

\* Reads TVEs from Kafka and writes them to a PartitionWriter.

\*/

public class EarlybirdKafkaConsumer implements Closeable {

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

private static final Duration POLL\_TIMEOUT = Duration.ofSeconds(1);

private static final String STATS\_PREFIX = "earlybird\_kafka\_consumer\_";

// See SEARCH-31827

private static final SearchCounter INGESTING\_DONE =

SearchCounter.export(STATS\_PREFIX + "ingesting\_done");

private static final SearchRateCounter POLL\_LOOP\_EXCEPTIONS =

SearchRateCounter.export(STATS\_PREFIX + "poll\_loop\_exceptions");

private static final SearchRateCounter FLUSHING\_EXCEPTIONS =

SearchRateCounter.export(STATS\_PREFIX + "flushing\_exceptions");

private static final SearchTimerStats TIMED\_POLLS =

SearchTimerStats.export(STATS\_PREFIX + "timed\_polls");

private static final SearchTimerStats TIMED\_INDEX\_EVENTS =

SearchTimerStats.export(STATS\_PREFIX + "timed\_index\_events");

private final AtomicBoolean running = new AtomicBoolean(true);

private final BalancingKafkaConsumer balancingKafkaConsumer;

private final PartitionWriter partitionWriter;

protected final TopicPartition tweetTopic;

protected final TopicPartition updateTopic;

private final KafkaConsumer<Long, ThriftVersionedEvents> underlyingKafkaConsumer;

private final CriticalExceptionHandler criticalExceptionHandler;

private final EarlybirdIndexFlusher earlybirdIndexFlusher;

private final SearchIndexingMetricSet searchIndexingMetricSet;

private boolean finishedIngestUntilCurrent;

private final CaughtUpMonitor indexCaughtUpMonitor;

protected class ConsumeBatchResult {

private boolean isCaughtUp;

private long readRecordsCount;

public ConsumeBatchResult(boolean isCaughtUp, long readRecordsCount) {

this.isCaughtUp = isCaughtUp;

this.readRecordsCount = readRecordsCount;

}

public boolean isCaughtUp() {

return isCaughtUp;

}

public long getReadRecordsCount() {

return readRecordsCount;

}

}

public EarlybirdKafkaConsumer(

KafkaConsumer<Long, ThriftVersionedEvents> underlyingKafkaConsumer,

SearchIndexingMetricSet searchIndexingMetricSet,

CriticalExceptionHandler criticalExceptionHandler,

PartitionWriter partitionWriter,

TopicPartition tweetTopic,

TopicPartition updateTopic,

EarlybirdIndexFlusher earlybirdIndexFlusher,

CaughtUpMonitor kafkaIndexCaughtUpMonitor

) {

this.partitionWriter = partitionWriter;

this.underlyingKafkaConsumer = underlyingKafkaConsumer;

this.criticalExceptionHandler = criticalExceptionHandler;

this.searchIndexingMetricSet = searchIndexingMetricSet;

this.tweetTopic = tweetTopic;

this.updateTopic = updateTopic;

this.earlybirdIndexFlusher = earlybirdIndexFlusher;

LOG.info("Reading from Kafka topics: tweetTopic={}, updateTopic={}", tweetTopic, updateTopic);

underlyingKafkaConsumer.assign(ImmutableList.of(updateTopic, tweetTopic));

this.balancingKafkaConsumer =

new BalancingKafkaConsumer(underlyingKafkaConsumer, tweetTopic, updateTopic);

this.finishedIngestUntilCurrent = false;

this.indexCaughtUpMonitor = kafkaIndexCaughtUpMonitor;

}

/\*\*

\* Run the consumer, indexing from Kafka.

\*/

@VisibleForTesting

public void run() {

while (isRunning()) {

ConsumeBatchResult result = consumeBatch(true);

indexCaughtUpMonitor.setAndNotify(result.isCaughtUp());

}

}

/\*\*

\* Reads from Kafka, starting at the given offsets, and applies the events until we are caught up

\* with the current streams.

\*/

public void ingestUntilCurrent(long tweetOffset, long updateOffset) {

Preconditions.checkState(!finishedIngestUntilCurrent);

Stopwatch stopwatch = Stopwatch.createStarted();

LOG.info("Ingest until current: seeking to Kafka offset {} for tweets and {} for updates.",

tweetOffset, updateOffset);

try {

underlyingKafkaConsumer.seek(tweetTopic, tweetOffset);

underlyingKafkaConsumer.seek(updateTopic, updateOffset);

} catch (ApiException kafkaApiException) {

throw new WrappedKafkaApiException("Can't seek to tweet and update offsets",

kafkaApiException);

}

Map<TopicPartition, Long> endOffsets;

try {

endOffsets = underlyingKafkaConsumer.endOffsets(ImmutableList.of(tweetTopic, updateTopic));

} catch (ApiException kafkaApiException) {

throw new WrappedKafkaApiException("Can't find end offsets",

kafkaApiException);

}

if (endOffsets.size() > 0) {

LOG.info(String.format("Records until current: tweets=%,d, updates=%,d",

endOffsets.get(tweetTopic) - tweetOffset + 1,

endOffsets.get(updateTopic) - updateOffset + 1));

}

consumeBatchesUntilCurrent(true);

LOG.info("ingestUntilCurrent finished in {}.", stopwatch);

partitionWriter.logState();

INGESTING\_DONE.increment();

finishedIngestUntilCurrent = true;

}

/\*\*

\* Consume tweets and updates from streams until we're up to date.

\*

\* @return total number of read records.

\*/

private long consumeBatchesUntilCurrent(boolean flushingEnabled) {

long totalRecordsRead = 0;

long batchesConsumed = 0;

while (isRunning()) {

ConsumeBatchResult result = consumeBatch(flushingEnabled);

batchesConsumed++;

totalRecordsRead += result.getReadRecordsCount();

if (isCurrent(result.isCaughtUp())) {

break;

}

}

LOG.info("Processed batches: {}", batchesConsumed);

return totalRecordsRead;

}

// This method is overriden in MockEarlybirdKafkaConsumer.

public boolean isCurrent(boolean current) {

return current;

}

/\*\*

\* We don't index during flushing, so after the flush is done, the index is stale.

\* We need to get to current, before we rejoin the serverset so that upon rejoining we're

\* not serving a stale index.

\*/

@VisibleForTesting

void getToCurrentPostFlush() {

LOG.info("Getting to current post flush");

Stopwatch stopwatch = Stopwatch.createStarted();

long totalRecordsRead = consumeBatchesUntilCurrent(false);

LOG.info("Post flush, became current in: {}, after reading {} records.",

stopwatch, LogFormatUtil.formatInt(totalRecordsRead));

}

/\*

\* @return true if we are current after indexing this batch.

\*/

@VisibleForTesting

protected ConsumeBatchResult consumeBatch(boolean flushingEnabled) {

long readRecordsCount = 0;

boolean isCaughtUp = false;

try {

// Poll.

SearchTimer pollTimer = TIMED\_POLLS.startNewTimer();

ConsumerRecords<Long, ThriftVersionedEvents> records =

balancingKafkaConsumer.poll(POLL\_TIMEOUT);

readRecordsCount += records.count();

TIMED\_POLLS.stopTimerAndIncrement(pollTimer);

// Index.

SearchTimer indexTimer = TIMED\_INDEX\_EVENTS.startNewTimer();

isCaughtUp = partitionWriter.indexBatch(records);

TIMED\_INDEX\_EVENTS.stopTimerAndIncrement(indexTimer);

} catch (Exception ex) {

POLL\_LOOP\_EXCEPTIONS.increment();

LOG.error("Exception in poll loop", ex);

}

try {

// Possibly flush the index.

if (isCaughtUp && flushingEnabled) {

long tweetOffset = 0;

long updateOffset = 0;

try {

tweetOffset = underlyingKafkaConsumer.position(tweetTopic);

updateOffset = underlyingKafkaConsumer.position(updateTopic);

} catch (ApiException kafkaApiException) {

throw new WrappedKafkaApiException("can't get topic positions", kafkaApiException);

}

EarlybirdIndexFlusher.FlushAttemptResult flushAttemptResult =

earlybirdIndexFlusher.flushIfNecessary(

tweetOffset, updateOffset, this::getToCurrentPostFlush);

if (flushAttemptResult == EarlybirdIndexFlusher.FlushAttemptResult.FLUSH\_ATTEMPT\_MADE) {

// Viz might show this as a fairly high number, so we're printing it here to confirm

// the value on the server.

LOG.info("Finished flushing. Index freshness in ms: {}",

LogFormatUtil.formatInt(searchIndexingMetricSet.getIndexFreshnessInMillis()));

}

if (!finishedIngestUntilCurrent) {

LOG.info("Became current on startup. Tried to flush with result: {}",

flushAttemptResult);

}

}

} catch (Exception ex) {

FLUSHING\_EXCEPTIONS.increment();

LOG.error("Exception while flushing", ex);

}

return new ConsumeBatchResult(isCaughtUp, readRecordsCount);

}

public boolean isRunning() {

return running.get() && EarlybirdStatus.getStatusCode() != EarlybirdStatusCode.STOPPING;

}

public void prepareAfterStartingWithIndex(long maxIndexedTweetId) {

partitionWriter.prepareAfterStartingWithIndex(maxIndexedTweetId);

}

public void close() {

balancingKafkaConsumer.close();

running.set(false);

}

}