package com.twitter.search.earlybird.partition;

import java.time.Duration;

import java.util.List;

import java.util.Map;

import java.util.concurrent.atomic.AtomicBoolean;

import java.util.stream.Collectors;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Verify;

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.PartitionInfo;

import org.apache.kafka.common.TopicPartition;

import org.apache.kafka.common.errors.WakeupException;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

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

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

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

/\*\*

\* Abstract base class for processing events from Kafka with the goal of indexing them and

\* keeping Earlybirds up to date with the latest events. Indexing is defined by the

\* implementation.

\*

\* NOTE: {@link EarlybirdKafkaConsumer} (tweet/tweet events consumer) is doing this in its

\* own way, we might merge in the future.

\*

\* @param <K> (Long)

\* @param <V> (Event/Thrift type to be consumed)

\*/

public abstract class SimpleStreamIndexer<K, V> {

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

private static final Duration POLL\_TIMEOUT = Duration.ofMillis(250);

private static final Duration CAUGHT\_UP\_FRESHNESS = Duration.ofSeconds(5);

protected static final int MAX\_POLL\_RECORDS = 1000;

private final SearchCounter numPollErrors;

protected SearchRateCounter indexingSuccesses;

protected SearchRateCounter indexingFailures;

protected List<TopicPartition> topicPartitionList;

protected final KafkaConsumer<K, V> kafkaConsumer;

private final AtomicBoolean running = new AtomicBoolean(true);

private final String topic;

private boolean isCaughtUp = false;

/\*\*

\* Create a simple stream indexer.

\*

\* @throws MissingKafkaTopicException - this shouldn't happen, but in case some

\* external stream is not present, we want to have the caller decide how to

\* handle it. Some missing streams might be fatal, for others it might not be

\* justified to block startup. There's no point in constructing this object if

\* a stream is missing, so we don't allow that to happen.

\*/

public SimpleStreamIndexer(KafkaConsumer<K, V> kafkaConsumer,

String topic) throws MissingKafkaTopicException {

this.kafkaConsumer = kafkaConsumer;

this.topic = topic;

List<PartitionInfo> partitionInfos = this.kafkaConsumer.partitionsFor(topic);

if (partitionInfos == null) {

LOG.error("Ooops, no partitions for {}", topic);

NonPagingAssert.assertFailed("missing\_topic\_" + topic);

throw new MissingKafkaTopicException(topic);

}

LOG.info("Discovered {} partitions for topic: {}", partitionInfos.size(), topic);

numPollErrors = SearchCounter.export("stream\_indexer\_poll\_errors\_" + topic);

this.topicPartitionList = partitionInfos

.stream()

.map(info -> new TopicPartition(topic, info.partition()))

.collect(Collectors.toList());

this.kafkaConsumer.assign(topicPartitionList);

}

/\*\*

\* Consume updates on startup until current (eg. until we've seen a record within 5 seconds

\* of current time.)

\*/

public void readRecordsUntilCurrent() {

do {

ConsumerRecords<K, V> records = poll();

for (ConsumerRecord<K, V> record : records) {

if (record.timestamp() > System.currentTimeMillis() - CAUGHT\_UP\_FRESHNESS.toMillis()) {

isCaughtUp = true;

}

validateAndIndexRecord(record);

}

} while (!isCaughtUp());

}

/\*\*

\* Run the consumer, indexing record values directly into their respective structures.

\*/

public void run() {

try {

while (running.get()) {

for (ConsumerRecord<K, V> record : poll()) {

validateAndIndexRecord(record);

}

}

} catch (WakeupException e) {

if (running.get()) {

LOG.error("Caught wakeup exception while running", e);

}

} finally {

kafkaConsumer.close();

LOG.info("Consumer closed.");

}

}

public boolean isCaughtUp() {

return isCaughtUp;

}

/\*\*

\* For every partition in the topic, seek to an offset that has a timestamp greater

\* than or equal to the given timestamp.

\* @param timestamp

\*/

public void seekToTimestamp(Long timestamp) {

Map<TopicPartition, Long> partitionTimestampMap = topicPartitionList.stream()

.collect(Collectors.toMap(tp -> tp, tp -> timestamp));

Map<TopicPartition, OffsetAndTimestamp> partitionOffsetMap =

kafkaConsumer.offsetsForTimes(partitionTimestampMap);

partitionOffsetMap.forEach((tp, offsetAndTimestamp) -> {

Verify.verify(offsetAndTimestamp != null,

"Couldn't find records after timestamp: " + timestamp);

kafkaConsumer.seek(tp, offsetAndTimestamp.offset());

});

}

/\*\*

\* Seeks the kafka consumer to the beginning.

\*/

public void seekToBeginning() {

kafkaConsumer.seekToBeginning(topicPartitionList);

}

/\*\*

\* Polls and returns at most MAX\_POLL\_RECORDS records.

\* @return

\*/

@VisibleForTesting

protected ConsumerRecords<K, V> poll() {

ConsumerRecords<K, V> records;

try {

records = kafkaConsumer.poll(POLL\_TIMEOUT);

} catch (Exception e) {

records = ConsumerRecords.empty();

if (e instanceof WakeupException) {

throw e;

} else {

LOG.warn("Error polling from {} kafka topic.", topic, e);

numPollErrors.increment();

}

}

return records;

}

protected abstract void validateAndIndexRecord(ConsumerRecord<K, V> record);

// Shutdown hook which can be called from a seperate thread. Calling consumer.wakeup() interrupts

// the running indexer and causes it to first stop polling for new records before gracefully

// closing the consumer.

public void close() {

LOG.info("Shutting down stream indexer for topic {}", topic);

running.set(false);

kafkaConsumer.wakeup();

}

}