package com.twitter.search.earlybird.partition;

import java.io.IOException;

import java.util.concurrent.ConcurrentLinkedQueue;

import java.util.concurrent.atomic.AtomicReference;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Preconditions;

import com.google.common.base.Stopwatch;

import com.google.common.base.Verify;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

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

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.index.EarlybirdSegment;

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

import com.twitter.util.Future;

import com.twitter.util.Promise;

/\*\*

\* This class optimizes a segment without blocking reads or writes.

\*

\* In steady state operation (Indexing or Optimized), it delegates operations directly to a

\* SegmentWriter.

\*

\* Optimization is naturally a copying operation -- we don't need to mutate anything internally.

\* We need to be able to apply updates to the unoptimized segment while we are creating

\* the optimized segment. We also need to be able to apply these updates to the optimized segment,

\* but we can't apply updates while a segment is being optimized, because document IDs will be

\* changing internally and posting lists could be any state. To deal with this, we queue updates

\* that occur during optimization, and then apply them as the last step of optimization. At that

\* point, the segment will be optimized and up to date, so we can swap the unoptimized segment for

\* the optimized one.

\*/

public class OptimizingSegmentWriter implements ISegmentWriter {

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

private final AtomicReference<State> state = new AtomicReference<>(State.Indexing);

private final ConcurrentLinkedQueue<ThriftVersionedEvents> queuedEvents =

new ConcurrentLinkedQueue<>();

private final CriticalExceptionHandler criticalExceptionHandler;

private final SearchIndexingMetricSet searchIndexingMetricSet;

private final String segmentName;

private final Promise<SegmentInfo> optimizationPromise = new Promise<>();

// We use the lock to ensure that the optimizing thread and the writer thread do not attempt

// to call indexThriftVersionedEvents on the underlying writer simultaneously.

private final Object lock = new Object();

// The reference to the current writer. Protected by lock.

private final AtomicReference<SegmentWriter> segmentWriterReference;

private final CaughtUpMonitor indexCaughtUpMonitor;

/\*\*

\* The state flow:

\* Indexing -> Optimizing ->

\* ONE OF:

\* - Optimized

\* - FailedToOptimize

\*/

@VisibleForTesting

enum State {

Indexing,

Optimizing,

FailedToOptimize,

Optimized,

}

public OptimizingSegmentWriter(

SegmentWriter segmentWriter,

CriticalExceptionHandler criticalExceptionHandler,

SearchIndexingMetricSet searchIndexingMetricSet,

CaughtUpMonitor indexCaughtUpMonitor

) {

Preconditions.checkState(!segmentWriter.getSegmentInfo().isOptimized());

segmentWriterReference = new AtomicReference<>(segmentWriter);

this.criticalExceptionHandler = criticalExceptionHandler;

this.searchIndexingMetricSet = searchIndexingMetricSet;

this.segmentName = segmentWriter.getSegmentInfo().getSegmentName();

this.indexCaughtUpMonitor = indexCaughtUpMonitor;

}

/\*\*

\* Start optimizing this segment in the background. Returns a Future that will complete when

\* the optimization is complete.

\* Acquires the optimizationAndFlushingCoordinationLock before attempting to optimize.

\*/

public Future<SegmentInfo> startOptimization(

CoordinatedEarlybirdActionInterface gcAction,

OptimizationAndFlushingCoordinationLock optimizationAndFlushingCoordinationLock) {

new Thread(() -> {

// Acquire lock to ensure that flushing is not in progress. If the lock is not available,

// then wait until it is.

LOG.info("Acquire coordination lock before beginning gc\_before\_optimization action.");

try {

optimizationAndFlushingCoordinationLock.lock();

LOG.info("Successfully acquired coordination lock for gc\_before\_optimization action.");

gcAction.retryActionUntilRan("gc before optimization", () -> {

LOG.info("Run GC before optimization");

GCUtil.runGC();

// Wait for indexing to catch up before gcAction rejoins the serverset. We only need to do

// this if the host has already finished startup.

if (EarlybirdStatus.hasStarted()) {

indexCaughtUpMonitor.resetAndWaitUntilCaughtUp();

}

});

} finally {

LOG.info("Finished gc\_before\_optimization action. "

+ "Releasing coordination lock and beginning optimization.");

optimizationAndFlushingCoordinationLock.unlock();

}

transition(State.Indexing, State.Optimizing);

SegmentInfo unoptimizedSegmentInfo = null;

try {

unoptimizedSegmentInfo = segmentWriterReference.get().getSegmentInfo();

Preconditions.checkState(!unoptimizedSegmentInfo.isOptimized());

Stopwatch stopwatch = Stopwatch.createStarted();

LOG.info("Started optimizing segment data {}.", segmentName);

EarlybirdSegment optimizedSegment =

unoptimizedSegmentInfo.getIndexSegment().makeOptimizedSegment();

LOG.info("Finished optimizing segment data {} in {}.", segmentName, stopwatch);

SegmentInfo newSegmentInfo = unoptimizedSegmentInfo

.copyWithEarlybirdSegment(optimizedSegment);

SegmentWriter optimizedWriter =

new SegmentWriter(newSegmentInfo, searchIndexingMetricSet.updateFreshness);

Verify.verify(optimizedWriter.getSegmentInfo().isOptimized());

// We want to apply all updates to the new segment twice, because this first call may apply

// many thousands of updates and take a while to complete.

applyAllPendingUpdates(optimizedWriter);

// We try to do as little as possible while holding the lock, so the writer can continue

// to make progress. First we apply all the updates that have been queued up before we

// grabbed the lock, then we need to swap the new writer for the old one.

synchronized (lock) {

applyAllPendingUpdates(optimizedWriter);

segmentWriterReference.getAndSet(optimizedWriter);

transition(State.Optimizing, State.Optimized);

}

if (!unoptimizedSegmentInfo.isEnabled()) {

LOG.info("Disabling segment: {}", unoptimizedSegmentInfo.getSegmentName());

newSegmentInfo.setIsEnabled(false);

}

optimizationPromise.setValue(newSegmentInfo);

} catch (Throwable e) {

if (unoptimizedSegmentInfo != null) {

unoptimizedSegmentInfo.setFailedOptimize();

}

transition(State.Optimizing, State.FailedToOptimize);

optimizationPromise.setException(e);

}

}, "optimizing-segment-writer").start();

return optimizationPromise;

}

private void applyAllPendingUpdates(SegmentWriter segmentWriter) throws IOException {

LOG.info("Applying {} queued updates to segment {}.", queuedEvents.size(), segmentName);

// More events can be enqueued while this method is running, so we track the total applied too.

long eventCount = 0;

Stopwatch stopwatch = Stopwatch.createStarted();

ThriftVersionedEvents update;

while ((update = queuedEvents.poll()) != null) {

segmentWriter.indexThriftVersionedEvents(update);

eventCount++;

}

LOG.info("Applied {} queued updates to segment {} in {}.",

eventCount, segmentName, stopwatch);

}

@Override

public Result indexThriftVersionedEvents(ThriftVersionedEvents tve) throws IOException {

synchronized (lock) {

if (state.get() == State.Optimizing) {

queuedEvents.add(tve);

}

return segmentWriterReference.get().indexThriftVersionedEvents(tve);

}

}

@Override

public SegmentInfo getSegmentInfo() {

return segmentWriterReference.get().getSegmentInfo();

}

private void transition(State from, State to) {

Preconditions.checkState(state.compareAndSet(from, to));

LOG.info("Transitioned from {} to {} for segment {}.", from, to, segmentName);

}

@VisibleForTesting

public Future<SegmentInfo> getOptimizationPromise() {

return optimizationPromise;

}

}