package com.twitter.search.earlybird.partition;

import java.io.IOException;

import java.util.ArrayList;

import java.util.Collection;

import java.util.Collections;

import java.util.Comparator;

import java.util.HashSet;

import java.util.Iterator;

import java.util.List;

import java.util.Map;

import java.util.Set;

import java.util.concurrent.ConcurrentSkipListMap;

import java.util.stream.Collectors;

import javax.annotation.Nullable;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Preconditions;

import com.google.common.base.Predicate;

import com.google.common.collect.Lists;

import com.google.common.collect.Maps;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.common.util.Clock;

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

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

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

import com.twitter.search.common.partitioning.base.Segment;

import com.twitter.search.common.partitioning.base.TimeSlice;

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

import com.twitter.search.common.schema.base.ImmutableSchemaInterface;

import com.twitter.search.earlybird.EarlybirdIndexConfig;

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

import com.twitter.search.earlybird.common.userupdates.UserScrubGeoMap;

import com.twitter.search.earlybird.common.userupdates.UserUpdate;

import com.twitter.search.earlybird.common.userupdates.UserUpdatesChecker;

import com.twitter.search.earlybird.common.userupdates.UserTable;

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

import com.twitter.search.earlybird.index.EarlybirdSegmentFactory;

import com.twitter.search.earlybird.index.EarlybirdSingleSegmentSearcher;

import com.twitter.search.earlybird.search.EarlybirdLuceneSearcher;

import com.twitter.search.earlybird.search.EarlybirdMultiSegmentSearcher;

import com.twitter.search.earlybird.stats.EarlybirdSearcherStats;

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

import com.twitter.tweetypie.thriftjava.UserScrubGeoEvent;

public class SegmentManager {

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

private final Clock clock;

private static final String STATS\_PREFIX = "segment\_manager\_";

private static final SearchLongGauge SEGMENT\_COUNT\_STATS =

SearchLongGauge.export(STATS\_PREFIX + "total\_segments");

private static final SearchCounter OPTIMIZED\_SEGMENTS =

SearchCounter.export(STATS\_PREFIX + "optimized\_segments");

private static final SearchCounter UNOPTIMIZED\_SEGMENTS =

SearchCounter.export(STATS\_PREFIX + "unoptimized\_segments");

public enum Filter {

All(info -> true),

Enabled(SegmentInfo::isEnabled),

NeedsIndexing(SegmentInfo::needsIndexing),

Complete(SegmentInfo::isComplete);

private final Predicate<SegmentInfo> predicate;

Filter(Predicate<SegmentInfo> predicate) {

this.predicate = predicate;

}

private static final Map<String, Filter> NAME\_INDEX =

Maps.newHashMapWithExpectedSize(Filter.values().length);

static {

for (Filter filter : Filter.values()) {

NAME\_INDEX.put(filter.name().toLowerCase(), filter);

}

}

/\*\*

\* Parses the filter from the given string, based on the filter name.

\*/

public static Filter fromStringIgnoreCase(String str) {

if (str == null) {

return null;

}

return NAME\_INDEX.get(str.toLowerCase());

}

}

public enum Order {

OLD\_TO\_NEW,

NEW\_TO\_OLD,

}

/\*\*

\* A listener that gets notified when the list of segments changes.

\*/

public interface SegmentUpdateListener {

/\*\*

\* Called with the new list of segments when it changes.

\*

\* @param segments The new list of segments.

\*/

void update(Collection<SegmentInfo> segments, String message);

}

private final List<SegmentUpdateListener> updateListeners =

Collections.synchronizedList(Lists.newLinkedList());

private final ConcurrentSkipListMap<Long, ISegmentWriter> segmentWriters =

new ConcurrentSkipListMap<>();

private final Set<Long> badTimesliceIds = new HashSet<>();

private final int maxEnabledSegments;

private final int maxSegmentSize;

private final EarlybirdSegmentFactory earlybirdSegmentFactory;

private final UserTable userTable;

private final UserScrubGeoMap userScrubGeoMap;

private final EarlybirdIndexConfig earlybirdIndexConfig;

private final DynamicPartitionConfig dynamicPartitionConfig;

private final UserUpdatesChecker userUpdatesChecker;

private final SegmentSyncConfig segmentSyncConfig;

private final EarlybirdSearcherStats searcherStats;

private final SearchIndexingMetricSet searchIndexingMetricSet;

private final CriticalExceptionHandler criticalExceptionHandler;

private final CaughtUpMonitor indexCaughtUpMonitor;

public SegmentManager(

DynamicPartitionConfig dynamicPartitionConfig,

EarlybirdIndexConfig earlybirdIndexConfig,

SearchIndexingMetricSet searchIndexingMetricSet,

EarlybirdSearcherStats searcherStats,

SearchStatsReceiver earlybirdStatsReceiver,

UserUpdatesChecker userUpdatesChecker,

SegmentSyncConfig segmentSyncConfig,

UserTable userTable,

UserScrubGeoMap userScrubGeoMap,

Clock clock,

int maxSegmentSize,

CriticalExceptionHandler criticalExceptionHandler,

CaughtUpMonitor indexCaughtUpMonitor) {

PartitionConfig curPartitionConfig = dynamicPartitionConfig.getCurrentPartitionConfig();

this.userTable = userTable;

this.userScrubGeoMap = userScrubGeoMap;

this.earlybirdSegmentFactory = new EarlybirdSegmentFactory(

earlybirdIndexConfig,

searchIndexingMetricSet,

searcherStats,

clock);

this.earlybirdIndexConfig = earlybirdIndexConfig;

this.maxEnabledSegments = curPartitionConfig.getMaxEnabledLocalSegments();

this.dynamicPartitionConfig = dynamicPartitionConfig;

this.userUpdatesChecker = userUpdatesChecker;

this.segmentSyncConfig = segmentSyncConfig;

this.searchIndexingMetricSet = searchIndexingMetricSet;

this.searcherStats = searcherStats;

this.clock = clock;

this.maxSegmentSize = maxSegmentSize;

this.criticalExceptionHandler = criticalExceptionHandler;

this.indexCaughtUpMonitor = indexCaughtUpMonitor;

earlybirdStatsReceiver.getCustomGauge("total\_loaded\_segments",

segmentWriters::size);

earlybirdStatsReceiver.getCustomGauge("total\_indexed\_documents",

this::getNumIndexedDocuments);

earlybirdStatsReceiver.getCustomGauge("total\_segment\_size\_bytes",

this::getTotalSegmentSizeOnDisk);

earlybirdStatsReceiver.getCustomGauge("earlybird\_index\_depth\_millis",

this::getIndexDepthMillis);

}

/\*\*

\* Logs the current state of this segment manager.

\*

\* @param label A label that should identify the segment manager.

\*/

public void logState(String label) {

StringBuilder sb = new StringBuilder();

sb.append("State of SegmentManager (" + label + "):\n");

sb.append("Number of segments: " + segmentWriters.size());

boolean hasSegments = false;

for (Map.Entry<Long, ISegmentWriter> entry : this.segmentWriters.entrySet()) {

SegmentInfo segmentInfo = entry.getValue().getSegmentInfo();

hasSegments = true;

sb.append(String.format("\nSegment (%s): isClosed: %5s, isComplete: %5s, "

+ "isEnabled: %5s, isIndexing: %5s, isOptimized: %5s, wasIndexed: %5s",

segmentInfo.getSegmentName(),

segmentInfo.isClosed(),

segmentInfo.isComplete(),

segmentInfo.isEnabled(),

segmentInfo.isIndexing(),

segmentInfo.isOptimized(),

segmentInfo.wasIndexed()

));

sb.append(String.format(" | Index stats: %s", segmentInfo.getIndexStats().toString()));

}

if (!hasSegments) {

sb.append(" No segments.");

}

LOG.info(sb.toString());

}

public PartitionConfig getPartitionConfig() {

return dynamicPartitionConfig.getCurrentPartitionConfig();

}

public int getMaxEnabledSegments() {

return maxEnabledSegments;

}

public EarlybirdSegmentFactory getEarlybirdSegmentFactory() {

return earlybirdSegmentFactory;

}

public EarlybirdIndexConfig getEarlybirdIndexConfig() {

return earlybirdIndexConfig;

}

public UserTable getUserTable() {

return userTable;

}

public UserScrubGeoMap getUserScrubGeoMap() {

return userScrubGeoMap;

}

@VisibleForTesting

public void reset() {

segmentWriters.clear();

}

/\*\*

\* Returns the list of all segments that match the given filter, in the given order.

\*/

public Iterable<SegmentInfo> getSegmentInfos(Filter filter, Order order) {

Comparator<SegmentInfo> comparator;

if (order == Order.OLD\_TO\_NEW) {

comparator = Comparator.naturalOrder();

} else {

comparator = Comparator.reverseOrder();

}

return () -> segmentWriters.values().stream()

.map(ISegmentWriter::getSegmentInfo)

.filter(filter.predicate::apply)

.sorted(comparator)

.iterator();

}

private void createAndPutSegmentInfo(Segment segment) throws IOException {

LOG.info("Creating new SegmentInfo for segment " + segment.getSegmentName());

putSegmentInfo(new SegmentInfo(segment, earlybirdSegmentFactory, segmentSyncConfig));

}

/\*\*

\* Updates the list of segments managed by this manager, based on the given list.

\*/

public void updateSegments(List<Segment> segmentsList) throws IOException {

// Truncate to the amount of segments we want to keep enabled.

List<Segment> truncatedSegmentList =

SegmentManager.truncateSegmentList(segmentsList, maxEnabledSegments);

final long newestTimeSliceID = getNewestTimeSliceID();

final Set<Long> segmentsToDisable = new HashSet<>(segmentWriters.keySet());

for (Segment segment : truncatedSegmentList) {

final long timeSliceID = segment.getTimeSliceID();

segmentsToDisable.remove(timeSliceID);

// On the first loop iteration of the first call to updateSegments(), newestTimeSliceID should

// be set to -1, so the condition should be false. After that, all segments should either be

// newer than the latest process segment, or if we're replacing an old segment, it should have

// a SegmentInfo instance associated with it.

if (timeSliceID <= newestTimeSliceID) {

ISegmentWriter segmentWriter = segmentWriters.get(timeSliceID);

// Old time slice ID. It should have a SegmentInfo instance associated with it.

if (segmentWriter == null) {

if (!badTimesliceIds.contains(timeSliceID)) {

// We're dealing with a bad timeslice. Log an error, but do it only once per timeslice.

LOG.error("The SegmentInfo instance associated with an old timeSliceID should never be "

+ "null. TimeSliceID: {}", timeSliceID);

badTimesliceIds.add(timeSliceID);

}

} else if (segmentWriter.getSegmentInfo().isClosed()) {

// If the SegmentInfo was closed, create a new one.

LOG.info("SegmentInfo for segment {} is closed.", segment.getSegmentName());

createAndPutSegmentInfo(segment);

}

} else {

// New time slice ID: create a SegmentInfo instance for it.

createAndPutSegmentInfo(segment);

}

}

// Anything we didn't see locally can be disabled.

for (Long segmentID : segmentsToDisable) {

disableSegment(segmentID);

}

// Update segment stats and other exported variables.

updateStats();

}

/\*\*

\* Re-export stats after a segment has changed, or the set of segments has changed.

\*/

public void updateStats() {

// Update the partition count stats.

SEGMENT\_COUNT\_STATS.set(segmentWriters.size());

OPTIMIZED\_SEGMENTS.reset();

UNOPTIMIZED\_SEGMENTS.reset();

for (ISegmentWriter writer : segmentWriters.values()) {

if (writer.getSegmentInfo().isOptimized()) {

OPTIMIZED\_SEGMENTS.increment();

} else {

UNOPTIMIZED\_SEGMENTS.increment();

}

}

}

private long getIndexDepthMillis() {

long oldestTimeSliceID = getOldestEnabledTimeSliceID();

if (oldestTimeSliceID == SegmentInfo.INVALID\_ID) {

return 0;

} else {

// Compute timestamp from timesliceId, which is also a snowflake tweetId

long timestamp = SnowflakeIdParser.getTimestampFromTweetId(oldestTimeSliceID);

// Set current index depth in milliseconds

long indexDepthInMillis = System.currentTimeMillis() - timestamp;

// Index depth should never be negative.

if (indexDepthInMillis < 0) {

LOG.warn("Negative index depth. Large time skew on this Earlybird?");

return 0;

} else {

return indexDepthInMillis;

}

}

}

private void updateExportedSegmentStats() {

int index = 0;

for (SegmentInfo segmentInfo : getSegmentInfos(Filter.Enabled, Order.NEW\_TO\_OLD)) {

SegmentIndexStatsExporter.export(segmentInfo, index++);

}

}

// Marks the SegmentInfo object matching this time slice as disabled.

private void disableSegment(long timeSliceID) {

SegmentInfo info = getSegmentInfo(timeSliceID);

if (info == null) {

LOG.warn("Tried to disable missing segment " + timeSliceID);

return;

}

info.setIsEnabled(false);

LOG.info("Disabled segment " + info);

}

public long getNewestTimeSliceID() {

final Iterator<SegmentInfo> segments = getSegmentInfos(Filter.All, Order.NEW\_TO\_OLD).iterator();

return segments.hasNext() ? segments.next().getTimeSliceID() : SegmentInfo.INVALID\_ID;

}

/\*\*

\* Returns the timeslice ID of the oldest enabled segment.

\*/

public long getOldestEnabledTimeSliceID() {

if (segmentWriters.size() == 0) {

return SegmentInfo.INVALID\_ID;

}

ISegmentWriter segmentWriter = segmentWriters.firstEntry().getValue();

return segmentWriter.getSegmentInfo().getTimeSliceID();

}

/\*\*

\* Returns the SegmentInfo for the given timeSliceID.

\*/

public final SegmentInfo getSegmentInfo(long timeSliceID) {

ISegmentWriter segmentWriter = segmentWriters.get(timeSliceID);

return segmentWriter == null ? null : segmentWriter.getSegmentInfo();

}

/\*\*

\* Returns the segment info for the segment that should contain the given tweet ID.

\*/

public final SegmentInfo getSegmentInfoFromStatusID(long tweetID) {

for (SegmentInfo segmentInfo : getSegmentInfos(Filter.All, Order.NEW\_TO\_OLD)) {

if (tweetID >= segmentInfo.getTimeSliceID()) {

return segmentInfo;

}

}

return null;

}

/\*\*

\* Removes the segment associated with the given timeslice ID from the segment manager. This will

\* also take care of all required clean up related to the segment being removed, such as closing

\* its writer.

\*/

public boolean removeSegmentInfo(long timeSliceID) {

if (timeSliceID == getNewestTimeSliceID()) {

throw new RuntimeException("Cannot drop segment of current time-slice " + timeSliceID);

}

ISegmentWriter removed = segmentWriters.get(timeSliceID);

if (removed == null) {

return false;

}

LOG.info("Removing segment {}", removed.getSegmentInfo());

Preconditions.checkState(!removed.getSegmentInfo().isEnabled());

removed.getSegmentInfo().getIndexSegment().close();

segmentWriters.remove(timeSliceID);

String segmentName = removed.getSegmentInfo().getSegmentName();

updateAllListeners("Removed segment " + segmentName);

LOG.info("Removed segment " + segmentName);

updateExportedSegmentStats();

updateStats();

return true;

}

/\*\*

\* Add the given SegmentWriter into the segmentWriters map.

\* If a segment with the same timesliceID already exists in the map, the old one is replaced

\* with the new one; this should only happen in the archive.

\*

\* The replaced segment is destroyed after a delay to allow in-flight requests to finish.

\*/

public ISegmentWriter putSegmentInfo(SegmentInfo info) {

ISegmentWriter usedSegmentWriter;

SegmentWriter segmentWriter

= new SegmentWriter(info, searchIndexingMetricSet.updateFreshness);

if (!info.isOptimized()) {

LOG.info("Inserting an optimizing segment writer for segment: {}",

info.getSegmentName());

usedSegmentWriter = new OptimizingSegmentWriter(

segmentWriter,

criticalExceptionHandler,

searchIndexingMetricSet,

indexCaughtUpMonitor);

} else {

usedSegmentWriter = segmentWriter;

}

putSegmentWriter(usedSegmentWriter);

return usedSegmentWriter;

}

private void putSegmentWriter(ISegmentWriter segmentWriter) {

SegmentInfo newSegmentInfo = segmentWriter.getSegmentInfo();

SegmentInfo oldSegmentInfo = getSegmentInfo(newSegmentInfo.getTimeSliceID());

// Some sanity checks.

if (oldSegmentInfo != null) {

// This map is thread safe, so this put can be considered atomic.

segmentWriters.put(newSegmentInfo.getTimeSliceID(), segmentWriter);

LOG.info("Replaced SegmentInfo with a new one in segmentWriters map. "

+ "Old SegmentInfo: {} New SegmentInfo: {}", oldSegmentInfo, newSegmentInfo);

if (!oldSegmentInfo.isClosed()) {

oldSegmentInfo.deleteIndexSegmentDirectoryAfterDelay();

}

} else {

long newestTimeSliceID = getNewestTimeSliceID();

if (newestTimeSliceID != SegmentInfo.INVALID\_ID

&& newestTimeSliceID > newSegmentInfo.getTimeSliceID()) {

LOG.error("Not adding out-of-order segment " + newSegmentInfo);

return;

}

segmentWriters.put(newSegmentInfo.getTimeSliceID(), segmentWriter);

LOG.info("Added segment " + newSegmentInfo);

}

updateAllListeners("Added segment " + newSegmentInfo.getTimeSliceID());

updateExportedSegmentStats();

updateStats();

}

private SegmentInfo createSegmentInfo(long timesliceID) throws IOException {

PartitionConfig partitionConfig = dynamicPartitionConfig.getCurrentPartitionConfig();

TimeSlice timeSlice = new TimeSlice(

timesliceID,

maxSegmentSize,

partitionConfig.getIndexingHashPartitionID(),

partitionConfig.getNumPartitions());

SegmentInfo segmentInfo =

new SegmentInfo(timeSlice.getSegment(), earlybirdSegmentFactory, segmentSyncConfig);

return segmentInfo;

}

/\*\*

\* Create a new optimizing segment writer and add it to the map.

\*/

public OptimizingSegmentWriter createAndPutOptimizingSegmentWriter(

long timesliceID) throws IOException {

SegmentInfo segmentInfo = createSegmentInfo(timesliceID);

OptimizingSegmentWriter writer = new OptimizingSegmentWriter(

new SegmentWriter(segmentInfo, searchIndexingMetricSet.updateFreshness),

criticalExceptionHandler,

searchIndexingMetricSet,

indexCaughtUpMonitor);

putSegmentWriter(writer);

return writer;

}

/\*\*

\* Create a new segment writer.

\*/

public SegmentWriter createSegmentWriter(long timesliceID) throws IOException {

SegmentInfo segmentInfo = createSegmentInfo(timesliceID);

SegmentWriter writer = new SegmentWriter(

segmentInfo, searchIndexingMetricSet.updateFreshness);

return writer;

}

private void updateAllListeners(String message) {

List<SegmentInfo> segmentInfos = segmentWriters.values().stream()

.map(ISegmentWriter::getSegmentInfo)

.collect(Collectors.toList());

for (SegmentUpdateListener listener : updateListeners) {

try {

listener.update(segmentInfos, message);

} catch (Exception e) {

LOG.warn("SegmentManager: Unable to call update() on listener.", e);

}

}

}

// Returns true if the map contains a SegmentInfo matching the given time slice.

public final boolean hasSegmentInfo(long timeSliceID) {

return segmentWriters.containsKey(timeSliceID);

}

public void addUpdateListener(SegmentUpdateListener listener) {

updateListeners.add(listener);

}

/\*\*

\* Look up the segment containing the given status id.

\* If found, its timeslice id is returned.

\* If none found, -1 is returned.

\*/

public long lookupTimeSliceID(long statusID) throws IOException {

SegmentInfo segmentInfo = getSegmentInfoForID(statusID);

if (segmentInfo == null) {

return -1;

}

if (!segmentInfo.getIndexSegment().hasDocument(statusID)) {

return -1;

}

return segmentInfo.getTimeSliceID();

}

/\*\*

\* Truncates the given segment list to the specified number of segments, by keeping the newest

\* segments.

\*/

@VisibleForTesting

public static List<Segment> truncateSegmentList(List<Segment> segmentList, int maxNumSegments) {

// Maybe cut-off the beginning of the sorted list of IDs.

if (maxNumSegments > 0 && maxNumSegments < segmentList.size()) {

return segmentList.subList(segmentList.size() - maxNumSegments, segmentList.size());

} else {

return segmentList;

}

}

@VisibleForTesting

public void setOffensive(long userID, boolean offensive) {

userTable.setOffensive(userID, offensive);

}

@VisibleForTesting

public void setAntisocial(long userID, boolean antisocial) {

userTable.setAntisocial(userID, antisocial);

}

/\*\*

\* Returns a searcher for all segments.

\*/

public EarlybirdMultiSegmentSearcher getMultiSearcher(ImmutableSchemaInterface schemaSnapshot)

throws IOException {

return new EarlybirdMultiSegmentSearcher(

schemaSnapshot,

getSearchers(schemaSnapshot, Filter.All, Order.NEW\_TO\_OLD),

searcherStats,

clock);

}

/\*\*

\* Returns a new searcher for the given segment.

\*/

@Nullable

public EarlybirdLuceneSearcher getSearcher(

Segment segment,

ImmutableSchemaInterface schemaSnapshot) throws IOException {

return getSearcher(segment.getTimeSliceID(), schemaSnapshot);

}

/\*\*

\* Get max tweet id across all enabled segments.

\* @return max tweet id or -1 if none found

\*/

public long getMaxTweetIdFromEnabledSegments() {

for (SegmentInfo segmentInfo : getSegmentInfos(Filter.Enabled, Order.NEW\_TO\_OLD)) {

long maxTweetId = segmentInfo.getIndexSegment().getMaxTweetId();

if (maxTweetId != -1) {

return maxTweetId;

}

}

return -1;

}

/\*\*

\* Create a tweet index searcher on the segment represented by the timeslice id. For production

\* search session, the schema snapshot should be always passed in to make sure that the schema

\* usage inside scoring is consistent.

\*

\* For non-production usage, like one-off debugging search, you can use the function call without

\* the schema snapshot.

\*

\* @param timeSliceID the timeslice id, which represents the index segment

\* @param schemaSnapshot the schema snapshot

\* @return the tweet index searcher

\*/

@Nullable

public EarlybirdSingleSegmentSearcher getSearcher(

long timeSliceID,

ImmutableSchemaInterface schemaSnapshot) throws IOException {

SegmentInfo segmentInfo = getSegmentInfo(timeSliceID);

if (segmentInfo == null) {

return null;

}

return segmentInfo.getIndexSegment().getSearcher(userTable, schemaSnapshot);

}

/\*\*

\* Returns a new searcher for the segment with the given timeslice ID. If the given timeslice ID

\* does not correspond to any active segment, {@code null} is returned.

\*

\* @param timeSliceID The segment's timeslice ID.

\* @return A new searcher for the segment with the given timeslice ID.

\*/

@Nullable

public EarlybirdSingleSegmentSearcher getSearcher(long timeSliceID) throws IOException {

SegmentInfo segmentInfo = getSegmentInfo(timeSliceID);

if (segmentInfo == null) {

return null;

}

return segmentInfo.getIndexSegment().getSearcher(userTable);

}

@Nullable

public EarlybirdResponseCode checkSegment(Segment segment) {

return checkSegmentInternal(getSegmentInfo(segment.getTimeSliceID()));

}

private static EarlybirdResponseCode checkSegmentInternal(SegmentInfo info) {

if (info == null) {

return EarlybirdResponseCode.PARTITION\_NOT\_FOUND;

} else if (info.isEnabled()) {

return EarlybirdResponseCode.SUCCESS;

} else {

return EarlybirdResponseCode.PARTITION\_DISABLED;

}

}

private List<EarlybirdSingleSegmentSearcher> getSearchers(

ImmutableSchemaInterface schemaSnapshot,

Filter filter,

Order order) throws IOException {

List<EarlybirdSingleSegmentSearcher> searchers = Lists.newArrayList();

for (SegmentInfo segmentInfo : getSegmentInfos(filter, order)) {

EarlybirdSingleSegmentSearcher searcher =

segmentInfo.getIndexSegment().getSearcher(userTable, schemaSnapshot);

if (searcher != null) {

searchers.add(searcher);

}

}

return searchers;

}

/\*\*

\* Gets metadata for segments for debugging purposes.

\*/

public List<String> getSegmentMetadata() {

List<String> segmentMetadata = new ArrayList<>();

for (SegmentInfo segment : getSegmentInfos(Filter.All, Order.OLD\_TO\_NEW)) {

segmentMetadata.add(segment.getSegmentMetadata());

}

return segmentMetadata;

}

/\*\*

\* Gets info for query caches to be displayed in an admin page.

\*/

public String getQueryCachesData() {

StringBuilder output = new StringBuilder();

for (SegmentInfo segment : getSegmentInfos(Filter.All, Order.OLD\_TO\_NEW)) {

output.append(segment.getQueryCachesData() + "\n");

}

return output.toString();

}

/\*\*

\* Index the given user update. Returns false if the given update is skipped.

\*/

public boolean indexUserUpdate(UserUpdate userUpdate) {

return userTable.indexUserUpdate(userUpdatesChecker, userUpdate);

}

/\*\*

\* Index the given UserScrubGeoEvent.

\* @param userScrubGeoEvent

\*/

public void indexUserScrubGeoEvent(UserScrubGeoEvent userScrubGeoEvent) {

userScrubGeoMap.indexUserScrubGeoEvent(userScrubGeoEvent);

}

/\*\*

\* Return how many documents this segment manager has indexed in all of its enabled segments.

\*/

public long getNumIndexedDocuments() {

// Order here doesn't matter, we just want all enabled segments, and allocate

// as little as needed.

long indexedDocs = 0;

for (SegmentInfo segmentInfo : getSegmentInfos(Filter.Enabled, Order.OLD\_TO\_NEW)) {

indexedDocs += segmentInfo.getIndexSegment().getIndexStats().getStatusCount();

}

return indexedDocs;

}

/\*\*

\* Return how many partial updates this segment manager has applied

\* in all of its enabled segments.

\*/

public long getNumPartialUpdates() {

long partialUpdates = 0;

for (SegmentInfo segmentInfo : getSegmentInfos(Filter.Enabled, Order.OLD\_TO\_NEW)) {

partialUpdates += segmentInfo.getIndexSegment().getIndexStats().getPartialUpdateCount();

}

return partialUpdates;

}

/\*\*

\* Returns the segment info for the segment containing the given tweet ID.

\*/

public SegmentInfo getSegmentInfoForID(long tweetID) {

ISegmentWriter segmentWriter = getSegmentWriterForID(tweetID);

return segmentWriter == null ? null : segmentWriter.getSegmentInfo();

}

/\*\*

\* Returns the segment writer for the segment containing the given tweet ID.

\*/

@Nullable

public ISegmentWriter getSegmentWriterForID(long tweetID) {

Map.Entry<Long, ISegmentWriter> entry = segmentWriters.floorEntry(tweetID);

return entry == null ? null : entry.getValue();

}

/\*\*

\* Remove old segments until we have less than or equal to the number of max enabled segments.

\*/

public void removeExcessSegments() {

int removedSegmentCount = 0;

while (segmentWriters.size() > getMaxEnabledSegments()) {

long timesliceID = getOldestEnabledTimeSliceID();

disableSegment(timesliceID);

removeSegmentInfo(timesliceID);

removedSegmentCount += 1;

}

LOG.info("Segment manager removed {} excess segments", removedSegmentCount);

}

/\*\*

\* Returns total index size on disk across all enabled segments in this segment manager.

\*/

private long getTotalSegmentSizeOnDisk() {

long totalIndexSize = 0;

for (SegmentInfo segmentInfo : getSegmentInfos(Filter.Enabled, Order.OLD\_TO\_NEW)) {

totalIndexSize += segmentInfo.getIndexSegment().getIndexStats().getIndexSizeOnDiskInBytes();

}

return totalIndexSize;

}

@VisibleForTesting

ISegmentWriter getSegmentWriterWithoutCreationForTests(long timesliceID) {

return segmentWriters.get(timesliceID);

}

@VisibleForTesting

ArrayList<Long> getTimeSliceIdsForTests() {

return new ArrayList<Long>(segmentWriters.keySet());

}

}