package com.twitter.search.earlybird.index;

import java.io.Closeable;

import java.io.File;

import java.io.IOException;

import java.time.Instant;

import java.time.ZoneOffset;

import java.time.ZonedDateTime;

import java.time.format.DateTimeFormatter;

import java.util.List;

import java.util.Map;

import java.util.Objects;

import java.util.concurrent.atomic.AtomicReference;

import javax.annotation.Nullable;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Preconditions;

import com.google.common.collect.HashBasedTable;

import com.google.common.collect.Table;

import com.google.common.collect.Lists;

import com.google.common.collect.Maps;

import org.apache.commons.io.FileUtils;

import org.apache.lucene.document.Document;

import org.apache.lucene.index.DirectoryReader;

import org.apache.lucene.index.IndexWriterConfig;

import org.apache.lucene.index.IndexableField;

import org.apache.lucene.store.Directory;

import org.apache.lucene.store.FSDirectory;

import org.apache.lucene.store.IOContext;

import org.apache.lucene.store.IndexOutput;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.common.collections.Pair;

import com.twitter.common.util.Clock;

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

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

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

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

import com.twitter.search.common.schema.earlybird.EarlybirdCluster;

import com.twitter.search.common.schema.earlybird.EarlybirdEncodedFeatures;

import com.twitter.search.common.schema.earlybird.EarlybirdEncodedFeaturesUtil;

import com.twitter.search.common.schema.earlybird.EarlybirdFieldConstants;

import com.twitter.search.common.schema.earlybird.EarlybirdFieldConstants.EarlybirdFieldConstant;

import com.twitter.search.common.schema.thriftjava.ThriftDocument;

import com.twitter.search.common.schema.thriftjava.ThriftField;

import com.twitter.search.common.schema.thriftjava.ThriftIndexingEvent;

import com.twitter.search.common.schema.thriftjava.ThriftIndexingEventType;

import com.twitter.search.common.util.io.flushable.DataDeserializer;

import com.twitter.search.common.util.io.flushable.DataSerializer;

import com.twitter.search.common.util.io.flushable.FlushInfo;

import com.twitter.search.core.earlybird.index.DocIDToTweetIDMapper;

import com.twitter.search.core.earlybird.index.EarlybirdIndexSegmentAtomicReader;

import com.twitter.search.core.earlybird.index.EarlybirdIndexSegmentData;

import com.twitter.search.core.earlybird.index.EarlybirdIndexSegmentWriter;

import com.twitter.search.core.earlybird.index.column.ColumnStrideFieldIndex;

import com.twitter.search.core.earlybird.index.column.DocValuesUpdate;

import com.twitter.search.core.earlybird.index.extensions.EarlybirdIndexExtensionsFactory;

import com.twitter.search.earlybird.EarlybirdIndexConfig;

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

import com.twitter.search.earlybird.document.TweetDocument;

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

import com.twitter.search.earlybird.partition.SearchIndexingMetricSet;

import com.twitter.search.earlybird.partition.SegmentIndexStats;

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

import com.twitter.snowflake.id.SnowflakeId;

public class EarlybirdSegment {

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

private static final Logger UPDATES\_ERRORS\_LOG =

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

private static final String SUCCESS\_FILE = "EARLYBIRD\_SUCCESS";

private static final DateTimeFormatter HOURLY\_COUNT\_DATE\_TIME\_FORMATTER =

DateTimeFormatter.ofPattern("yyyy\_MM\_dd\_HH");

@VisibleForTesting

public static final String NUM\_TWEETS\_CREATED\_AT\_PATTERN = "num\_tweets\_%s\_%s\_created\_at\_%s";

private static final String INVALID\_FEATURE\_UPDATES\_DROPPED\_PREFIX =

"invalid\_index\_feature\_update\_dropped\_";

// The number of tweets not indexed because they have been previously indexed.

private static final SearchCounter DUPLICATE\_TWEET\_SKIPPED\_COUNTER =

SearchCounter.export("duplicate\_tweet\_skipped");

// The number of tweets that came out of order.

private static final SearchCounter OUT\_OF\_ORDER\_TWEET\_COUNTER =

SearchCounter.export("out\_of\_order\_tweet");

// The number partial updates dropped because the field could not be found in the schema.

// This counter is incremented once per field rather than once per partial update event.

// Note: caller may retry update, this counter will be incremented multiple times for same update.

private static final SearchCounter INVALID\_FIELDS\_IN\_PARTIAL\_UPDATES =

SearchCounter.export("invalid\_fields\_in\_partial\_updates");

// The number partial updates dropped because the tweet id could not be found in the segment.

// Note: caller may retry update, this counter will be incremented multiple times for same update.

private static final SearchCounter PARTIAL\_UPDATE\_FOR\_TWEET\_NOT\_IN\_INDEX =

SearchCounter.export("partial\_update\_for\_tweet\_id\_not\_in\_index");

// The number of partial updates that were applied only partially, because the update could not

// be applied for at least one of the fields.

private static final SearchCounter PARTIAL\_UPDATE\_PARTIAL\_FAILURE =

SearchCounter.export("partial\_update\_partial\_failure");

// Both the indexing chain and the index writer are lazily initialized when adding docs for

// the first time.

private final AtomicReference<EarlybirdIndexSegmentWriter> segmentWriterReference =

new AtomicReference<>();

// Stats from the PartitionIndexer / SimpleSegmentIndexer.

private final SegmentIndexStats indexStats;

private final String segmentName;

private final int maxSegmentSize;

private final long timeSliceID;

private final AtomicReference<EarlybirdIndexSegmentAtomicReader> luceneIndexReader =

new AtomicReference<>();

private final Directory luceneDir;

private final File luceneDirFile;

private final EarlybirdIndexConfig indexConfig;

private final List<Closeable> closableResources = Lists.newArrayList();

private long lastInOrderTweetId = 0;

private final EarlybirdIndexExtensionsFactory extensionsFactory;

private final SearchIndexingMetricSet searchIndexingMetricSet;

private final EarlybirdSearcherStats searcherStats;

private final Map<String, SearchCounter> indexedTweetsCounters = Maps.newHashMap();

private final PerFieldCounters perFieldCounters;

private final Clock clock;

@VisibleForTesting

public volatile boolean appendedLuceneIndex = false;

public EarlybirdSegment(

String segmentName,

long timeSliceID,

int maxSegmentSize,

Directory luceneDir,

EarlybirdIndexConfig indexConfig,

SearchIndexingMetricSet searchIndexingMetricSet,

EarlybirdSearcherStats searcherStats,

Clock clock) {

this.segmentName = segmentName;

this.maxSegmentSize = maxSegmentSize;

this.timeSliceID = timeSliceID;

this.luceneDir = luceneDir;

this.indexConfig = indexConfig;

this.indexStats = new SegmentIndexStats();

this.perFieldCounters = new PerFieldCounters();

this.extensionsFactory = new TweetSearchIndexExtensionsFactory();

if (luceneDir != null && luceneDir instanceof FSDirectory) {

// getDirectory() throws if the luceneDir is already closed.

// To delete a directory, we need to close it first.

// Obtain a reference to the File now, so we can delete it later.

// See SEARCH-5281

this.luceneDirFile = ((FSDirectory) luceneDir).getDirectory().toFile();

} else {

this.luceneDirFile = null;

}

this.searchIndexingMetricSet = Preconditions.checkNotNull(searchIndexingMetricSet);

this.searcherStats = searcherStats;

this.clock = clock;

}

@VisibleForTesting

public Directory getLuceneDirectory() {

return luceneDir;

}

public SegmentIndexStats getIndexStats() {

return indexStats;

}

/\*\*

\* Returns the smallest tweet ID in this segment. If the segment is not loaded yet, or is empty,

\* DocIDToTweetIDMapper.ID\_NOT\_FOUND is returned (-1).

\*

\* @return The smallest tweet ID in this segment.

\*/

public long getLowestTweetId() {

EarlybirdIndexSegmentWriter segmentWriter = segmentWriterReference.get();

if (segmentWriter == null) {

return DocIDToTweetIDMapper.ID\_NOT\_FOUND;

}

DocIDToTweetIDMapper mapper = segmentWriter.getSegmentData().getDocIDToTweetIDMapper();

int highestDocID = mapper.getPreviousDocID(Integer.MAX\_VALUE);

return mapper.getTweetID(highestDocID);

}

/\*\*

\* Returns the cardinality (size) sum of the cardinality of each

\* query cache set.

\*/

public long getQueryCachesCardinality() {

EarlybirdIndexSegmentWriter writer = getIndexSegmentWriter();

if (writer == null) {

// The segment is not loaded yet, or the query caches for this segment are not built yet.

return -1;

}

EarlybirdIndexSegmentData earlybirdIndexSegmentData = writer.getSegmentData();

return earlybirdIndexSegmentData.getQueryCachesCardinality();

}

public List<Pair<String, Long>> getQueryCachesData() {

return getIndexSegmentWriter().getSegmentData().getPerQueryCacheCardinality();

}

/\*\*

\* Returns the highest tweet ID in this segment. If the segment is not loaded yet, or is empty,

\* DocIDToTweetIDMapper.ID\_NOT\_FOUND is returned (-1).

\*

\* @return The highest tweet ID in this segment.

\*/

public long getHighestTweetId() {

EarlybirdIndexSegmentWriter segmentWriter = segmentWriterReference.get();

if (segmentWriter == null) {

return DocIDToTweetIDMapper.ID\_NOT\_FOUND;

}

DocIDToTweetIDMapper mapper = segmentWriter.getSegmentData().getDocIDToTweetIDMapper();

int lowestDocID = mapper.getNextDocID(-1);

return mapper.getTweetID(lowestDocID);

}

/\*\*

\* Optimizes the underlying segment data.

\*/

public void optimizeIndexes() throws IOException {

EarlybirdIndexSegmentWriter unoptimizedWriter = segmentWriterReference.get();

Preconditions.checkNotNull(unoptimizedWriter);

unoptimizedWriter.forceMerge();

unoptimizedWriter.close();

// Optimize our own data structures in the indexing chain

// In the archive this is pretty much a no-op.

// The indexWriter in writeableSegment should no longer be used and referenced, and

// writeableSegment.writer can be garbage collected at this point.

EarlybirdIndexSegmentData optimized = indexConfig.optimize(unoptimizedWriter.getSegmentData());

resetSegmentWriterReference(newWriteableSegment(optimized), true);

addSuccessFile();

}

/\*\*

\* Returns a new, optimized, realtime segment, by copying the data in this segment.

\*/

public EarlybirdSegment makeOptimizedSegment() throws IOException {

EarlybirdIndexSegmentWriter unoptimizedWriter = segmentWriterReference.get();

Preconditions.checkNotNull(unoptimizedWriter);

EarlybirdSegment optimizedSegment = new EarlybirdSegment(

segmentName,

timeSliceID,

maxSegmentSize,

luceneDir,

indexConfig,

searchIndexingMetricSet,

searcherStats,

clock);

EarlybirdIndexSegmentData optimizedSegmentData =

indexConfig.optimize(unoptimizedWriter.getSegmentData());

LOG.info("Done optimizing, setting segment data");

optimizedSegment.setSegmentData(

optimizedSegmentData,

indexStats.getPartialUpdateCount(),

indexStats.getOutOfOrderUpdateCount());

return optimizedSegment;

}

public String getSegmentName() {

return segmentName;

}

public boolean isOptimized() {

EarlybirdIndexSegmentWriter segmentWriter = segmentWriterReference.get();

return segmentWriter != null && segmentWriter.getSegmentData().isOptimized();

}

/\*\*

\* Removes the document for the given tweet ID from this segment, if this segment contains a

\* document for this tweet ID.

\*/

public boolean delete(long tweetID) throws IOException {

EarlybirdIndexSegmentWriter segmentWriter = segmentWriterReference.get();

if (!hasDocument(tweetID)) {

return false;

}

segmentWriter.deleteDocuments(new TweetIDQuery(tweetID));

return true;

}

protected void updateDocValues(long tweetID, String field, DocValuesUpdate update)

throws IOException {

EarlybirdIndexSegmentWriter segmentWriter = segmentWriterReference.get();

segmentWriter.updateDocValues(new TweetIDQuery(tweetID), field, update);

}

/\*\*

\* Appends the Lucene index from another segment to this segment.

\*/

public void append(EarlybirdSegment otherSegment) throws IOException {

if (indexConfig.isIndexStoredOnDisk()) {

EarlybirdIndexSegmentWriter segmentWriter = segmentWriterReference.get();

Preconditions.checkNotNull(segmentWriter);

EarlybirdIndexSegmentWriter otherSegmentWriter = otherSegment.segmentWriterReference.get();

if (otherSegmentWriter != null) {

otherSegmentWriter.close();

}

segmentWriter.addIndexes(otherSegment.luceneDir);

LOG.info("Calling forceMerge now after appending segment.");

segmentWriter.forceMerge();

appendedLuceneIndex = true;

LOG.info("Appended {} docs to segment {}. New doc count = {}",

otherSegment.indexStats.getStatusCount(), luceneDir.toString(),

indexStats.getStatusCount());

indexStats.setIndexSizeOnDiskInBytes(getSegmentSizeOnDisk());

}

}

/\*\*

\* Only needed for the on disk archive.

\* Creates TwitterIndexReader used for searching. This is shared by all Searchers.

\* This method also initializes the Lucene based mappers and CSF for the on disk archive.

\*

\* This method should be called after optimizing/loading a segment, but before the segment starts

\* to serve search queries.

\*/

public void warmSegment() throws IOException {

EarlybirdIndexSegmentWriter segmentWriter = segmentWriterReference.get();

Preconditions.checkNotNull(segmentWriter);

// only need to pre-create reader and initialize mappers and CSF in the on disk archive cluster

if (indexConfig.isIndexStoredOnDisk() && luceneIndexReader.get() == null) {

EarlybirdIndexSegmentAtomicReader luceneAtomicReader =

segmentWriter.getSegmentData().createAtomicReader();

luceneIndexReader.set(luceneAtomicReader);

closableResources.add(luceneAtomicReader);

closableResources.add(luceneDir);

}

}

/\*\*

\* Create a tweet index searcher on the segment.

\*

\* 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.

\*/

@Nullable

public EarlybirdSingleSegmentSearcher getSearcher(

UserTable userTable,

ImmutableSchemaInterface schemaSnapshot) throws IOException {

EarlybirdIndexSegmentWriter segmentWriter = segmentWriterReference.get();

if (segmentWriter == null) {

return null;

}

return new EarlybirdSingleSegmentSearcher(

schemaSnapshot, getIndexReader(segmentWriter), userTable, searcherStats, clock);

}

/\*\*

\* Returns a new searcher for this segment.

\*/

@Nullable

public EarlybirdSingleSegmentSearcher getSearcher(

UserTable userTable) throws IOException {

EarlybirdIndexSegmentWriter segmentWriter = segmentWriterReference.get();

if (segmentWriter == null) {

return null;

}

return new EarlybirdSingleSegmentSearcher(

segmentWriter.getSegmentData().getSchema().getSchemaSnapshot(),

getIndexReader(segmentWriter),

userTable,

searcherStats,

clock);

}

/\*\*

\* Returns a new reader for this segment.

\*/

@Nullable

public EarlybirdIndexSegmentAtomicReader getIndexReader() throws IOException {

EarlybirdIndexSegmentWriter segmentWriter = segmentWriterReference.get();

if (segmentWriter == null) {

return null;

}

return getIndexReader(segmentWriter);

}

private EarlybirdIndexSegmentAtomicReader getIndexReader(

EarlybirdIndexSegmentWriter segmentWriter

) throws IOException {

EarlybirdIndexSegmentAtomicReader reader = luceneIndexReader.get();

if (reader != null) {

return reader;

}

Preconditions.checkState(!indexConfig.isIndexStoredOnDisk());

// Realtime EB mode.

return segmentWriter.getSegmentData().createAtomicReader();

}

/\*\*

\* Gets max tweet id in this segment.

\*

\* @return the tweet id or -1 if not found.

\*/

public long getMaxTweetId() {

EarlybirdIndexSegmentWriter segmentWriter = segmentWriterReference.get();

if (segmentWriter == null) {

return -1;

} else {

TweetIDMapper tweetIDMapper =

(TweetIDMapper) segmentWriter.getSegmentData().getDocIDToTweetIDMapper();

return tweetIDMapper.getMaxTweetID();

}

}

private EarlybirdIndexSegmentWriter newWriteableSegment(EarlybirdIndexSegmentData segmentData)

throws IOException {

EarlybirdIndexSegmentWriter old = segmentWriterReference.get();

if (old != null) {

old.close();

}

LOG.info("Creating new segment writer for {} on {}", segmentName, luceneDir);

IndexWriterConfig indexWriterConfig = indexConfig.newIndexWriterConfig();

return segmentData.createEarlybirdIndexSegmentWriter(indexWriterConfig);

}

private void resetSegmentWriterReference(

EarlybirdIndexSegmentWriter segmentWriter, boolean previousSegmentWriterAllowed) {

EarlybirdIndexSegmentWriter previousSegmentWriter =

segmentWriterReference.getAndSet(segmentWriter);

if (!previousSegmentWriterAllowed) {

Preconditions.checkState(

previousSegmentWriter == null,

"A previous segment writer must have been set for segment " + segmentName);

}

// Reset the stats for the number of indexed tweets per hour and recompute them.

// See SEARCH-23619

for (SearchCounter indexedTweetsCounter : indexedTweetsCounters.values()) {

indexedTweetsCounter.reset();

}

if (segmentWriter != null) {

indexStats.setSegmentData(segmentWriter.getSegmentData());

if (indexConfig.getCluster() != EarlybirdCluster.FULL\_ARCHIVE) {

initHourlyTweetCounts(segmentWriterReference.get());

}

} else {

// It's important to unset segment data so that there are no references to it

// and it can be GC-ed.

indexStats.unsetSegmentDataAndSaveCounts();

}

}

/\*\*

\* Add a document if it is not already in segment.

\*/

public void addDocument(TweetDocument doc) throws IOException {

if (indexConfig.isIndexStoredOnDisk()) {

addDocumentToArchiveSegment(doc);

} else {

addDocumentToRealtimeSegment(doc);

}

}

private void addDocumentToArchiveSegment(TweetDocument doc) throws IOException {

// For archive, the document id should come in order, to drop duplicates, only need to

// compare current id with last one.

long tweetId = doc.getTweetID();

if (tweetId == lastInOrderTweetId) {

LOG.warn("Dropped duplicate tweet for archive: {}", tweetId);

DUPLICATE\_TWEET\_SKIPPED\_COUNTER.increment();

return;

}

if (tweetId > lastInOrderTweetId && lastInOrderTweetId != 0) {

// Archive orders document from newest to oldest, so this shouldn't happen

LOG.warn("Encountered out-of-order tweet for archive: {}", tweetId);

OUT\_OF\_ORDER\_TWEET\_COUNTER.increment();

} else {

lastInOrderTweetId = tweetId;

}

addDocumentInternal(doc);

}

private void addDocumentToRealtimeSegment(TweetDocument doc) throws IOException {

long tweetId = doc.getTweetID();

boolean outOfOrder = tweetId <= lastInOrderTweetId;

if (outOfOrder) {

OUT\_OF\_ORDER\_TWEET\_COUNTER.increment();

} else {

lastInOrderTweetId = tweetId;

}

// We only need to call hasDocument() for out-of-order tweets.

if (outOfOrder && hasDocument(tweetId)) {

// We do get duplicates sometimes so you'll see some amount of these.

DUPLICATE\_TWEET\_SKIPPED\_COUNTER.increment();

} else {

addDocumentInternal(doc);

incrementHourlyTweetCount(doc.getTweetID());

}

}

private void addDocumentInternal(TweetDocument tweetDocument) throws IOException {

Document doc = tweetDocument.getDocument();

// Never write blank documents into the index.

if (doc == null || doc.getFields() == null || doc.getFields().size() == 0) {

return;

}

EarlybirdIndexSegmentWriter segmentWriter = segmentWriterReference.get();

if (segmentWriter == null) {

EarlybirdIndexSegmentData segmentData = indexConfig.newSegmentData(

maxSegmentSize,

timeSliceID,

luceneDir,

extensionsFactory);

segmentWriter = newWriteableSegment(segmentData);

resetSegmentWriterReference(segmentWriter, false);

}

Preconditions.checkState(segmentWriter.numDocs() < maxSegmentSize,

"Reached max segment size %s", maxSegmentSize);

IndexableField[] featuresField = doc.getFields(

EarlybirdFieldConstants.ENCODED\_TWEET\_FEATURES\_FIELD\_NAME);

Preconditions.checkState(featuresField.length == 1,

"featuresField.length should be 1, but is %s", featuresField.length);

// We require the createdAt field to be set so we can properly filter tweets based on time.

IndexableField[] createdAt =

doc.getFields(EarlybirdFieldConstant.CREATED\_AT\_FIELD.getFieldName());

Preconditions.checkState(createdAt.length == 1);

EarlybirdEncodedFeatures features = EarlybirdEncodedFeaturesUtil.fromBytes(

indexConfig.getSchema().getSchemaSnapshot(),

EarlybirdFieldConstant.ENCODED\_TWEET\_FEATURES\_FIELD,

featuresField[0].binaryValue().bytes,

featuresField[0].binaryValue().offset);

boolean currentDocIsOffensive = features.isFlagSet(EarlybirdFieldConstant.IS\_OFFENSIVE\_FLAG);

perFieldCounters.increment(ThriftIndexingEventType.INSERT, doc);

segmentWriter.addTweet(doc, tweetDocument.getTweetID(), currentDocIsOffensive);

}

private void incrementHourlyTweetCount(long tweetId) {

// SEARCH-23619, We won't attempt to increment the count for pre-snowflake IDs, since

// extracting an exact create time is pretty tricky at this point, and the stat is mostly

// useful for checking realtime tweet indexing.

if (SnowflakeId.isSnowflakeId(tweetId)) {

long tweetCreateTime = SnowflakeId.unixTimeMillisFromId(tweetId);

String tweetHour = HOURLY\_COUNT\_DATE\_TIME\_FORMATTER.format(

ZonedDateTime.ofInstant(Instant.ofEpochMilli(tweetCreateTime), ZoneOffset.UTC));

String segmentOptimizedSuffix = isOptimized() ? "optimized" : "unoptimized";

SearchCounter indexedTweetsCounter = indexedTweetsCounters.computeIfAbsent(

tweetHour + "\_" + segmentOptimizedSuffix,

(tweetHourKey) -> SearchCounter.export(String.format(

NUM\_TWEETS\_CREATED\_AT\_PATTERN, segmentOptimizedSuffix, segmentName, tweetHour)));

indexedTweetsCounter.increment();

}

}

private void initHourlyTweetCounts(EarlybirdIndexSegmentWriter segmentWriter) {

DocIDToTweetIDMapper mapper = segmentWriter.getSegmentData().getDocIDToTweetIDMapper();

int docId = Integer.MIN\_VALUE;

while ((docId = mapper.getNextDocID(docId)) != DocIDToTweetIDMapper.ID\_NOT\_FOUND) {

incrementHourlyTweetCount(mapper.getTweetID(docId));

}

}

/\*\*

\* Adds the given document for the given tweet ID to the segment, potentially out of order.

\*/

public boolean appendOutOfOrder(Document doc, long tweetID) throws IOException {

// Never write blank documents into the index.

if (doc == null || doc.getFields() == null || doc.getFields().size() == 0) {

return false;

}

EarlybirdIndexSegmentWriter segmentWriter = segmentWriterReference.get();

if (segmentWriter == null) {

logAppendOutOfOrderFailure(tweetID, doc, "segment is null");

return false;

}

if (!indexConfig.supportOutOfOrderIndexing()) {

logAppendOutOfOrderFailure(tweetID, doc, "out of order indexing not supported");

return false;

}

if (!hasDocument(tweetID)) {

logAppendOutOfOrderFailure(tweetID, doc, "tweet ID index lookup failed");

searchIndexingMetricSet.updateOnMissingTweetCounter.increment();

perFieldCounters.incrementTweetNotInIndex(ThriftIndexingEventType.OUT\_OF\_ORDER\_APPEND, doc);

return false;

}

perFieldCounters.increment(ThriftIndexingEventType.OUT\_OF\_ORDER\_APPEND, doc);

segmentWriter.appendOutOfOrder(new TweetIDQuery(tweetID), doc);

indexStats.incrementOutOfOrderUpdateCount();

return true;

}

private void logAppendOutOfOrderFailure(long tweetID, Document doc, String reason) {

UPDATES\_ERRORS\_LOG.debug(

"appendOutOfOrder() failed to apply update document with hash {} on tweet ID {}: {}",

Objects.hashCode(doc), tweetID, reason);

}

/\*\*

\* Determines if this segment contains the given tweet ID.

\*/

public boolean hasDocument(long tweetID) throws IOException {

EarlybirdIndexSegmentWriter segmentWriter = segmentWriterReference.get();

if (segmentWriter == null) {

return false;

}

return segmentWriter.getSegmentData().getDocIDToTweetIDMapper().getDocID(tweetID)

!= DocIDToTweetIDMapper.ID\_NOT\_FOUND;

}

private static final String VERSION\_PROP\_NAME = "version";

private static final String VERSION\_DESC\_PROP\_NAME = "versionDescription";

private static final String PARTIAL\_UPDATES\_COUNT = "partialUpdatesCount";

private static final String OUT\_OF\_ORDER\_UPDATES\_COUNT = "outOfOrderUpdatesCount";

private void checkIfFlushedDataVersionMatchesExpected(FlushInfo flushInfo) throws IOException {

int expectedVersionNumber = indexConfig.getSchema().getMajorVersionNumber();

String expectedVersionDesc = indexConfig.getSchema().getVersionDescription();

int version = flushInfo.getIntProperty(VERSION\_PROP\_NAME);

final String versionDesc = flushInfo.getStringProperty(VERSION\_DESC\_PROP\_NAME);

if (version != expectedVersionNumber) {

throw new FlushVersionMismatchException("Flushed version mismatch. Expected: "

+ expectedVersionNumber + ", but was: " + version);

}

if (!expectedVersionDesc.equals(versionDesc)) {

final String message = "Flush version " + expectedVersionNumber + " is ambiguous"

+ " Expected: " + expectedVersionDesc

+ " Found: " + versionDesc

+ " Please clean up segments with bad flush version from HDFS and Earlybird local disk.";

throw new FlushVersionMismatchException(message);

}

}

/\*\*

\* Loads the segment data and properties from the given deserializer and flush info.

\*

\* @param in The deserializer from which the segment's data will be read.

\* @param flushInfo The flush info from which the segment's properties will be read.

\*/

public void load(DataDeserializer in, FlushInfo flushInfo) throws IOException {

checkIfFlushedDataVersionMatchesExpected(flushInfo);

int partialUpdatesCount = flushInfo.getIntProperty(PARTIAL\_UPDATES\_COUNT);

int outOfOrderUpdatesCount = flushInfo.getIntProperty(OUT\_OF\_ORDER\_UPDATES\_COUNT);

EarlybirdIndexSegmentData loadedSegmentData = indexConfig.loadSegmentData(

flushInfo, in, luceneDir, extensionsFactory);

setSegmentData(loadedSegmentData, partialUpdatesCount, outOfOrderUpdatesCount);

}

/\*\*

\* Update the data backing this EarlyirdSegment.

\*/

public void setSegmentData(

EarlybirdIndexSegmentData segmentData,

int partialUpdatesCount,

int outOfOrderUpdatesCount) throws IOException {

resetSegmentWriterReference(newWriteableSegment(segmentData), false);

try {

warmSegment();

} catch (IOException e) {

LOG.error("Failed to create IndexReader for segment {}. Will destroy unreadable segment.",

segmentName, e);

destroyImmediately();

throw e;

}

LOG.info("Starting segment {} with {} partial updates, {} out of order updates and {} deletes.",

segmentName, partialUpdatesCount, outOfOrderUpdatesCount, indexStats.getDeleteCount());

indexStats.setPartialUpdateCount(partialUpdatesCount);

indexStats.setOutOfOrderUpdateCount(outOfOrderUpdatesCount);

indexStats.setIndexSizeOnDiskInBytes(getSegmentSizeOnDisk());

}

/\*\*

\* Flushes the this segment's properties to the given FlushInfo instance, and this segment's data

\* to the given DataSerializer instance.

\*

\* @param flushInfo The FlushInfo instance where all segment properties should be added.

\* @param out The serializer to which all segment data should be flushed.

\*/

public void flush(FlushInfo flushInfo, DataSerializer out) throws IOException {

flushInfo.addIntProperty(VERSION\_PROP\_NAME, indexConfig.getSchema().getMajorVersionNumber());

flushInfo.addStringProperty(VERSION\_DESC\_PROP\_NAME,

indexConfig.getSchema().getVersionDescription());

flushInfo.addIntProperty(PARTIAL\_UPDATES\_COUNT, indexStats.getPartialUpdateCount());

flushInfo.addIntProperty(OUT\_OF\_ORDER\_UPDATES\_COUNT, indexStats.getOutOfOrderUpdateCount());

if (segmentWriterReference.get() == null) {

LOG.warn("Segment writer is null. flushInfo: {}", flushInfo);

} else if (segmentWriterReference.get().getSegmentData() == null) {

LOG.warn("Segment data is null. segment writer: {}, flushInfo: {}",

segmentWriterReference.get(), flushInfo);

}

segmentWriterReference.get().getSegmentData().flushSegment(flushInfo, out);

indexStats.setIndexSizeOnDiskInBytes(getSegmentSizeOnDisk());

}

/\*\*

\* Check to see if this segment can be loaded from an on-disk index, and load it if it can be.

\*

\* This should only be applicable to the current segment for the on-disk archive. It's not

\* fully flushed until it's full, but we do have a lucene index on local disk which can be

\* used at startup (rather than have to reindex all the current timeslice documents again).

\*

\* If loaded, the index reader will be pre-created, and the segment will be marked as

\* optimized.

\*

\* If the index directory exists but it cannot be loaded, the index directory will be deleted.

\*

\* @return true if the index exists on disk, and was loaded.

\*/

public boolean tryToLoadExistingIndex() throws IOException {

Preconditions.checkState(segmentWriterReference.get() == null);

if (indexConfig.isIndexStoredOnDisk()) {

if (DirectoryReader.indexExists(luceneDir) && checkSuccessFile()) {

LOG.info("Index directory already exists for {} at {}", segmentName, luceneDir);

// set the optimized flag, since we don't need to optimize any more, and pre-create

// the index reader (for the on-disk index optimize() is a noop that just sets the

// optimized flag).

EarlybirdIndexSegmentData earlybirdIndexSegmentData = indexConfig.newSegmentData(

maxSegmentSize,

timeSliceID,

luceneDir,

extensionsFactory);

EarlybirdIndexSegmentData optimizedEarlybirdIndexSegmentData =

indexConfig.optimize(earlybirdIndexSegmentData);

resetSegmentWriterReference(newWriteableSegment(optimizedEarlybirdIndexSegmentData), false);

warmSegment();

LOG.info("Used existing lucene index for {} with {} documents",

segmentName, indexStats.getStatusCount());

indexStats.setIndexSizeOnDiskInBytes(getSegmentSizeOnDisk());

return true;

} else {

// Check if there is an existing lucene dir without a SUCCESS file on disk.

// If so, we will remove it and reindex from scratch.

if (moveFSDirectoryIfExists(luceneDir)) {

// Throw here to be cleaned up and retried by SimpleSegmentIndexer.

throw new IOException("Found invalid existing lucene directory at: " + luceneDir);

}

}

}

return false;

}

/\*\*

\* Partially updates a document with the field value(s) specified by event.

\* Returns true if all writes were successful and false if one or more writes fail or if

\* tweet id isn't found in the segment.

\*/

public boolean applyPartialUpdate(ThriftIndexingEvent event) throws IOException {

Preconditions.checkArgument(event.getEventType() == ThriftIndexingEventType.PARTIAL\_UPDATE);

Preconditions.checkArgument(event.isSetUid());

Preconditions.checkArgument(!ThriftDocumentUtil.hasDuplicateFields(event.getDocument()));

ImmutableSchemaInterface schemaSnapshot = indexConfig.getSchema().getSchemaSnapshot();

long tweetId = event.getUid();

ThriftDocument doc = event.getDocument();

if (!hasDocument(tweetId)) {

// no need to attempt field writes, fail early

PARTIAL\_UPDATE\_FOR\_TWEET\_NOT\_IN\_INDEX.increment();

perFieldCounters.incrementTweetNotInIndex(

ThriftIndexingEventType.PARTIAL\_UPDATE, doc);

return false;

}

int invalidFields = 0;

for (ThriftField field : doc.getFields()) {

String featureName = schemaSnapshot.getFieldName(field.getFieldConfigId());

FeatureConfiguration featureConfig =

schemaSnapshot.getFeatureConfigurationByName(featureName);

if (featureConfig == null) {

INVALID\_FIELDS\_IN\_PARTIAL\_UPDATES.increment();

invalidFields++;

continue;

}

perFieldCounters.increment(ThriftIndexingEventType.PARTIAL\_UPDATE, featureName);

updateDocValues(

tweetId,

featureName,

(docValues, docID) -> updateFeatureValue(docID, featureConfig, docValues, field));

}

if (invalidFields > 0 && invalidFields != doc.getFieldsSize()) {

PARTIAL\_UPDATE\_PARTIAL\_FAILURE.increment();

}

if (invalidFields == 0) {

indexStats.incrementPartialUpdateCount();

} else {

UPDATES\_ERRORS\_LOG.warn("Failed to apply update for tweetID {}, found {} invalid fields: {}",

tweetId, invalidFields, event);

}

return invalidFields == 0;

}

@VisibleForTesting

static void updateFeatureValue(int docID,

FeatureConfiguration featureConfig,

ColumnStrideFieldIndex docValues,

ThriftField updateField) {

int oldValue = Math.toIntExact(docValues.get(docID));

int newValue = updateField.getFieldData().getIntValue();

if (!featureConfig.validateFeatureUpdate(oldValue, newValue)) {

// Counter values can only increase

SearchCounter.export(

INVALID\_FEATURE\_UPDATES\_DROPPED\_PREFIX + featureConfig.getName()).increment();

} else {

docValues.setValue(docID, newValue);

}

}

/\*\*

\* Checks if the provided directory exists and is not empty,

\* and if it does moves it out to a diff directory for later inspection.

\* @param luceneDirectory the dir to move if it exists.

\* @return true iff we found an existing directory.

\*/

private static boolean moveFSDirectoryIfExists(Directory luceneDirectory) {

Preconditions.checkState(luceneDirectory instanceof FSDirectory);

File directory = ((FSDirectory) luceneDirectory).getDirectory().toFile();

if (directory != null && directory.exists() && directory.list().length > 0) {

// Save the bad lucene index by moving it out, for later inspection.

File movedDir = new File(directory.getParent(),

directory.getName() + ".failed." + System.currentTimeMillis());

LOG.warn("Moving existing non-successful index for {} from {} to {}",

luceneDirectory, directory, movedDir);

boolean success = directory.renameTo(movedDir);

if (!success) {

LOG.warn("Unable to rename non-successful index: {}", luceneDirectory);

}

return true;

}

return false;

}

/\*\*

\* For the on-disk archive, if we were able to successfully merge and flush the Lucene index to

\* disk, we mark it explicitly with a SUCCESS file, so that it can be safely reused.

\*/

private void addSuccessFile() throws IOException {

if (indexConfig.isIndexStoredOnDisk()) {

IndexOutput successFile = luceneDir.createOutput(SUCCESS\_FILE, IOContext.DEFAULT);

successFile.close();

}

}

/\*\*

\* Returns the current number of documents in this segment.

\*/

public int getNumDocs() throws IOException {

return indexStats.getStatusCount();

}

/\*\*

\* Reclaim resources used by this segment (E.g. closing lucene index reader).

\* Resources will be reclaimed within the calling thread with no delay.

\*/

public void destroyImmediately() {

try {

closeSegmentWriter();

maybeDeleteSegmentOnDisk();

unloadSegmentFromMemory();

} finally {

indexConfig.getResourceCloser().closeResourcesImmediately(closableResources);

}

}

/\*\*

\* Close the in-memory resources belonging to this segment. This should allow the in-memory

\* segment data to be garbage collected. After closing, the segment is not writable.

\*/

public void close() {

if (segmentWriterReference.get() == null) {

LOG.info("Segment {} already closed.", segmentName);

return;

}

LOG.info("Closing segment {}.", segmentName);

try {

closeSegmentWriter();

unloadSegmentFromMemory();

} finally {

indexConfig.getResourceCloser().closeResourcesImmediately(closableResources);

}

}

private void closeSegmentWriter() {

EarlybirdIndexSegmentWriter segmentWriter = segmentWriterReference.get();

if (segmentWriter != null) {

closableResources.add(() -> {

LOG.info("Closing writer for segment: {}", segmentName);

segmentWriter.close();

});

}

}

private void maybeDeleteSegmentOnDisk() {

if (indexConfig.isIndexStoredOnDisk()) {

Preconditions.checkState(

luceneDir instanceof FSDirectory,

"On-disk indexes should have an underlying directory that we can close and remove.");

closableResources.add(luceneDir);

if (luceneDirFile != null && luceneDirFile.exists()) {

closableResources.add(new Closeable() {

@Override

public void close() throws IOException {

FileUtils.deleteDirectory(luceneDirFile);

}

@Override

public String toString() {

return "delete {" + luceneDirFile + "}";

}

});

}

}

}

private void unloadSegmentFromMemory() {

// Make sure we don't retain a reference to the IndexWriter or SegmentData.

resetSegmentWriterReference(null, true);

}

private long getSegmentSizeOnDisk() throws IOException {

searchIndexingMetricSet.segmentSizeCheckCount.increment();

long totalSize = 0;

if (luceneDir != null) {

for (String file : luceneDir.listAll()) {

totalSize += luceneDir.fileLength(file);

}

}

return totalSize;

}

//////////////////////////

// for unit tests only

//////////////////////////

public EarlybirdIndexConfig getEarlybirdIndexConfig() {

return indexConfig;

}

@VisibleForTesting

public boolean checkSuccessFile() {

return new File(luceneDirFile, SUCCESS\_FILE).exists();

}

@VisibleForTesting

EarlybirdIndexSegmentWriter getIndexSegmentWriter() {

return segmentWriterReference.get();

}

// Helper class to encapsulate counter tables, patterns and various ways to increment

private class PerFieldCounters {

// The number of update/append events for each field in the schema.

private static final String PER\_FIELD\_EVENTS\_COUNTER\_PATTERN = "%s\_for\_field\_%s";

// The number of dropped update/append events for each field due to tweetId not found

private static final String TWEET\_NOT\_IN\_INDEX\_PER\_FIELD\_EVENTS\_COUNTER\_PATTERN =

"%s\_for\_tweet\_id\_not\_in\_index\_for\_field\_%s";

private final Table<ThriftIndexingEventType, String, SearchCounter> perFieldTable =

HashBasedTable.create();

private final Table<ThriftIndexingEventType, String, SearchCounter> notInIndexPerFieldTable =

HashBasedTable.create();

public void increment(

ThriftIndexingEventType eventType, ThriftDocument doc) {

ImmutableSchemaInterface schemaSnapshot = indexConfig.getSchema().getSchemaSnapshot();

for (ThriftField field : doc.getFields()) {

String fieldName = schemaSnapshot.getFieldName(field.getFieldConfigId());

incrementForPattern(

eventType, fieldName, perFieldTable, PER\_FIELD\_EVENTS\_COUNTER\_PATTERN);

}

}

public void incrementTweetNotInIndex(

ThriftIndexingEventType eventType, ThriftDocument doc) {

ImmutableSchemaInterface schemaSnapshot = indexConfig.getSchema().getSchemaSnapshot();

for (ThriftField field : doc.getFields()) {

String fieldName = schemaSnapshot.getFieldName(field.getFieldConfigId());

incrementForPattern(

eventType, fieldName, notInIndexPerFieldTable,

TWEET\_NOT\_IN\_INDEX\_PER\_FIELD\_EVENTS\_COUNTER\_PATTERN);

}

}

public void increment(ThriftIndexingEventType eventType, Document doc) {

for (IndexableField field : doc.getFields()) {

incrementForPattern(

eventType, field.name(),

perFieldTable, PER\_FIELD\_EVENTS\_COUNTER\_PATTERN);

}

}

public void increment(ThriftIndexingEventType eventType, String fieldName) {

incrementForPattern(eventType, fieldName, perFieldTable, PER\_FIELD\_EVENTS\_COUNTER\_PATTERN);

}

public void incrementTweetNotInIndex(ThriftIndexingEventType eventType, Document doc) {

for (IndexableField field : doc.getFields()) {

incrementForPattern(

eventType, field.name(),

notInIndexPerFieldTable,

TWEET\_NOT\_IN\_INDEX\_PER\_FIELD\_EVENTS\_COUNTER\_PATTERN);

}

}

private void incrementForPattern(

ThriftIndexingEventType eventType, String fieldName,

Table<ThriftIndexingEventType, String, SearchCounter> counterTable, String pattern) {

SearchCounter stat;

if (counterTable.contains(eventType, fieldName)) {

stat = counterTable.get(eventType, fieldName);

} else {

stat = SearchCounter.export(String.format(pattern, eventType, fieldName).toLowerCase());

counterTable.put(eventType, fieldName, stat);

}

stat.increment();

}

}

}