package com.twitter.search.earlybird.partition;

import java.io.File;

import java.io.IOException;

import java.io.OutputStreamWriter;

import java.util.concurrent.atomic.AtomicBoolean;

import java.util.concurrent.atomic.AtomicInteger;

import java.util.concurrent.atomic.AtomicLong;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Preconditions;

import org.apache.commons.io.FileUtils;

import org.apache.lucene.store.Directory;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.common.collections.Pair;

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

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

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

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

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

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

import com.twitter.search.earlybird.EarlybirdIndexConfig;

import com.twitter.search.earlybird.common.config.EarlybirdConfig;

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

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

public class SegmentInfo implements Comparable<SegmentInfo> {

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

private static final String UPDATE\_STREAM\_OFFSET\_TIMESTAMP = "updateStreamOffsetTimestamp";

public static final int INVALID\_ID = -1;

// Delay before deleting a segment

private final long timeToWaitBeforeClosingMillis = EarlybirdConfig.getLong(

"defer\_index\_closing\_time\_millis", 600000L);

// How many times deletions are retired.

private final AtomicInteger deletionRetries = new AtomicInteger(5);

// Base segment information, including database name, minStatusId.

private final Segment segment;

// Bits managed by various SegmentProcessors and PartitionManager.

private volatile boolean isEnabled = true; // True if the segment is enabled.

private volatile boolean isIndexing = false; // True during indexing.

private volatile boolean isComplete = false; // True when indexing is complete.

private volatile boolean isClosed = false; // True if indexSegment is closed.

private volatile boolean wasIndexed = false; // True if the segment was indexed from scratch.

private volatile boolean failedOptimize = false; // optimize attempt failed.

private AtomicBoolean beingUploaded = new AtomicBoolean(); // segment is being copied to HDFS

private final SegmentSyncInfo segmentSyncInfo;

private final EarlybirdIndexConfig earlybirdIndexConfig;

private final EarlybirdSegment indexSegment;

private final AtomicLong updatesStreamOffsetTimestamp = new AtomicLong(0);

public SegmentInfo(Segment segment,

EarlybirdSegmentFactory earlybirdSegmentFactory,

SegmentSyncConfig syncConfig) throws IOException {

this(segment, earlybirdSegmentFactory, new SegmentSyncInfo(syncConfig, segment));

}

@VisibleForTesting

public SegmentInfo(Segment segment,

EarlybirdSegmentFactory earlybirdSegmentFactory,

SegmentSyncInfo segmentSyncInfo) throws IOException {

this(earlybirdSegmentFactory.newEarlybirdSegment(segment, segmentSyncInfo),

segmentSyncInfo,

segment,

earlybirdSegmentFactory.getEarlybirdIndexConfig());

}

public SegmentInfo(

EarlybirdSegment earlybirdSegment,

SegmentSyncInfo segmentSyncInfo,

Segment segment,

EarlybirdIndexConfig earlybirdIndexConfig

) {

this.indexSegment = earlybirdSegment;

this.segmentSyncInfo = segmentSyncInfo;

this.earlybirdIndexConfig = earlybirdIndexConfig;

this.segment = segment;

}

public EarlybirdSegment getIndexSegment() {

return indexSegment;

}

public SegmentIndexStats getIndexStats() {

return indexSegment.getIndexStats();

}

public EarlybirdIndexConfig getEarlybirdIndexConfig() {

return earlybirdIndexConfig;

}

public long getTimeSliceID() {

return segment.getTimeSliceID();

}

public String getSegmentName() {

return segment.getSegmentName();

}

public int getNumPartitions() {

return segment.getNumHashPartitions();

}

public boolean isEnabled() {

return isEnabled;

}

public void setIsEnabled(boolean isEnabled) {

this.isEnabled = isEnabled;

}

public boolean isOptimized() {

return indexSegment.isOptimized();

}

public boolean wasIndexed() {

return wasIndexed;

}

public void setWasIndexed(boolean wasIndexed) {

this.wasIndexed = wasIndexed;

}

public boolean isFailedOptimize() {

return failedOptimize;

}

public void setFailedOptimize() {

this.failedOptimize = true;

}

public boolean isIndexing() {

return isIndexing;

}

public void setIndexing(boolean indexing) {

this.isIndexing = indexing;

}

public boolean isComplete() {

return isComplete;

}

public boolean isClosed() {

return isClosed;

}

public boolean isBeingUploaded() {

return beingUploaded.get();

}

public void setBeingUploaded(boolean beingUploaded) {

this.beingUploaded.set(beingUploaded);

}

public boolean casBeingUploaded(boolean expectation, boolean updateValue) {

return beingUploaded.compareAndSet(expectation, updateValue);

}

@VisibleForTesting

public void setComplete(boolean complete) {

this.isComplete = complete;

}

public boolean needsIndexing() {

return isEnabled && !isIndexing && !isComplete;

}

@Override

public int compareTo(SegmentInfo other) {

return Long.compare(getTimeSliceID(), other.getTimeSliceID());

}

@Override

public boolean equals(Object obj) {

return obj instanceof SegmentInfo && compareTo((SegmentInfo) obj) == 0;

}

@Override

public int hashCode() {

return new Long(getTimeSliceID()).hashCode();

}

public long getUpdatesStreamOffsetTimestamp() {

return updatesStreamOffsetTimestamp.get();

}

public void setUpdatesStreamOffsetTimestamp(long timestamp) {

updatesStreamOffsetTimestamp.set(timestamp);

}

@Override

public String toString() {

StringBuilder builder = new StringBuilder();

builder.append(getSegmentName()).append(" [");

builder.append(isEnabled ? "enabled, " : "disabled, ");

if (isIndexing) {

builder.append("indexing, ");

}

if (isComplete) {

builder.append("complete, ");

}

if (isOptimized()) {

builder.append("optimized, ");

}

if (wasIndexed) {

builder.append("wasIndexed, ");

}

builder.append("IndexSync:");

this.segmentSyncInfo.addDebugInfo(builder);

return builder.append("]").toString();

}

public Segment getSegment() {

return segment;

}

/\*\*

\* Delete the index segment directory corresponding to this segment info. Return true if deleted

\* successfully; otherwise, false.

\*/

public boolean deleteLocalIndexedSegmentDirectoryImmediately() {

if (isClosed) {

LOG.info("SegmentInfo is already closed: " + toString());

return true;

}

Preconditions.checkNotNull(indexSegment, "indexSegment should never be null.");

isClosed = true;

indexSegment.destroyImmediately();

SegmentSyncConfig sync = getSyncInfo().getSegmentSyncConfig();

try {

String dirToClear = sync.getLocalSyncDirName(segment);

FileUtils.forceDelete(new File(dirToClear));

LOG.info("Deleted segment directory: " + toString());

return true;

} catch (IOException e) {

LOG.error("Cannot clean up segment directory for segment: " + toString(), e);

return false;

}

}

/\*\*

\* Delete the index segment directory after some configured delay.

\* Note that we don't delete segments that are being uploaded.

\* If a segment is being uploaded when we try to delete, close() retries the deletion later.

\*/

public void deleteIndexSegmentDirectoryAfterDelay() {

LOG.info("Scheduling SegmentInfo for deletion: " + toString());

getEarlybirdIndexConfig().getResourceCloser().closeResourceQuietlyAfterDelay(

timeToWaitBeforeClosingMillis, () -> {

// Atomically check and set the being uploaded flag, if it is not set.

if (beingUploaded.compareAndSet(false, true)) {

// If successfully set the flag to true, we can delete immediately

setIsEnabled(false);

deleteLocalIndexedSegmentDirectoryImmediately();

LOG.info("Deleted index segment dir for segment: "

+ getSegment().getSegmentName());

} else {

// If the flag is already true (compareAndSet fails), we need to reschedule.

if (deletionRetries.decrementAndGet() > 0) {

LOG.warn("Segment is being uploaded, will retry deletion later. SegmentInfo: "

+ getSegment().getSegmentName());

deleteIndexSegmentDirectoryAfterDelay();

} else {

LOG.warn("Failed to cleanup index segment dir for segment: "

+ getSegment().getSegmentName());

}

}

});

}

public SegmentSyncInfo getSyncInfo() {

return segmentSyncInfo;

}

public FlushVersion getFlushVersion() {

return FlushVersion.CURRENT\_FLUSH\_VERSION;

}

public String getZkNodeName() {

return getSegmentName() + getFlushVersion().getVersionFileExtension();

}

static String getSyncDirName(String parentDir, String dbName, String version) {

return parentDir + "/" + dbName + version;

}

/\*\*

\* Parses the segment name from the name of the flushed directory.

\*/

public static String getSegmentNameFromFlushedDir(String flushedDir) {

String segmentName = null;

String[] fields = flushedDir.split("/");

if (fields.length > 0) {

segmentName = fields[fields.length - 1];

segmentName = segmentName.replaceAll(FlushVersion.DELIMITER + ".\*", "");

}

return segmentName;

}

/\*\*

\* Flushes this segment to the given directory.

\*

\* @param dir The directory to flush the segment to.

\* @throws IOException If the segment could not be flushed.

\*/

public void flush(Directory dir) throws IOException {

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

try (PersistentFile.Writer writer = PersistentFile.getWriter(dir, getSegmentName())) {

FlushInfo flushInfo = new FlushInfo();

flushInfo.addLongProperty(UPDATE\_STREAM\_OFFSET\_TIMESTAMP, getUpdatesStreamOffsetTimestamp());

getIndexSegment().flush(flushInfo, writer.getDataSerializer());

OutputStreamWriter infoFileWriter = new OutputStreamWriter(writer.getInfoFileOutputStream());

FlushInfo.flushAsYaml(flushInfo, infoFileWriter);

}

}

/\*\*

\* Makes a new SegmentInfo out of the current segment info, except that we switch the underlying

\* segment.

\*/

public SegmentInfo copyWithEarlybirdSegment(EarlybirdSegment optimizedSegment) {

// Take everything from the current segment info that doesn't change for the new segment

// info and rebuild everything that can change.

TimeSlice newTimeSlice = new TimeSlice(

getTimeSliceID(),

EarlybirdConfig.getMaxSegmentSize(),

segment.getHashPartitionID(),

segment.getNumHashPartitions()

);

Segment newSegment = newTimeSlice.getSegment();

return new SegmentInfo(

optimizedSegment,

new SegmentSyncInfo(

segmentSyncInfo.getSegmentSyncConfig(),

newSegment),

newSegment,

earlybirdIndexConfig

);

}

/\*\*

\* Loads the segment from the given directory.

\*

\* @param dir The directory to load the segment from.

\* @throws IOException If the segment could not be loaded.

\*/

public void load(Directory dir) throws IOException {

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

try (PersistentFile.Reader reader = PersistentFile.getReader(dir, getSegmentName())) {

FlushInfo flushInfo = FlushInfo.loadFromYaml(reader.getInfoInputStream());

setUpdatesStreamOffsetTimestamp(flushInfo.getLongProperty(UPDATE\_STREAM\_OFFSET\_TIMESTAMP));

getIndexSegment().load(reader.getDataInputStream(), flushInfo);

}

}

private String getShortStatus() {

if (!isEnabled()) {

return "disabled";

}

if (isIndexing()) {

return "indexing";

}

if (isComplete()) {

return "indexed";

}

return "pending";

}

/\*\*

\* Get a string to be shown in admin commands which shows the query caches' sizes for this

\* segment.

\*/

public String getQueryCachesData() {

StringBuilder out = new StringBuilder();

out.append("Segment: " + getSegmentName() + "\n");

out.append("Total documents: " + LogFormatUtil.formatInt(

getIndexStats().getStatusCount()) + "\n");

out.append("Query caches:\n");

for (Pair<String, Long> data : indexSegment.getQueryCachesData()) {

out.append(" " + data.getFirst());

out.append(": ");

out.append(LogFormatUtil.formatInt(data.getSecond()));

out.append("\n");

}

return out.toString();

}

public String getSegmentMetadata() {

return "status: " + getShortStatus() + "\n"

+ "id: " + getTimeSliceID() + "\n"

+ "name: " + getSegmentName() + "\n"

+ "statusCount: " + getIndexStats().getStatusCount() + "\n"

+ "deleteCount: " + getIndexStats().getDeleteCount() + "\n"

+ "partialUpdateCount: " + getIndexStats().getPartialUpdateCount() + "\n"

+ "outOfOrderUpdateCount: " + getIndexStats().getOutOfOrderUpdateCount() + "\n"

+ "isEnabled: " + isEnabled() + "\n"

+ "isIndexing: " + isIndexing() + "\n"

+ "isComplete: " + isComplete() + "\n"

+ "isFlushed: " + getSyncInfo().isFlushed() + "\n"

+ "isOptimized: " + isOptimized() + "\n"

+ "isLoaded: " + getSyncInfo().isLoaded() + "\n"

+ "wasIndexed: " + wasIndexed() + "\n"

+ "queryCachesCardinality: " + indexSegment.getQueryCachesCardinality() + "\n";

}

}