package com.twitter.search.earlybird.partition;

import java.io.File;

import java.io.IOException;

import java.util.List;

import java.util.Set;

import java.util.SortedSet;

import java.util.TreeSet;

import javax.annotation.Nonnull;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Preconditions;

import com.google.common.collect.Sets;

import org.apache.commons.io.FileUtils;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

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

import com.twitter.search.earlybird.archive.ArchiveSearchPartitionManager;

import com.twitter.search.earlybird.archive.ArchiveTimeSlicer;

import com.twitter.search.earlybird.archive.ArchiveTimeSlicer.ArchiveTimeSlice;

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

import com.twitter.search.earlybird.factory.EarlybirdIndexConfigUtil;

/\*\*

\* This class removes older flush version segments.

\* Considering that we almost never increase status flush versions, old statuses are not cleaned up

\* automatically.

\*/

public final class SegmentVulture {

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

@VisibleForTesting // Not final for testing.

protected static int numIndexFlushVersionsToKeep =

EarlybirdConfig.getInt("number\_of\_flush\_versions\_to\_keep", 2);

private SegmentVulture() {

// this never gets called

}

/\*\*

\* Delete old build generations, keep currentGeneration.

\*/

@VisibleForTesting

static void removeOldBuildGenerations(String rootDirPath, String currentGeneration) {

File rootDir = new File(rootDirPath);

if (!rootDir.exists() || !rootDir.isDirectory()) {

LOG.error("Root directory is invalid: " + rootDirPath);

return;

}

File[] buildGenerations = rootDir.listFiles();

for (File generation : buildGenerations) {

if (generation.getName().equals(currentGeneration)) {

LOG.info("Skipping current generation: " + generation.getAbsoluteFile());

continue;

}

try {

FileUtils.deleteDirectory(generation);

LOG.info("Deleted old build generation: " + generation.getAbsolutePath());

} catch (IOException e) {

LOG.error("Failed to delete old build generation at: " + generation.getAbsolutePath(), e);

}

}

LOG.info("Successfully deleted all old generations");

}

/\*\*

\* Delete all the timeslice data outside the serving range.

\*/

@VisibleForTesting

static void removeArchiveTimesliceOutsideServingRange(PartitionConfig partitionConfig,

ArchiveTimeSlicer timeSlicer, SegmentSyncConfig segmentSyncConfig) {

try {

long servingStartTimesliceId = Long.MAX\_VALUE;

long servingEndTimesliceId = 0;

int partitionID = partitionConfig.getIndexingHashPartitionID();

List<ArchiveTimeSlice> timeSliceList = timeSlicer.getTimeSlicesInTierRange();

for (ArchiveTimeSlice timeSlice : timeSliceList) {

if (timeSlice.getMinStatusID(partitionID) < servingStartTimesliceId) {

servingStartTimesliceId = timeSlice.getMinStatusID(partitionID);

}

if (timeSlice.getMaxStatusID(partitionID) > servingEndTimesliceId) {

servingEndTimesliceId = timeSlice.getMaxStatusID(partitionID);

}

}

LOG.info("Got the serving range: [" + servingStartTimesliceId + ", "

+ servingEndTimesliceId + "], " + "[" + partitionConfig.getTierStartDate() + ", "

+ partitionConfig.getTierEndDate() + ") for tier: " + partitionConfig.getTierName());

// The tier configuration does not have valid serving range: do not do anything.

if (servingEndTimesliceId <= servingStartTimesliceId) {

LOG.error("Invalid serving range [" + partitionConfig.getTierStartDate() + ", "

+ partitionConfig.getTierEndDate() + "] for tier: " + partitionConfig.getTierName());

return;

}

int numDeleted = 0;

File[] segments = getSegmentsOnRootDir(segmentSyncConfig);

for (File segment : segments) {

String segmentName = SegmentInfo.getSegmentNameFromFlushedDir(segment.getName());

if (segmentName == null) {

LOG.error("Invalid directory for segments: " + segment.getAbsolutePath());

continue;

}

long timesliceId = Segment.getTimeSliceIdFromName(segmentName);

if (timesliceId < 0) {

LOG.error("Unknown dir/file found: " + segment.getAbsolutePath());

continue;

}

if (timesliceId < servingStartTimesliceId || timesliceId > servingEndTimesliceId) {

LOG.info(segment.getAbsolutePath() + " will be deleted for outside serving Range["

+ partitionConfig.getTierStartDate() + ", " + partitionConfig.getTierEndDate() + ")");

if (deleteSegment(segment)) {

numDeleted++;

}

}

}

LOG.info("Deleted " + numDeleted + " segments out of " + segments.length + " segments");

} catch (IOException e) {

LOG.error("Can not timeslice based on the document data: ", e);

throw new RuntimeException(e);

}

}

/\*\*

\* Deleted segments from other partitions. When boxes are moved between

\* partitions, segments from other partitions may stay, we will have to

\* delete them.

\*/

@VisibleForTesting

static void removeIndexesFromOtherPartitions(int myPartition, int numPartitions,

SegmentSyncConfig segmentSyncConfig) {

File[] segments = getSegmentsOnRootDir(segmentSyncConfig);

int numDeleted = 0;

for (File segment : segments) {

int segmentNumPartitions = Segment.numPartitionsFromName(segment.getName());

int segmentPartition = Segment.getPartitionFromName(segment.getName());

if (segmentNumPartitions < 0 || segmentPartition < 0) { // Not a segment file, ignoring

LOG.info("Unknown dir/file found: " + segment.getAbsolutePath());

continue;

}

if (segmentNumPartitions != numPartitions || segmentPartition != myPartition) {

if (deleteSegment(segment)) {

numDeleted++;

}

}

}

LOG.info("Deleted " + numDeleted + " segments out of " + segments.length + " segments");

}

/\*\*

\* Delete flushed segments of older flush versions.

\*/

@VisibleForTesting

static void removeOldFlushVersionIndexes(int currentFlushVersion,

SegmentSyncConfig segmentSyncConfig) {

SortedSet<Integer> indexFlushVersions =

listFlushVersions(segmentSyncConfig, currentFlushVersion);

if (indexFlushVersions == null

|| indexFlushVersions.size() <= numIndexFlushVersionsToKeep) {

return;

}

Set<String> suffixesToKeep = Sets.newHashSetWithExpectedSize(numIndexFlushVersionsToKeep);

int flushVersionsToKeep = numIndexFlushVersionsToKeep;

while (flushVersionsToKeep > 0 && !indexFlushVersions.isEmpty()) {

Integer oldestFlushVersion = indexFlushVersions.last();

String flushFileExtension = FlushVersion.getVersionFileExtension(oldestFlushVersion);

if (flushFileExtension != null) {

suffixesToKeep.add(flushFileExtension);

flushVersionsToKeep--;

} else {

LOG.warn("Found unknown flush versions: " + oldestFlushVersion

+ " Segments with this flush version will be deleted to recover disk space.");

}

indexFlushVersions.remove(oldestFlushVersion);

}

String segmentSyncRootDir = segmentSyncConfig.getLocalSegmentSyncRootDir();

File dir = new File(segmentSyncRootDir);

File[] segments = dir.listFiles();

for (File segment : segments) {

boolean keepSegment = false;

for (String suffix : suffixesToKeep) {

if (segment.getName().endsWith(suffix)) {

keepSegment = true;

break;

}

}

if (!keepSegment) {

try {

FileUtils.deleteDirectory(segment);

LOG.info("Deleted old flushed segment: " + segment.getAbsolutePath());

} catch (IOException e) {

LOG.error("Failed to delete old flushed segment.", e);

}

}

}

}

private static File[] getSegmentsOnRootDir(SegmentSyncConfig segmentSyncConfig) {

String segmentSyncRootDir = segmentSyncConfig.getLocalSegmentSyncRootDir();

File dir = new File(segmentSyncRootDir);

File[] segments = dir.listFiles();

if (segments == null) {

return new File[0];

} else {

return segments;

}

}

private static boolean deleteSegment(File segment) {

try {

FileUtils.deleteDirectory(segment);

LOG.info("Deleted segment from other partition: " + segment.getAbsolutePath());

return true;

} catch (IOException e) {

LOG.error("Failed to delete segment from other partition.", e);

return false;

}

}

// Returns FlushVersions found on disk.

// Current FlushVersion is always added into the list, even if segments are not found on disk,

// because they may not have appeared yet.

@Nonnull

@VisibleForTesting

static SortedSet<Integer> listFlushVersions(SegmentSyncConfig sync, int currentFlushVersion) {

TreeSet<Integer> flushVersions = Sets.newTreeSet();

// Always add current flush version.

// It is possible that on startup when this is run, the current flush version

// segments have not appeared yet.

flushVersions.add(currentFlushVersion);

String segmentSyncRootDir = sync.getLocalSegmentSyncRootDir();

File dir = new File(segmentSyncRootDir);

if (!dir.exists()) {

LOG.info("segmentSyncRootDir [" + segmentSyncRootDir

+ "] does not exist");

return flushVersions;

}

if (!dir.isDirectory()) {

LOG.error("segmentSyncRootDir [" + segmentSyncRootDir

+ "] does not point to a directory");

return flushVersions;

}

if (!dir.canRead()) {

LOG.error("No permission to read from segmentSyncRootDir ["

+ segmentSyncRootDir + "]");

return flushVersions;

}

if (!dir.canWrite()) {

LOG.error("No permission to write to segmentSyncRootDir ["

+ segmentSyncRootDir + "]");

return flushVersions;

}

File[] segments = dir.listFiles();

for (File segment : segments) {

String name = segment.getName();

if (!name.contains(FlushVersion.DELIMITER)) {

// This is a not a segment with a FlushVersion, skip.

LOG.info("Found segment directory without a flush version: " + name);

continue;

}

String[] nameSplits = name.split(FlushVersion.DELIMITER);

if (nameSplits.length != 2) {

LOG.warn("Found segment with bad name: " + segment.getAbsolutePath());

continue;

}

// Second half contains flush version

try {

int flushVersion = Integer.parseInt(nameSplits[1]);

flushVersions.add(flushVersion);

} catch (NumberFormatException e) {

LOG.warn("Bad flush version number in segment name: " + segment.getAbsolutePath());

}

}

return flushVersions;

}

/\*\*

\* Removes old segments in the current build gen.

\*/

@VisibleForTesting

static void removeOldSegments(SegmentSyncConfig sync) {

if (!sync.getScrubGen().isPresent()) {

return;

}

File currentScrubGenSegmentDir = new File(sync.getLocalSegmentSyncRootDir());

// The unscrubbed segment root directory, used for rebuilds and for segments created before

// we introduced scrub gens. The getLocalSegmentSyncRootDir should be something like:

// $unscrubbedSegmentDir/scrubbed/$scrub\_gen/,

// get unscrubbedSegmentDir from string name here in case scrubbed dir does not exist yet

File unscrubbedSegmentDir = new File(sync.getLocalSegmentSyncRootDir().split("scrubbed")[0]);

if (!unscrubbedSegmentDir.exists()) {

// For a new host that swapped in, it might not have flushed\_segment dir yet.

// return directly in that case.

LOG.info(unscrubbedSegmentDir.getAbsoluteFile() + "does not exist, nothing to remove.");

return;

}

Preconditions.checkArgument(unscrubbedSegmentDir.exists());

for (File file : unscrubbedSegmentDir.listFiles()) {

if (file.getName().matches("scrubbed")) {

continue;

}

try {

LOG.info("Deleting old unscrubbed segment: " + file.getAbsolutePath());

FileUtils.deleteDirectory(file);

} catch (IOException e) {

LOG.error("Failed to delete directory: " + file.getPath(), e);

}

}

// Delete all segments from previous scrub generations.

File allScrubbedSegmentsDir = currentScrubGenSegmentDir.getParentFile();

if (allScrubbedSegmentsDir.exists()) {

for (File file : allScrubbedSegmentsDir.listFiles()) {

if (file.getPath().equals(currentScrubGenSegmentDir.getPath())) {

continue;

}

try {

LOG.info("Deleting old scrubbed segment: " + file.getAbsolutePath());

FileUtils.deleteDirectory(file);

} catch (IOException e) {

LOG.error("Failed to delete directory: " + file.getPath(), e);

}

}

}

}

/\*\*

\* Removes the data for all unused segments from the local disk. This includes:

\* - data for old segments

\* - data for segments belonging to another partition

\* - data for segments belonging to a different flush version.

\*/

public static void removeUnusedSegments(

PartitionManager partitionManager,

PartitionConfig partitionConfig,

int schemaMajorVersion,

SegmentSyncConfig segmentSyncConfig) {

if (EarlybirdIndexConfigUtil.isArchiveSearch()) {

removeOldBuildGenerations(

EarlybirdConfig.getString("root\_dir"),

EarlybirdConfig.getString("offline\_segment\_build\_gen")

);

removeOldSegments(segmentSyncConfig);

Preconditions.checkState(partitionManager instanceof ArchiveSearchPartitionManager);

removeArchiveTimesliceOutsideServingRange(

partitionConfig,

((ArchiveSearchPartitionManager) partitionManager).getTimeSlicer(), segmentSyncConfig);

}

// Remove segments from other partitions

removeIndexesFromOtherPartitions(

partitionConfig.getIndexingHashPartitionID(),

partitionConfig.getNumPartitions(), segmentSyncConfig);

// Remove old flushed segments

removeOldFlushVersionIndexes(schemaMajorVersion, segmentSyncConfig);

}

}