package com.twitter.search.earlybird.archive.segmentbuilder;

import java.io.IOException;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Date;

import java.util.HashMap;

import java.util.Iterator;

import java.util.List;

import java.util.Map;

import java.util.Optional;

import java.util.Random;

import java.util.concurrent.TimeUnit;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Preconditions;

import com.google.common.base.Stopwatch;

import com.google.common.collect.ComparisonChain;

import com.google.common.collect.ImmutableList;

import com.google.common.util.concurrent.Uninterruptibles;

import com.google.inject.Inject;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.common.quantity.Amount;

import com.twitter.common.quantity.Time;

import com.twitter.common.util.Clock;

import com.twitter.decider.Decider;

import com.twitter.inject.annotations.Flag;

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.metrics.SearchStatsReceiverImpl;

import com.twitter.search.common.partitioning.zookeeper.SearchZkClient;

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

import com.twitter.search.common.util.zktrylock.ZooKeeperTryLockFactory;

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

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

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

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

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

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

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

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

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

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

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

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

/\*\*

\* This class provides the core logic to build segment indices offline.

\* For each server, it coordinate via zookeeper to pick the next segment, build the indices for it

\* and upload them to HDFS. A state machine is used to handle the build state transitions. There

\* are three states:

\* NOT\_BUILD\_YET: a segment that needs to be built

\* SOMEONE\_ELSE\_IS\_BUILDING: another server is building the segment.

\* BUILT\_AND\_FINALIZED: the indices of this segment have already been built.

\*/

public class SegmentBuilder {

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

private final boolean onlyRunOnce;

private final int waitBetweenLoopsMins;

private final int startUpBatchSize;

private final int instance;

private final int waitBetweenSegmentsSecs;

private final int waitBeforeQuitMins;

// When multiple segment builders start simultaneously, they might make the HDFS name node and

// zookeeper overwhelmed. So, we let some instances sleep sometimes before they start to avoid

// the issues.

private final long startUpSleepMins;

// If no more segments to built, wait this interval before checking again.

private final long processWaitingInterval = TimeUnit.MINUTES.toMillis(10);

// The hash partitions that segments will be built.

private final ImmutableList<Integer> hashPartitions;

private final SearchStatsReceiver statsReceiver = new SearchStatsReceiverImpl();

private final SearchIndexingMetricSet searchIndexingMetricSet =

new SearchIndexingMetricSet(statsReceiver);

private final EarlybirdSearcherStats searcherStats =

new EarlybirdSearcherStats(statsReceiver);

private final ArchiveOnDiskEarlybirdIndexConfig earlybirdIndexConfig;

private final ZooKeeperTryLockFactory zkTryLockFactory;

private final RateLimitingSegmentHandler segmentHandler;

private final Clock clock;

private final int numSegmentBuilderPartitions;

private final int myPartitionId;

private final SegmentConfig segmentConfig;

private final EarlybirdSegmentFactory segmentFactory;

private final SegmentBuilderCoordinator segmentBuilderCoordinator;

private final SegmentSyncConfig segmentSyncConfig;

private final Random random = new Random();

private static final double SLEEP\_RANDOMIZATION\_RATIO = .2;

// Stats

// The flush version used to build segments

private static final SearchLongGauge CURRENT\_FLUSH\_VERSION =

SearchLongGauge.export("current\_flush\_version");

// Accumulated number and time in seconds spent on building segments locally

private static SearchCounter segmentsBuiltLocally =

SearchCounter.export("segments\_built\_locally");

private static SearchCounter timeSpentOnSuccessfulBuildSecs =

SearchCounter.export("time\_spent\_on\_successful\_build\_secs");

// The total number of segments to be built

private static final SearchLongGauge SEGMENTS\_TO\_BUILD =

SearchLongGauge.export("segments\_to\_build");

// How many segments failed locally

private static final SearchCounter FAILED\_SEGMENTS =

SearchCounter.export("failed\_segments");

@Inject

protected SegmentBuilder(@Flag("onlyRunOnce") boolean onlyRunOnceFlag,

@Flag("waitBetweenLoopsMins") int waitBetweenLoopsMinsFlag,

@Flag("startup\_batch\_size") int startUpBatchSizeFlag,

@Flag("instance") int instanceFlag,

@Flag("segmentZkLockExpirationHours")

int segmentZkLockExpirationHoursFlag,

@Flag("startupSleepMins") long startupSleepMinsFlag,

@Flag("maxRetriesOnFailure") int maxRetriesOnFailureFlag,

@Flag("hash\_partitions") List<Integer> hashPartitionsFlag,

@Flag("numSegmentBuilderPartitions") int numSegmentBuilderPartitionsFlag,

@Flag("waitBetweenSegmentsSecs") int waitBetweenSegmentsSecsFlag,

@Flag("waitBeforeQuitMins") int waitBeforeQuitMinsFlag,

@Flag("scrubGen") String scrubGen,

Decider decider) {

this(onlyRunOnceFlag,

waitBetweenLoopsMinsFlag,

startUpBatchSizeFlag,

instanceFlag,

segmentZkLockExpirationHoursFlag,

startupSleepMinsFlag,

hashPartitionsFlag,

maxRetriesOnFailureFlag,

waitBetweenSegmentsSecsFlag,

waitBeforeQuitMinsFlag,

SearchZkClient.getSZooKeeperClient().createZooKeeperTryLockFactory(),

new RateLimitingSegmentHandler(TimeUnit.MINUTES.toMillis(10), Clock.SYSTEM\_CLOCK),

Clock.SYSTEM\_CLOCK,

numSegmentBuilderPartitionsFlag,

decider,

getSyncConfig(scrubGen));

}

@VisibleForTesting

protected SegmentBuilder(boolean onlyRunOnceFlag,

int waitBetweenLoopsMinsFlag,

int startUpBatchSizeFlag,

int instanceFlag,

int segmentZkLockExpirationHoursFlag,

long startupSleepMinsFlag,

List<Integer> hashPartitions,

int maxRetriesOnFailure,

int waitBetweenSegmentsSecsFlag,

int waitBeforeQuitMinsFlag,

ZooKeeperTryLockFactory zooKeeperTryLockFactory,

RateLimitingSegmentHandler segmentHandler,

Clock clock,

int numSegmentBuilderPartitions,

Decider decider,

SegmentSyncConfig syncConfig) {

LOG.info("Creating SegmentBuilder");

LOG.info("Penguin version in use: " + EarlybirdConfig.getPenguinVersion());

// Set command line flag values

this.onlyRunOnce = onlyRunOnceFlag;

this.waitBetweenLoopsMins = waitBetweenLoopsMinsFlag;

this.startUpBatchSize = startUpBatchSizeFlag;

this.instance = instanceFlag;

this.waitBetweenSegmentsSecs = waitBetweenSegmentsSecsFlag;

this.waitBeforeQuitMins = waitBeforeQuitMinsFlag;

this.segmentHandler = segmentHandler;

this.zkTryLockFactory = zooKeeperTryLockFactory;

this.segmentSyncConfig = syncConfig;

this.startUpSleepMins = startupSleepMinsFlag;

if (!hashPartitions.isEmpty()) {

this.hashPartitions = ImmutableList.copyOf(hashPartitions);

} else {

this.hashPartitions = null;

}

Amount<Long, Time> segmentZKLockExpirationTime = Amount.of((long)

segmentZkLockExpirationHoursFlag, Time.HOURS);

this.earlybirdIndexConfig =

new ArchiveOnDiskEarlybirdIndexConfig(decider, searchIndexingMetricSet,

new CriticalExceptionHandler());

this.segmentConfig = new SegmentConfig(

earlybirdIndexConfig,

segmentZKLockExpirationTime,

maxRetriesOnFailure,

zkTryLockFactory);

this.segmentFactory = new EarlybirdSegmentFactory(

earlybirdIndexConfig,

searchIndexingMetricSet,

searcherStats,

clock);

this.segmentBuilderCoordinator = new SegmentBuilderCoordinator(

zkTryLockFactory, syncConfig, clock);

this.clock = clock;

this.numSegmentBuilderPartitions = numSegmentBuilderPartitions;

this.myPartitionId = instance % numSegmentBuilderPartitions;

SearchLongGauge.export("segment\_builder\_partition\_id\_" + myPartitionId).set(1);

CURRENT\_FLUSH\_VERSION.set(earlybirdIndexConfig.getSchema().getMajorVersionNumber());

}

void run() {

LOG.info("Config values: {}", EarlybirdConfig.allValuesAsString());

// Sleep some time uninterruptibly before get started so that if multiple instances are running,

// the HDFS name node and zookeeper wont be overwhelmed

// Say, we have 100 instances (instance\_arg will have value from 0 - 99, our

// STARTUP\_BATCH\_SIZE\_ARG is 20 and startUpSleepMins is 3 mins. Then the first 20 instances

// will not sleep, but start immediately. then instance 20 - 39 will sleep 3 mins and then

// start to run. instance 40 - 59 will sleep 6 mins then start to run. instances 60 - 79 will

// sleep 9 mins and then start to run and so forth.

long sleepTime = instance / startUpBatchSize \* startUpSleepMins;

LOG.info("Instance={}, Start up batch size={}", instance, startUpBatchSize);

LOG.info("Sleep {} minutes to void HDFS name node and ZooKeeper overwhelmed.", sleepTime);

Uninterruptibles.sleepUninterruptibly(sleepTime, TimeUnit.MINUTES);

// Kinit here.

Kerberos.kinit(

EarlybirdConfig.getString("kerberos\_user", ""),

EarlybirdConfig.getString("kerberos\_keytab\_path", "")

);

long waitBetweenLoopsMs = TimeUnit.MINUTES.toMillis(waitBetweenLoopsMins);

if (onlyRunOnce) {

LOG.info("This segment builder will run the full rebuild of all the segments");

} else {

LOG.info("This segment builder will incrementally check for new data and rebuilt "

+ "current segments as needed.");

LOG.info("The waiting interval between two new data checking is: "

+ waitBetweenLoopsMs + " ms.");

}

boolean scrubGenPresent = segmentSyncConfig.getScrubGen().isPresent();

LOG.info("Scrub gen present: {}", scrubGenPresent);

boolean scrubGenDataFullyBuilt = segmentBuilderCoordinator.isScrubGenDataFullyBuilt(instance);

LOG.info("Scrub gen data fully built: {}", scrubGenDataFullyBuilt);

if (!scrubGenPresent || scrubGenDataFullyBuilt) {

LOG.info("Starting segment building loop...");

while (!Thread.currentThread().isInterrupted()) {

try {

indexingLoop();

if (onlyRunOnce) {

LOG.info("only run once is true, breaking");

break;

}

clock.waitFor(waitBetweenLoopsMs);

} catch (InterruptedException e) {

LOG.info("Interrupted, quitting segment builder");

Thread.currentThread().interrupt();

} catch (SegmentInfoConstructionException e) {

LOG.error("Error creating new segmentInfo, quitting segment builder: ", e);

break;

} catch (SegmentUpdaterException e) {

FAILED\_SEGMENTS.increment();

// Before the segment builder quits, sleep for WAIT\_BEFORE\_QUIT\_MINS minutes so that the

// FAILED\_SEGMENTS stat can be exported.

try {

clock.waitFor(TimeUnit.MINUTES.toMillis(waitBeforeQuitMins));

} catch (InterruptedException ex) {

LOG.info("Interrupted, quitting segment builder");

Thread.currentThread().interrupt();

}

LOG.error("SegmentUpdater processing segment error, quitting segment builder: ", e);

break;

}

}

} else {

LOG.info("Cannot build the segments for scrub gen yet.");

}

}

// Refactoring the run loop to here for unittest

@VisibleForTesting

void indexingLoop()

throws SegmentInfoConstructionException, InterruptedException, SegmentUpdaterException {

// This map contains all the segments to be processed; if a segment is built, it will be removed

// from the map.

Map<String, SegmentBuilderSegment> buildableSegmentInfoMap;

try {

buildableSegmentInfoMap = createSegmentInfoMap();

printSegmentInfoMap(buildableSegmentInfoMap);

} catch (IOException e) {

LOG.error("Error creating segmentInfoMap: ", e);

return;

}

while (!buildableSegmentInfoMap.isEmpty()) {

boolean hasBuiltSegment = processSegments(buildableSegmentInfoMap);

if (!hasBuiltSegment) {

// If we successfully built a segment, no need to sleep since building a segment takes a

// long time

clock.waitFor(processWaitingInterval);

}

}

}

// Actual shutdown.

protected void doShutdown() {

LOG.info("doShutdown()...");

try {

earlybirdIndexConfig.getResourceCloser().shutdownExecutor();

} catch (InterruptedException e) {

LOG.error("Interrupted during shutdown. ", e);

}

LOG.info("Segment builder stopped!");

}

private List<ArchiveTimeSlicer.ArchiveTimeSlice> createTimeSlices() throws IOException {

Preconditions.checkState(segmentSyncConfig.getScrubGen().isPresent());

Date scrubGen = ScrubGenUtil.parseScrubGenToDate(segmentSyncConfig.getScrubGen().get());

final DailyStatusBatches dailyStatusBatches =

new DailyStatusBatches(zkTryLockFactory, scrubGen);

final ArchiveTimeSlicer archiveTimeSlicer = new ArchiveTimeSlicer(

EarlybirdConfig.getMaxSegmentSize(), dailyStatusBatches, earlybirdIndexConfig);

Stopwatch stopwatch = Stopwatch.createStarted();

List<ArchiveTimeSlicer.ArchiveTimeSlice> timeSlices = archiveTimeSlicer.getTimeSlices();

if (timeSlices == null) {

LOG.error("Failed to load timeslice map after {}", stopwatch);

return Collections.emptyList();

}

LOG.info("Took {} to get timeslices", stopwatch);

return timeSlices;

}

private static class TimeSliceAndHashPartition implements Comparable<TimeSliceAndHashPartition> {

public final ArchiveTimeSlicer.ArchiveTimeSlice timeSlice;

public final Integer hashPartition;

public TimeSliceAndHashPartition(

ArchiveTimeSlicer.ArchiveTimeSlice timeSlice,

Integer hashPartition) {

this.timeSlice = timeSlice;

this.hashPartition = hashPartition;

}

@Override

public int compareTo(TimeSliceAndHashPartition o) {

Integer myHashPartition = this.hashPartition;

Integer otherHashPartition = o.hashPartition;

long myTimeSliceId = this.timeSlice.getMinStatusID(myHashPartition);

long otherTimeSliceId = o.timeSlice.getMinStatusID(otherHashPartition);

return ComparisonChain.start()

.compare(myHashPartition, otherHashPartition)

.compare(myTimeSliceId, otherTimeSliceId)

.result();

}

}

/\*\*

\* For all the timeslices, create the corresponding SegmentInfo and store in a map

\*/

@VisibleForTesting

Map<String, SegmentBuilderSegment> createSegmentInfoMap() throws IOException {

final List<ArchiveTimeSlicer.ArchiveTimeSlice> timeSlices = createTimeSlices();

List<TimeSliceAndHashPartition> timeSlicePairs = createPairs(timeSlices);

// Export how many segments should be built

SEGMENTS\_TO\_BUILD.set(timeSlicePairs.size());

LOG.info("Total number of segments to be built across all segment builders: {}",

timeSlicePairs.size());

List<TimeSliceAndHashPartition> mySegments = getSegmentsForMyPartition(timeSlicePairs);

Map<String, SegmentBuilderSegment> segmentInfoMap = new HashMap<>();

for (TimeSliceAndHashPartition mySegment : mySegments) {

ArchiveSegment segment = new ArchiveSegment(mySegment.timeSlice, mySegment.hashPartition,

EarlybirdConfig.getMaxSegmentSize());

SegmentInfo segmentInfo = new SegmentInfo(segment, segmentFactory, segmentSyncConfig);

segmentInfoMap.put(segmentInfo.getSegment().getSegmentName(), new NotYetBuiltSegment(

segmentInfo, segmentConfig, segmentFactory, 0, segmentSyncConfig));

}

return segmentInfoMap;

}

private List<TimeSliceAndHashPartition> createPairs(

List<ArchiveTimeSlicer.ArchiveTimeSlice> timeSlices) {

List<TimeSliceAndHashPartition> timeSlicePairs = new ArrayList<>();

for (ArchiveTimeSlicer.ArchiveTimeSlice slice : timeSlices) {

List<Integer> localPartitions = hashPartitions;

if (localPartitions == null) {

localPartitions = range(slice.getNumHashPartitions());

}

for (Integer partition : localPartitions) {

timeSlicePairs.add(new TimeSliceAndHashPartition(slice, partition));

}

}

return timeSlicePairs;

}

private List<TimeSliceAndHashPartition> getSegmentsForMyPartition(

List<TimeSliceAndHashPartition> timeSlicePairs) {

Collections.sort(timeSlicePairs);

List<TimeSliceAndHashPartition> myTimeSlices = new ArrayList<>();

for (int i = myPartitionId; i < timeSlicePairs.size(); i += numSegmentBuilderPartitions) {

myTimeSlices.add(timeSlicePairs.get(i));

}

LOG.info("Getting segments to be built for partition: {}", myPartitionId);

LOG.info("Total number of partitions: {}", numSegmentBuilderPartitions);

LOG.info("Number of segments picked: {}", myTimeSlices.size());

return myTimeSlices;

}

/\*\*

\* Print out the segmentInfo Map for debugging

\*/

private void printSegmentInfoMap(Map<String, SegmentBuilderSegment> segmentInfoMap) {

LOG.info("SegmentInfoMap: ");

for (Map.Entry<String, SegmentBuilderSegment> entry : segmentInfoMap.entrySet()) {

LOG.info(entry.getValue().toString());

}

LOG.info("Total SegmentInfoMap size: " + segmentInfoMap.size() + ". done.");

}

/\*\*

\* Build indices or refresh state for the segments in the specified segmentInfoMap, which only

\* contains the segments that need to build or are building. When a segment has not been built,

\* it is built here. If built successfully, it will be removed from the map; otherwise, its

\* state will be updated in the map.

\*

\* Returns true iff this process has built a segment.

\*/

@VisibleForTesting

boolean processSegments(Map<String, SegmentBuilderSegment> segmentInfoMap)

throws SegmentInfoConstructionException, SegmentUpdaterException, InterruptedException {

boolean hasBuiltSegment = false;

Iterator<Map.Entry<String, SegmentBuilderSegment>> iter =

segmentInfoMap.entrySet().iterator();

while (iter.hasNext()) {

Map.Entry<String, SegmentBuilderSegment> entry = iter.next();

SegmentBuilderSegment originalSegment = entry.getValue();

LOG.info("About to process segment: {}", originalSegment.getSegmentName());

long startMillis = System.currentTimeMillis();

SegmentBuilderSegment updatedSegment = segmentHandler.processSegment(originalSegment);

if (updatedSegment.isBuilt()) {

iter.remove();

hasBuiltSegment = true;

if (originalSegment instanceof NotYetBuiltSegment) {

// Record the total time spent on successfully building a semgent, used to compute the

// average segment building time.

long timeSpent = System.currentTimeMillis() - startMillis;

segmentsBuiltLocally.increment();

timeSpentOnSuccessfulBuildSecs.add(timeSpent / 1000);

}

} else {

entry.setValue(updatedSegment);

}

clock.waitFor(getSegmentSleepTime());

}

return hasBuiltSegment;

}

private long getSegmentSleepTime() {

// The Hadoop name node can handle only about 200 requests/sec before it gets overloaded.

// Updating the state of a node that has been built takes about 1 second. In the worst case

// scenario with 800 segment builders, we end up with about 800 requests/sec. Adding a 10

// second sleep lowers the worst case to about 80 requests/sec.

long sleepMillis = TimeUnit.SECONDS.toMillis(waitBetweenSegmentsSecs);

// Use randomization so that we can't get all segment builders hitting it at the exact same time

int lowerSleepBoundMillis = (int) (sleepMillis \* (1.0 - SLEEP\_RANDOMIZATION\_RATIO));

int upperSleepBoundMillis = (int) (sleepMillis \* (1.0 + SLEEP\_RANDOMIZATION\_RATIO));

return randRange(lowerSleepBoundMillis, upperSleepBoundMillis);

}

/\*\*

\* Returns a pseudo-random number between min and max, inclusive.

\*/

private int randRange(int min, int max) {

return random.nextInt((max - min) + 1) + min;

}

/\*\*

\* Returns list of integers 0, 1, 2, ..., count-1.

\*/

private static List<Integer> range(int count) {

List<Integer> nums = new ArrayList<>(count);

for (int i = 0; i < count; i++) {

nums.add(i);

}

return nums;

}

private static SegmentSyncConfig getSyncConfig(String scrubGen) {

if (scrubGen == null || scrubGen.isEmpty()) {

throw new RuntimeException(

"Scrub gen expected, but could not get it from the arguments.");

}

LOG.info("Scrub gen: " + scrubGen);

return new SegmentSyncConfig(Optional.of(scrubGen));

}

}