package com.twitter.search.earlybird.partition;

import java.io.File;

import java.io.IOException;

import java.io.OutputStreamWriter;

import java.text.DateFormat;

import java.text.ParseException;

import java.text.SimpleDateFormat;

import java.time.Duration;

import java.util.ArrayList;

import java.util.Date;

import java.util.SortedMap;

import java.util.TreeMap;

import java.util.concurrent.TimeoutException;

import scala.runtime.BoxedUnit;

import com.google.common.base.Preconditions;

import org.apache.commons.compress.utils.Lists;

import org.apache.commons.lang.RandomStringUtils;

import org.apache.hadoop.fs.FSDataOutputStream;

import org.apache.hadoop.fs.FileStatus;

import org.apache.hadoop.fs.FileSystem;

import org.apache.hadoop.fs.Path;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.common.util.Clock;

import com.twitter.search.common.config.Config;

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

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

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

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

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

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

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

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

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

/\*\*

\* Flushes an EarlybirdIndex to HDFS, so that when Earlybird starts, it can read the index from

\* HDFS instead of indexing from scratch.

\*

\* The path looks like:

\* /smf1/rt2/user/search/earlybird/loadtest/realtime/indexes/flush\_version\_158/partition\_8/index\_2020\_02\_25\_02

\*/

public class EarlybirdIndexFlusher {

public enum FlushAttemptResult {

CHECKED\_RECENTLY,

FOUND\_INDEX,

FLUSH\_ATTEMPT\_MADE,

FAILED\_LOCK\_ATTEMPT,

HADOOP\_TIMEOUT

}

@FunctionalInterface

public interface PostFlushOperation {

/\*\*

\* Run this after we finish flushing an index, before we rejoin the serverset.

\*/

void execute();

}

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

private static final SearchCounter FLUSH\_SUCCESS\_COUNTER =

SearchCounter.export("successfully\_flushed\_index");

public static final String TWEET\_KAFKA\_OFFSET = "tweet\_kafka\_offset";

public static final String UPDATE\_KAFKA\_OFFSET = "update\_kafka\_offset";

public static final String FLUSHED\_FROM\_REPLICA = "flushed\_from\_replica";

public static final String SEGMENTS = "segments";

public static final String TIMESLICE\_ID = "timeslice\_id";

public static final String DATA\_SUFFIX = ".data";

public static final String INFO\_SUFFIX = ".info";

public static final String INDEX\_INFO = "earlybird\_index.info";

private static final String INDEX\_PATH\_FORMAT = "%s/flush\_version\_%d/partition\_%d";

public static final DateFormat INDEX\_DATE\_SUFFIX = new SimpleDateFormat("yyyy\_MM\_dd\_HH");

public static final String INDEX\_PREFIX = "index\_";

public static final String TMP\_PREFIX = "tmp\_";

// Check if we need to flush every five minutes.

private static final long FLUSH\_CHECK\_PERIOD = Duration.ofMinutes(5).toMillis();

// Make sure we don't keep more than 3 copies of the index in HDFS, so that we don't run out of

// HDFS space.

private static final int INDEX\_COPIES = 3;

private static final NonPagingAssert FLUSHING\_TOO\_MANY\_NON\_OPTIMIZED\_SEGMENTS =

new NonPagingAssert("flushing\_too\_many\_non\_optimized\_segments");

private final CoordinatedEarlybirdActionInterface actionCoordinator;

private final FileSystem fileSystem;

private final Path indexPath;

private final Clock clock;

private final SegmentManager segmentManager;

private final int replicaId;

private final TimeLimitedHadoopExistsCall timeLimitedHadoopExistsCall;

private final OptimizationAndFlushingCoordinationLock optimizationAndFlushingCoordinationLock;

private long checkedAt = 0;

public EarlybirdIndexFlusher(

CoordinatedEarlybirdActionInterface actionCoordinator,

FileSystem fileSystem,

String indexHDFSPath,

SegmentManager segmentManager,

PartitionConfig partitionConfig,

Clock clock,

TimeLimitedHadoopExistsCall timeLimitedHadoopExistsCall,

OptimizationAndFlushingCoordinationLock optimizationAndFlushingCoordinationLock

) {

this.actionCoordinator = actionCoordinator;

this.fileSystem = fileSystem;

this.indexPath = buildPathToIndexes(indexHDFSPath, partitionConfig);

this.segmentManager = segmentManager;

this.clock = clock;

this.replicaId = partitionConfig.getHostPositionWithinHashPartition();

this.timeLimitedHadoopExistsCall = timeLimitedHadoopExistsCall;

this.optimizationAndFlushingCoordinationLock = optimizationAndFlushingCoordinationLock;

}

/\*\*

\* Periodically checks if an index needs to be uploaded to HDFS, and uploads it if necessary.

\* Skips flush if unable to acquire the optimizationAndFlushingCoordinationLock.

\*/

public FlushAttemptResult flushIfNecessary(

long tweetOffset,

long updateOffset,

PostFlushOperation postFlushOperation) throws Exception {

long now = clock.nowMillis();

if (now - checkedAt < FLUSH\_CHECK\_PERIOD) {

return FlushAttemptResult.CHECKED\_RECENTLY;

}

checkedAt = now;

// Try to aqcuire lock to ensure that we are not in the gc\_before\_optimization or the

// post\_optimization\_rebuilds step of optimization. If the lock is not available, then skip

// flushing.

if (!optimizationAndFlushingCoordinationLock.tryLock()) {

return FlushAttemptResult.FAILED\_LOCK\_ATTEMPT;

}

// Acquired the lock, so wrap the flush in a try/finally block to ensure we release the lock

try {

Path flushPath = pathForHour();

try {

// If this doesn't execute on time, it will throw an exception and this function

// finishes its execution.

boolean result = timeLimitedHadoopExistsCall.exists(flushPath);

if (result) {

return FlushAttemptResult.FOUND\_INDEX;

}

} catch (TimeoutException e) {

LOG.warn("Timeout while calling hadoop", e);

return FlushAttemptResult.HADOOP\_TIMEOUT;

}

boolean flushedIndex = false;

try {

// this function returns a boolean.

actionCoordinator.execute("index\_flushing", isCoordinated ->

flushIndex(flushPath, isCoordinated, tweetOffset, updateOffset, postFlushOperation));

flushedIndex = true;

} catch (CoordinatedEarlybirdActionLockFailed e) {

// This only happens when we fail to grab the lock, which is fine because another Earlybird

// is already working on flushing this index, so we don't need to.

LOG.debug("Failed to grab lock", e);

}

if (flushedIndex) {

// We don't return with a guarantee that we actually flushed something. It's possible

// that the .execute() function above was not able to leave the server set to flush.

return FlushAttemptResult.FLUSH\_ATTEMPT\_MADE;

} else {

return FlushAttemptResult.FAILED\_LOCK\_ATTEMPT;

}

} finally {

optimizationAndFlushingCoordinationLock.unlock();

}

}

/\*\*

\* Create a subpath to the directory with many indexes in it. Will have an index for each hour.

\*/

public static Path buildPathToIndexes(String root, PartitionConfig partitionConfig) {

return new Path(String.format(

INDEX\_PATH\_FORMAT,

root,

FlushVersion.CURRENT\_FLUSH\_VERSION.getVersionNumber(),

partitionConfig.getIndexingHashPartitionID()));

}

/\*\*

\* Returns a sorted map from the unix time in millis an index was flushed to the path of an index.

\* The last element will be the path of the most recent index.

\*/

public static SortedMap<Long, Path> getIndexPathsByTime(

Path indexPath,

FileSystem fileSystem

) throws IOException, ParseException {

LOG.info("Getting index paths from file system: {}", fileSystem.getUri().toASCIIString());

SortedMap<Long, Path> pathByTime = new TreeMap<>();

Path globPattern = indexPath.suffix("/" + EarlybirdIndexFlusher.INDEX\_PREFIX + "\*");

LOG.info("Lookup glob pattern: {}", globPattern);

for (FileStatus indexDir : fileSystem.globStatus(globPattern)) {

String name = new File(indexDir.getPath().toString()).getName();

String dateString = name.substring(EarlybirdIndexFlusher.INDEX\_PREFIX.length());

Date date = EarlybirdIndexFlusher.INDEX\_DATE\_SUFFIX.parse(dateString);

pathByTime.put(date.getTime(), indexDir.getPath());

}

LOG.info("Found {} files matching the pattern.", pathByTime.size());

return pathByTime;

}

private boolean flushIndex(

Path flushPath,

boolean isCoordinated,

long tweetOffset,

long updateOffset,

PostFlushOperation postFlushOperation

) throws Exception {

Preconditions.checkState(isCoordinated);

if (fileSystem.exists(flushPath)) {

return false;

}

LOG.info("Starting index flush");

// In case the process is killed suddenly, we wouldn't be able to clean up the temporary

// directory, and we don't want other processes to reuse it, so add some randomness.

Path tmpPath = indexPath.suffix("/" + TMP\_PREFIX + RandomStringUtils.randomAlphabetic(8));

boolean creationSucceed = fileSystem.mkdirs(tmpPath);

if (!creationSucceed) {

throw new IOException("Couldn't create HDFS directory at " + flushPath);

}

LOG.info("Temp path: {}", tmpPath);

try {

ArrayList<SegmentInfo> segmentInfos = Lists.newArrayList(segmentManager.getSegmentInfos(

SegmentManager.Filter.Enabled, SegmentManager.Order.NEW\_TO\_OLD).iterator());

segmentManager.logState("Before flushing");

EarlybirdIndex index = new EarlybirdIndex(segmentInfos, tweetOffset, updateOffset);

ActionLogger.run(

"Flushing index to " + tmpPath,

() -> flushIndex(tmpPath, index));

} catch (Exception e) {

LOG.error("Exception while flushing index. Rethrowing.");

if (fileSystem.delete(tmpPath, true)) {

LOG.info("Successfully deleted temp output");

} else {

LOG.error("Couldn't delete temp output");

}

throw e;

}

// We flush it to a temporary directory, then rename the temporary directory so that it the

// change is atomic, and other Earlybirds will either see the old indexes, or the new, complete

// index, but never an in progress index.

boolean renameSucceeded = fileSystem.rename(tmpPath, flushPath);

if (!renameSucceeded) {

throw new IOException("Couldn't rename HDFS from " + tmpPath + " to " + flushPath);

}

LOG.info("Flushed index to {}", flushPath);

cleanupOldIndexes();

FLUSH\_SUCCESS\_COUNTER.increment();

LOG.info("Executing post flush operation...");

postFlushOperation.execute();

return true;

}

private void cleanupOldIndexes() throws Exception {

LOG.info("Looking up whether we need to clean up old indexes...");

SortedMap<Long, Path> pathsByTime =

EarlybirdIndexFlusher.getIndexPathsByTime(indexPath, fileSystem);

while (pathsByTime.size() > INDEX\_COPIES) {

Long key = pathsByTime.firstKey();

Path oldestHourPath = pathsByTime.remove(key);

LOG.info("Deleting old index at path '{}'.", oldestHourPath);

if (fileSystem.delete(oldestHourPath, true)) {

LOG.info("Successfully deleted old index");

} else {

LOG.error("Couldn't delete old index");

}

}

}

private Path pathForHour() {

Date date = new Date(clock.nowMillis());

String time = INDEX\_DATE\_SUFFIX.format(date);

return indexPath.suffix("/" + INDEX\_PREFIX + time);

}

private void flushIndex(Path flushPath, EarlybirdIndex index) throws Exception {

int numOfNonOptimized = index.numOfNonOptimizedSegments();

if (numOfNonOptimized > EarlybirdIndex.MAX\_NUM\_OF\_NON\_OPTIMIZED\_SEGMENTS) {

LOG.error(

"Found {} non-optimized segments when flushing to disk!", numOfNonOptimized);

FLUSHING\_TOO\_MANY\_NON\_OPTIMIZED\_SEGMENTS.assertFailed();

}

int numSegments = index.getSegmentInfoList().size();

int flushingThreadPoolSize = numSegments;

if (Config.environmentIsTest()) {

// SEARCH-33763: Limit the thread pool size for tests to avoid using too much memory on scoot.

flushingThreadPoolSize = 2;

}

LOG.info("Flushing index using a thread pool size of {}", flushingThreadPoolSize);

ParallelUtil.parmap("flush-index", flushingThreadPoolSize, si -> ActionLogger.call(

"Flushing segment " + si.getSegmentName(),

() -> flushSegment(flushPath, si)), index.getSegmentInfoList());

FlushInfo indexInfo = new FlushInfo();

indexInfo.addLongProperty(UPDATE\_KAFKA\_OFFSET, index.getUpdateOffset());

indexInfo.addLongProperty(TWEET\_KAFKA\_OFFSET, index.getTweetOffset());

indexInfo.addIntProperty(FLUSHED\_FROM\_REPLICA, replicaId);

FlushInfo segmentFlushInfos = indexInfo.newSubProperties(SEGMENTS);

for (SegmentInfo segmentInfo : index.getSegmentInfoList()) {

FlushInfo segmentFlushInfo = segmentFlushInfos.newSubProperties(segmentInfo.getSegmentName());

segmentFlushInfo.addLongProperty(TIMESLICE\_ID, segmentInfo.getTimeSliceID());

}

Path indexInfoPath = flushPath.suffix("/" + INDEX\_INFO);

try (FSDataOutputStream infoOutputStream = fileSystem.create(indexInfoPath)) {

OutputStreamWriter infoFileWriter = new OutputStreamWriter(infoOutputStream);

FlushInfo.flushAsYaml(indexInfo, infoFileWriter);

}

}

private BoxedUnit flushSegment(Path flushPath, SegmentInfo segmentInfo) throws Exception {

Path segmentPrefix = flushPath.suffix("/" + segmentInfo.getSegmentName());

Path segmentPath = segmentPrefix.suffix(DATA\_SUFFIX);

FlushInfo flushInfo = new FlushInfo();

try (FSDataOutputStream outputStream = fileSystem.create(segmentPath)) {

DataSerializer out = new DataSerializer(segmentPath.toString(), outputStream);

segmentInfo.getIndexSegment().flush(flushInfo, out);

}

Path infoPath = segmentPrefix.suffix(INFO\_SUFFIX);

try (FSDataOutputStream infoOutputStream = fileSystem.create(infoPath)) {

OutputStreamWriter infoFileWriter = new OutputStreamWriter(infoOutputStream);

FlushInfo.flushAsYaml(flushInfo, infoFileWriter);

}

return BoxedUnit.UNIT;

}

}