package com.twitter.search.earlybird.partition;

import java.io.File;

import java.io.IOException;

import java.util.concurrent.TimeUnit;

import org.apache.commons.io.FileUtils;

import org.apache.hadoop.fs.FileSystem;

import org.apache.hadoop.fs.Path;

import org.apache.lucene.store.Directory;

import org.apache.lucene.store.FSDirectory;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.common.base.Command;

import com.twitter.common.quantity.Amount;

import com.twitter.common.quantity.Time;

import com.twitter.search.common.database.DatabaseConfig;

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

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

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

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

/\*\*

\* Flush segments to disk and upload them to HDFS.

\*/

public class SegmentHdfsFlusher {

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

private static final Amount<Long, Time> HDFS\_UPLOADER\_TRY\_LOCK\_NODE\_EXPIRATION\_TIME\_MILLIS =

Amount.of(1L, Time.HOURS);

private final SegmentSyncConfig sync;

private final boolean holdLockWhileUploading;

private final ZooKeeperTryLockFactory zkTryLockFactory;

public SegmentHdfsFlusher(ZooKeeperTryLockFactory zooKeeperTryLockFactory,

SegmentSyncConfig sync,

boolean holdLockWhileUploading) {

this.zkTryLockFactory = zooKeeperTryLockFactory;

this.sync = sync;

this.holdLockWhileUploading = holdLockWhileUploading;

}

public SegmentHdfsFlusher(

ZooKeeperTryLockFactory zooKeeperTryLockFactory,

SegmentSyncConfig sync) {

this(zooKeeperTryLockFactory, sync, true);

}

private boolean shouldFlushSegment(SegmentInfo segmentInfo) {

return segmentInfo.isEnabled()

&& !segmentInfo.getSyncInfo().isFlushed()

&& segmentInfo.isComplete()

&& segmentInfo.isOptimized()

&& !segmentInfo.isFailedOptimize()

&& !segmentInfo.getSyncInfo().isLoaded();

}

/\*\*

\* Flushes a segment to local disk and to HDFS.

\*/

public boolean flushSegmentToDiskAndHDFS(SegmentInfo segmentInfo) {

if (!shouldFlushSegment(segmentInfo)) {

return false;

}

try {

if (segmentInfo.isIndexing()) {

LOG.error("Tried to flush current segment!");

return false;

}

// Check-and-set the beingUploaded flag from false to true. If the CAS fails, it means the

// segment is being flushed already, or being deleted. In this case, we can just return false.

if (!segmentInfo.casBeingUploaded(false, true)) {

LOG.warn("Tried to flush a segment that's being flushed or deleted.");

return false;

}

// At this point, the above CAS must have returned false. This mean the beingUploaded flag

// was false, and set to true now. We can proceed with flushing the segment.

try {

checkAndFlushSegmentToHdfs(segmentInfo);

} finally {

segmentInfo.setBeingUploaded(false);

}

return true;

} catch (Exception e) {

LOG.error("Exception while flushing IndexSegment to "

+ segmentInfo.getSyncInfo().getHdfsFlushDir(), e);

return false;

}

}

/\*\*

\* First try to acquire a lock in Zookeeper for this segment, so multiple Earlybirds in the same

\* partition don't flush or upload the segment at the same time. When the lock is acquired, check

\* for the segment in HDFS. If the data already exists, don't flush to disk.

\*/

private void checkAndFlushSegmentToHdfs(final SegmentInfo segment) {

LOG.info("Checking and flushing segment {}", segment);

try {

// Always flush the segment locally.

Directory dir = FSDirectory.open(createFlushDir(segment).toPath());

segment.flush(dir);

LOG.info("Completed local flush of segment {}. Flush to HDFS enabled: {}",

segment, sync.isFlushToHdfsEnabled());

} catch (IOException e) {

LOG.error("Failed to flush segment " + segment + " locally", e);

return;

}

if (!holdLockWhileUploading) {

flushToHdfsIfNecessary(segment);

} else {

TryLock lock = zkTryLockFactory.createTryLock(

DatabaseConfig.getLocalHostname(),

sync.getZooKeeperSyncFullPath(),

sync.getVersionedName(segment.getSegment()),

HDFS\_UPLOADER\_TRY\_LOCK\_NODE\_EXPIRATION\_TIME\_MILLIS

);

boolean gotLock = lock.tryWithLock((Command) () -> flushToHdfsIfNecessary(segment));

if (!gotLock) {

LOG.info("Failed to get zk upload lock for segment {}", segment);

}

}

}

/\*\*

\* Check whether the segment has already been flushed to HDFS. If not, flush the segment to disk

\* and upload the files to HDFS.

\*

\* If the ZK lock isn't used, there is a race between the existence check and the upload (in

\* which another Earlybird can sneak in and upload the segment), so we will potentially upload

\* the same segment from different hosts. Thus, the Earlybird hostname is part of the segment's

\* path on HDFS.

\*/

private void flushToHdfsIfNecessary(SegmentInfo segmentInfo) {

Timer timer = new Timer(TimeUnit.MILLISECONDS);

String status = "flushed";

try (FileSystem fs = HdfsUtil.getHdfsFileSystem()) {

// If we can't load segments from HDFS, don't bother checking HDFS for the segment

if (sync.isSegmentLoadFromHdfsEnabled()

&& (segmentInfo.getSyncInfo().isFlushed()

|| HdfsUtil.segmentExistsOnHdfs(fs, segmentInfo))) {

status = "existing";

} else if (sync.isFlushToHdfsEnabled()) {

copyLocalFilesToHdfs(fs, segmentInfo);

status = "uploaded";

}

// whether we uploaded, or someone else did, this segment should now be on HDFS. If

// uploading to HDFS is disabled, we still consider it complete.

segmentInfo.getSyncInfo().setFlushed(true);

} catch (IOException e) {

LOG.error("Failed copying segment {} to HDFS after {} ms", segmentInfo, timer.stop(), e);

status = "exception";

} finally {

if (timer.running()) {

timer.stop();

}

LOG.info("Flush of segment {} to HDFS completed in {} milliseconds. Status: {}",

segmentInfo, timer.getElapsed(), status);

}

}

/\*\*

\* Copy local segment files to HDFS. Files are first copied into a temporary directory

\* in the form <hostname>\_<segmentname> and when all the files are written out to HDFS,

\* the dir is renamed to <segmentname>\_<hostname>, where it is accessible to other Earlybirds.

\*/

private void copyLocalFilesToHdfs(FileSystem fs, SegmentInfo segment) throws IOException {

String hdfsTempBaseDir = segment.getSyncInfo().getHdfsTempFlushDir();

// If the temp dir already exists on HDFS, a prior flush must have been interrupted.

// Delete it and start fresh.

removeHdfsTempDir(fs, hdfsTempBaseDir);

for (String fileName : sync.getAllSyncFileNames(segment)) {

String hdfsFileName = hdfsTempBaseDir + "/" + fileName;

String localBaseDir = segment.getSyncInfo().getLocalSyncDir();

String localFileName = localBaseDir + "/" + fileName;

LOG.debug("About to start copying {} to HDFS, from {} to {}",

fileName, localFileName, hdfsFileName);

Timer timer = new Timer(TimeUnit.MILLISECONDS);

fs.copyFromLocalFile(new Path(localFileName), new Path(hdfsFileName));

LOG.debug("Completed copying {} to HDFS, from {} to {}, in {} ms",

fileName, localFileName, hdfsFileName, timer.stop());

}

// now let's rename the dir into its proper form.

String hdfsBaseDir = segment.getSyncInfo().getHdfsFlushDir();

if (fs.rename(new Path(hdfsTempBaseDir), new Path(hdfsBaseDir))) {

LOG.info("Renamed segment dir on HDFS from {} to {}", hdfsTempBaseDir, hdfsBaseDir);

} else {

String errorMessage = String.format("Failed to rename segment dir on HDFS from %s to %s",

hdfsTempBaseDir, hdfsBaseDir);

LOG.error(errorMessage);

removeHdfsTempDir(fs, hdfsTempBaseDir);

// Throw an IOException so the calling code knows that the copy failed

throw new IOException(errorMessage);

}

}

private void removeHdfsTempDir(FileSystem fs, String tempDir) throws IOException {

Path tempDirPath = new Path(tempDir);

if (fs.exists(tempDirPath)) {

LOG.info("Found existing temporary flush dir {} on HDFS, removing", tempDir);

if (!fs.delete(tempDirPath, true /\* recursive \*/)) {

LOG.error("Failed to delete temp dir {}", tempDir);

}

}

}

// Create or replace the local flush directory

private File createFlushDir(SegmentInfo segmentInfo) throws IOException {

final String flushDirStr = segmentInfo.getSyncInfo().getLocalSyncDir();

File flushDir = new File(flushDirStr);

if (flushDir.exists()) {

// Delete just the flushed persistent files if they are there.

// We may also have the lucene on-disk indexed in the same dir here,

// that we do not want to delete.

for (String persistentFile : sync.getPersistentFileNames(segmentInfo)) {

for (String fileName : PersistentFile.getAllFileNames(persistentFile)) {

File file = new File(flushDir, fileName);

if (file.exists()) {

LOG.info("Deleting incomplete flush file {}", file.getAbsolutePath());

FileUtils.forceDelete(file);

}

}

}

return flushDir;

}

// Try to create the flush directory

if (!flushDir.mkdirs()) {

throw new IOException("Not able to create segment flush directory \"" + flushDirStr + "\"");

}

return flushDir;

}

}