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.commons.io.IOUtils;

import org.apache.hadoop.fs.FileStatus;

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.util.Clock;

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

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

import com.twitter.search.common.partitioning.snowflakeparser.SnowflakeIdParser;

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

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

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

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

public class SegmentLoader {

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

private static final SegmentSyncStats SEGMENT\_LOAD\_FROM\_HDFS\_STATS =

new SegmentSyncStats("load\_from\_hdfs");

private final CriticalExceptionHandler criticalExceptionHandler;

private final SegmentSyncConfig segmentSyncConfig;

private final Clock clock;

public SegmentLoader(SegmentSyncConfig sync,

CriticalExceptionHandler criticalExceptionHandler) {

this(sync, criticalExceptionHandler, Clock.SYSTEM\_CLOCK);

}

public SegmentLoader(SegmentSyncConfig sync,

CriticalExceptionHandler criticalExceptionHandler,

Clock clock) {

this.criticalExceptionHandler = criticalExceptionHandler;

this.segmentSyncConfig = sync;

this.clock = clock;

}

public boolean load(SegmentInfo segmentInfo) {

return downloadSegment(segmentInfo) && loadSegmentFromDisk(segmentInfo);

}

/\*\*

\* Determines if the Earlybird should attempt to download the given segment from HDFS. This

\* returns true if the segment is not already present on local disk, and the segment does exist

\* on HDFS.

\*/

public boolean shouldDownloadSegmentWhileInServerSet(SegmentInfo segmentInfo) {

if (isValidSegmentOnDisk(segmentInfo)) {

return false;

}

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

return HdfsUtil.segmentExistsOnHdfs(fs, segmentInfo);

} catch (IOException e) {

LOG.error("Failed to check HDFS for segment " + segmentInfo, e);

return false;

}

}

/\*\*

\* Verifies if the data for the given segment is present on the local disk, and if it's not,

\* downloads it from HDFS.

\*/

public boolean downloadSegment(SegmentInfo segmentInfo) {

if (!segmentInfo.isEnabled()) {

LOG.debug("Segment is disabled: " + segmentInfo);

return false;

}

if (segmentInfo.isIndexing() || segmentInfo.getSyncInfo().isLoaded()) {

LOG.debug("Cannot load indexing or loaded segment: " + segmentInfo);

return false;

}

// Return whether the appropriate version is on disk, and if not, download it from HDFS.

return isValidSegmentOnDisk(segmentInfo) || checkSegmentOnHdfsAndCopyLocally(segmentInfo);

}

/\*\*

\* Loads the data for the given segment from the local disk.

\*/

public boolean loadSegmentFromDisk(SegmentInfo segmentInfo) {

if (segmentInfo.isIndexing()) {

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

return false;

}

segmentInfo.setIndexing(true);

try {

File flushDir = new File(segmentInfo.getSyncInfo().getLocalSyncDir());

Directory loadDir = FSDirectory.open(flushDir.toPath());

segmentInfo.load(loadDir);

if (!verifySegmentStatusCountLargeEnough(segmentInfo)) {

SearchRateCounter.export(

"segment\_loader\_failed\_too\_few\_tweets\_in\_segment\_" + segmentInfo.getSegmentName())

.increment();

return false;

}

segmentInfo.setIndexing(false);

segmentInfo.setComplete(true);

segmentInfo.getSyncInfo().setLoaded(true);

return true;

} catch (FlushVersionMismatchException e) {

handleException(segmentInfo, e);

// If earlybird is in starting state, handler will terminate it

criticalExceptionHandler.handle(this, e);

} catch (Exception e) {

handleException(segmentInfo, e);

}

SearchRateCounter.export("segment\_loader\_failed\_" + segmentInfo.getSegmentName()).increment();

return false;

}

// Check to see if the segment exists on disk, and its checksum passes.

private boolean isValidSegmentOnDisk(SegmentInfo segment) {

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

File loadDir = new File(loadDirStr);

if (!loadDir.exists()) {

return false;

}

for (String persistentFileName : segmentSyncConfig.getPersistentFileNames(segment)) {

if (!verifyInfoChecksum(loadDir, persistentFileName)) {

return false;

}

}

return true;

}

private static boolean verifyInfoChecksum(File loadDir, String databaseName) {

if (checksumFileExists(loadDir, databaseName)) {

try {

Directory dir = FSDirectory.open(loadDir.toPath());

PersistentFile.Reader reader = PersistentFile.getReader(dir, databaseName);

try {

reader.verifyInfoChecksum();

return true;

} finally {

IOUtils.closeQuietly(reader);

IOUtils.closeQuietly(dir);

}

} catch (PersistentFile.CorruptFileException e) {

LOG.error("Failed checksum verification.", e);

} catch (IOException e) {

LOG.error("Error while trying to read checksum file", e);

}

}

return false;

}

// Check that the loaded segment's status count is higher than the configured threshold

private boolean verifySegmentStatusCountLargeEnough(SegmentInfo segmentInfo) {

long segmentStatusCount = segmentInfo.getIndexStats().getStatusCount();

if (segmentStatusCount > segmentSyncConfig.getMinSegmentStatusCountThreshold()) {

return true;

} else if (segmentInfo.getEarlybirdIndexConfig().isIndexStoredOnDisk()

&& couldBeMostRecentArchiveSegment(segmentInfo)) {

// The most recent archive earlybird segment is expected to be incomplete

LOG.info("Segment status count (" + segmentStatusCount + ") is below the threshold of "

+ segmentSyncConfig.getMinSegmentStatusCountThreshold()

+ ", but this is expected because the most recent segment is expected to be incomplete: "

+ segmentInfo);

return true;

} else {

// The segment status count is small so the segment is likely incomplete.

LOG.error("Segment status count (" + segmentStatusCount + ") is below the threshold of "

+ segmentSyncConfig.getMinSegmentStatusCountThreshold() + ": " + segmentInfo);

segmentInfo.setIndexing(false);

segmentInfo.getSyncInfo().setLoaded(false);

// Remove segment from local disk

if (!segmentInfo.deleteLocalIndexedSegmentDirectoryImmediately()) {

LOG.error("Failed to cleanup unloadable segment directory.");

}

return false;

}

}

// Check if this segment could be the most recent archive earlybird segment (would be on the

// latest tier). Archive segments tend to span around 12 days, so using a conservative threshold

// of 20 days.

private boolean couldBeMostRecentArchiveSegment(SegmentInfo segmentInfo) {

long timesliceAgeMs =

SnowflakeIdParser.getTweetAgeInMs(clock.nowMillis(), segmentInfo.getTimeSliceID());

return (timesliceAgeMs / 1000 / 60 / 60 / 24) <= 20;

}

/\*\*

\* Check to see if the segment exists on hdfs. Will look for the correct segment version

\* uploaded by any of the hosts.

\* If the segment exists on hdfs, the segment will be copied from hdfs to the local file

\* system, and we will verify the checksum against the copied version.

\* @return true iff the segment was copied to local disk, and the checksum is verified.

\*/

private boolean checkSegmentOnHdfsAndCopyLocally(SegmentInfo segment) {

if (!segmentSyncConfig.isSegmentLoadFromHdfsEnabled()) {

return isValidSegmentOnDisk(segment);

}

LOG.info("About to start downloading segment from hdfs: " + segment);

Timer timer = new Timer(TimeUnit.MILLISECONDS);

String status = null;

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

FileSystem fs = null;

try {

fs = HdfsUtil.getHdfsFileSystem();

String hdfsBaseDirPrefix = segment.getSyncInfo().getHdfsSyncDirPrefix();

FileStatus[] statuses = fs.globStatus(new Path(hdfsBaseDirPrefix));

if (statuses != null && statuses.length > 0) {

Path hdfsSyncPath = statuses[0].getPath();

copySegmentFilesFromHdfs(segment, segmentSyncConfig, fs, hdfsSyncPath);

status = "loaded";

} else {

LOG.info("No segments found in hdfs under: " + hdfsBaseDirPrefix);

status = "notloaded";

}

fs.close();

} catch (IOException ex) {

LOG.error("Failed copying segment from hdfs: " + segment + " after: "

+ timer.stop() + " ms", ex);

status = "exception";

SEGMENT\_LOAD\_FROM\_HDFS\_STATS.recordError();

try {

FileUtils.deleteDirectory(new File(localBaseDir));

} catch (IOException e) {

LOG.error("Error cleaning up local segment directory: " + segment, e);

}

} finally {

timer.stop();

SEGMENT\_LOAD\_FROM\_HDFS\_STATS.actionComplete(timer);

LOG.info("Download from hdfs completed in "

+ timer.getElapsed() + " milliseconds: " + segment + " status: " + status);

IOUtils.closeQuietly(fs);

}

// now check to see if we have successfully copied the segment

return isValidSegmentOnDisk(segment);

}

private static void copySegmentFilesFromHdfs(SegmentInfo segment,

SegmentSyncConfig syncConfig,

FileSystem fs,

Path hdfsSyncPath) throws IOException {

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

File localBaseDirFile = new File(localBaseDir);

FileUtils.deleteQuietly(localBaseDirFile);

if (localBaseDirFile.exists()) {

LOG.warn("Cannot delete the existing path: " + localBaseDir);

}

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

Path hdfsFilePath = new Path(hdfsSyncPath, fileName);

String localFileName = localBaseDir + "/" + fileName;

LOG.debug("About to start loading from hdfs: " + fileName + " from: "

+ hdfsFilePath + " to: " + localFileName);

Timer timer = new Timer(TimeUnit.MILLISECONDS);

fs.copyToLocalFile(hdfsFilePath, new Path(localFileName));

LOG.debug("Loaded segment file from hdfs: " + fileName + " from: "

+ hdfsFilePath + " to: " + localFileName + " in: " + timer.stop() + " ms.");

}

LOG.info("Finished downloading segments from " + hdfsSyncPath);

}

private static boolean checksumFileExists(File loadDir, String databaseName) {

String checksumFileName = PersistentFile.genChecksumFileName(databaseName);

File checksumFile = new File(loadDir, checksumFileName);

return checksumFile.exists();

}

private void handleException(SegmentInfo segmentInfo, Exception e) {

LOG.error("Exception while loading IndexSegment from "

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

segmentInfo.setIndexing(false);

segmentInfo.getSyncInfo().setLoaded(false);

if (!segmentInfo.deleteLocalIndexedSegmentDirectoryImmediately()) {

LOG.error("Failed to cleanup unloadable segment directory.");

}

}

}