package com.twitter.search.earlybird.partition;

import java.util.ArrayList;

import java.util.Collection;

import java.util.Collections;

import java.util.Date;

import java.util.Optional;

import java.util.concurrent.TimeUnit;

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

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

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

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

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

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

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

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

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

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

import com.twitter.util.TwitterDateFormat;

/\*\*

\* Encapsulates config information related to reading and writing segments to local filesystem or

\* HDFS.

\*/

public class SegmentSyncConfig {

public static final String LUCENE\_DIR\_PREFIX = "lucene\_";

private final Optional<String> scrubGen;

public SegmentSyncConfig(Optional<String> scrubGen) {

this.scrubGen = scrubGen;

String scrubGenStat = scrubGen.orElse("unset");

SearchLongGauge.export("scrub\_gen\_" + scrubGenStat).set(1);

if (scrubGen.isPresent()) {

// Export a stat for the number of days between the scrub gen date and now

SearchCustomGauge.export("scrub\_gen\_age\_in\_days", () -> {

long scrubGenMillis = ScrubGenUtil.parseScrubGenToDate(scrubGen.get()).getTime();

return TimeUnit.MILLISECONDS.toDays(System.currentTimeMillis() - scrubGenMillis);

});

}

}

/\*\*

\* Returns the file extension to be used for the current flush version.

\*/

public String getVersionFileExtension() {

return FlushVersion.CURRENT\_FLUSH\_VERSION.getVersionFileExtension();

}

/\*\*

\* Returns the threshold for how large a segment's status count must be at load time to be

\* considered valid.

\*/

public int getMinSegmentStatusCountThreshold() {

double minSegmentTweetCountProportionThreshold =

EarlybirdConfig.getDouble("min\_segment\_tweet\_count\_percentage\_threshold", 0) / 100;

return (int) (EarlybirdConfig.getMaxSegmentSize() \* minSegmentTweetCountProportionThreshold);

}

/\*\*

\* Determines if this earlybird is allowed to flush segments to HDFS.

\*/

public boolean isFlushToHdfsEnabled() {

return EarlybirdProperty.SEGMENT\_FLUSH\_TO\_HDFS\_ENABLED.get(false)

// Flush to HDFS is always disabled if FlushVersion is not official.

&& FlushVersion.CURRENT\_FLUSH\_VERSION.isOfficial();

}

/\*\*

\* Determines if this earlybird is allowed to load segments from HDFS.

\*/

public boolean isSegmentLoadFromHdfsEnabled() {

return EarlybirdProperty.SEGMENT\_LOAD\_FROM\_HDFS\_ENABLED.get(false);

}

/\*\*

\* Determines if this earlybird is allowed to delete flushed segments.

\*/

public boolean isDeleteFlushedSegmentsEnabled() {

return EarlybirdConfig.getBool("segment\_dropper\_delete\_flushed", true);

}

/\*\*

\* Returns the root of the segment directory on the local disk.

\*/

public String getLocalSegmentSyncRootDir() {

return EarlybirdConfig.getString("segment\_sync\_dir", "partitions")

+ getScrubGenFlushDirSuffix();

}

/\*\*

\* Returns the root of the segment directory on HDFS.

\*/

public String getHdfsSegmentSyncRootDir() {

return EarlybirdProperty.HDFS\_SEGMENT\_SYNC\_DIR.get("partitions")

+ getScrubGenFlushDirSuffix();

}

/\*\*

\* Returns the HDFS root directory where all segments should be uploaded.

\*/

public String getHdfsSegmentUploadRootDir() {

String hdfsSegmentUploadDir = EarlybirdProperty.HDFS\_SEGMENT\_UPLOAD\_DIR.get(null);

return hdfsSegmentUploadDir != null

? hdfsSegmentUploadDir + getScrubGenFlushDirSuffix()

: getHdfsSegmentSyncRootDir();

}

/\*\*

\* Returns the ZooKeeper path used for segment sync'ing.

\*/

public String getZooKeeperSyncFullPath() {

return EarlybirdProperty.ZK\_APP\_ROOT.get() + "/"

+ EarlybirdConfig.getString("segment\_flush\_sync\_relative\_path", "segment\_flush\_sync");

}

/\*\*

\* Returns the list of directories that should be persisted for this segment.

\*/

public Collection<String> getPersistentFileNames(SegmentInfo segment) {

return Collections.singleton(segment.getSegmentName());

}

/\*\*

\* Returns the list of all files that should be sync'ed for this segment.

\*/

public Collection<String> getAllSyncFileNames(SegmentInfo segment) {

Collection<String> allFileNames = PersistentFile.getAllFileNames(segment.getSegmentName());

if (segment.getEarlybirdIndexConfig().isIndexStoredOnDisk()) {

allFileNames = new ArrayList<>(allFileNames);

// Just the file name, not the full path

allFileNames.add(getLocalLuceneSyncDirFileName(segment.getSegment()));

}

return allFileNames;

}

/\*\*

\* Returns the local sync directory for the given segment.

\*/

public String getLocalSyncDirName(Segment segment) {

return getLocalSegmentSyncRootDir() + "/" + segment.getSegmentName()

+ getVersionFileExtension();

}

/\*\*

\* Returns the local Lucene directory for the given segment.

\*/

public String getLocalLuceneSyncDirName(Segment segment) {

return getLocalSyncDirName(segment) + "/" + getLocalLuceneSyncDirFileName(segment);

}

/\*\*

\* Returns the name (not the path) of the Lucene directory for the given segment.

\*/

private String getLocalLuceneSyncDirFileName(Segment segment) {

if (segment instanceof ArchiveSegment) {

Date endDate = ((ArchiveSegment) segment).getDataEndDate();

String endDateString = TwitterDateFormat.apply("yyyyMMdd").format(endDate);

return LUCENE\_DIR\_PREFIX + endDateString;

} else {

return LUCENE\_DIR\_PREFIX + "realtime";

}

}

/\*\*

\* Returns the HDFS sync directory for the given segment.

\*/

public String getHdfsSyncDirNamePrefix(Segment segment) {

return getHdfsSegmentSyncRootDir() + "/" + segment.getSegmentName()

+ getVersionFileExtension() + "\*";

}

/\*\*

\* Returns the prefix of the HDFS directory where the files for this segment should be uploaded.

\*/

public String getHdfsUploadDirNamePrefix(Segment segment) {

return getHdfsSegmentUploadRootDir() + "/" + segment.getSegmentName()

+ getVersionFileExtension() + "\*";

}

/\*\*

\* Returns the HDFS directory where the files for this segment should be uploaded.

\*/

public String getHdfsFlushDirName(Segment segment) {

return getHdfsSegmentUploadRootDir() + "/" + segment.getSegmentName()

+ getVersionFileExtension() + "\_" + DatabaseConfig.getLocalHostname();

}

/\*\*

\* Returns a temp HDFS directory to be used for this segment.

\*/

public String getHdfsTempFlushDirName(Segment segment) {

return getHdfsSegmentUploadRootDir() + "/temp\_"

+ DatabaseConfig.getLocalHostname() + "\_" + segment.getSegmentName()

+ getVersionFileExtension();

}

/\*\*

\* Concatenates the name of this segment with the flush version extension.

\*/

public String getVersionedName(Segment segment) {

return segment.getSegmentName() + getVersionFileExtension();

}

private String getScrubGenFlushDirSuffix() {

return scrubGen

.map(s -> "/scrubbed/" + s)

.orElse("");

}

/\*\*

\* Returns the scrub gen set for this earlybird.

\*/

public Optional<String> getScrubGen() {

return scrubGen;

}

}