package com.twitter.search.earlybird.archive;

import java.io.FileNotFoundException;

import java.io.IOException;

import java.util.Comparator;

import java.util.Date;

import java.util.List;

import java.util.concurrent.TimeUnit;

import java.util.regex.Matcher;

import java.util.regex.Pattern;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Function;

import com.google.common.base.Predicate;

import com.google.common.collect.ComparisonChain;

import com.google.common.collect.Lists;

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.hadoop.fs.PathFilter;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

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

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

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

import com.twitter.search.common.schema.thriftjava.ThriftIndexingEvent;

import com.twitter.search.common.util.date.DateUtil;

import com.twitter.search.common.util.io.EmptyRecordReader;

import com.twitter.search.common.util.io.LzoThriftBlockFileReader;

import com.twitter.search.common.util.io.MergingSortedRecordReader;

import com.twitter.search.common.util.io.TransformingRecordReader;

import com.twitter.search.common.util.io.recordreader.RecordReader;

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

import com.twitter.search.earlybird.document.DocumentFactory;

import com.twitter.search.earlybird.document.TweetDocument;

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

/\*\*

\* A batch of pre-processed tweets for a single hash partition from a particular day.

\*/

public class PartitionedBatch {

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

private static final Date START\_DATE\_INCLUSIVE = DateUtil.toDate(2006, 03, 21);

private static final String STATUS\_COUNT\_FILE\_PREFIX = "\_status\_count\_";

private static final Pattern STATUS\_COUNT\_FILE\_PATTERN =

Pattern.compile(STATUS\_COUNT\_FILE\_PREFIX + "(\\d+)\_minid\_(\\d+)\_maxid\_(\\d+)");

private static final int MAXIMUM\_OUT\_OF\_ORDER\_TOLERANCE\_HOURS =

EarlybirdConfig.getInt("archive\_max\_out\_of\_order\_tolerance\_hours", 12);

private static final int READER\_INIT\_IOEXCEPTION\_RETRIES = 20;

private static final PathFilter LZO\_DATA\_FILES\_FILTER = file -> file.getName().endsWith(".lzo");

private static final PathFilter TXT\_DATA\_FILES\_FILTER = file -> file.getName().endsWith(".txt");

private static final Comparator<ThriftIndexingEvent> DESC\_THRIFT\_INDEXING\_EVENT\_COMPARATOR =

(o1, o2) -> ComparisonChain.start()

.compare(o2.getSortId(), o1.getSortId())

.compare(o2.getUid(), o1.getUid())

.result();

// Number archive tweets skipped because they are too out-of-order.

private static final SearchCounter OUT\_OF\_ORDER\_STATUSES\_SKIPPED =

SearchCounter.export("out\_of\_order\_archive\_statuses\_skipped");

@VisibleForTesting

protected static final long MAXIMUM\_OUT\_OF\_ORDER\_TOLERANCE\_MILLIS =

TimeUnit.HOURS.toMillis(MAXIMUM\_OUT\_OF\_ORDER\_TOLERANCE\_HOURS);

private final Date date;

private final Path path;

private int statusCount;

private long minStatusID;

private long maxStatusID;

private final int hashPartitionID;

private boolean hasStatusCountFile;

private final int numHashPartitions;

@VisibleForTesting

public PartitionedBatch(

Path path,

int hashPartitionID,

int numHashPartitions,

Date date) {

this.path = path;

this.hashPartitionID = hashPartitionID;

this.numHashPartitions = numHashPartitions;

this.date = date;

}

/\*\*

\* Loads all the information (tweet count, etc.) for this partition and day from HDFS.

\*/

public void load(FileSystem hdfs) throws IOException {

FileStatus[] dailyBatchFiles = null;

try {

// listStatus() javadoc says it throws FileNotFoundException when path does not exist.

// However, the actual implementations return null or an empty array instead.

// We handle all 3 cases: null, empty array, or FileNotFoundException.

dailyBatchFiles = hdfs.listStatus(path);

} catch (FileNotFoundException e) {

// don't do anything here and the day will be handled as empty.

}

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

for (FileStatus file : dailyBatchFiles) {

String fileName = file.getPath().getName();

if (fileName.equals(STATUS\_COUNT\_FILE\_PREFIX)) {

// zero tweets in this partition - this can happen for early days in 2006

handleEmptyPartition();

} else {

Matcher matcher = STATUS\_COUNT\_FILE\_PATTERN.matcher(fileName);

if (matcher.matches()) {

try {

statusCount = Integer.parseInt(matcher.group(1));

// Only adjustMinStatusId in production. For tests, this makes the tests harder to

// understand.

minStatusID = Config.environmentIsTest() ? Long.parseLong(matcher.group(2))

: adjustMinStatusId(Long.parseLong(matcher.group(2)), date);

maxStatusID = Long.parseLong(matcher.group(3));

hasStatusCountFile = true;

} catch (NumberFormatException e) {

// invalid file - ignore

LOG.warn("Could not parse status count file name.", e);

}

}

}

}

} else {

// Partition folder does not exist. This case can happen for early days of twitter

// where some partitions are empty. Set us to having a status count file, the validity of

// the parent DailyStatusBatch will still be determined by whether there was a \_SUCCESS file

// in the day root.

handleEmptyPartition();

if (date.after(getEarliestDenseDay())) {

LOG.error("Unexpected empty directory {} for {}", path, date);

}

}

}

private void handleEmptyPartition() {

statusCount = 0;

minStatusID = DailyStatusBatch.EMPTY\_BATCH\_STATUS\_ID;

maxStatusID = DailyStatusBatch.EMPTY\_BATCH\_STATUS\_ID;

hasStatusCountFile = true;

}

/\*\*

\* Sometimes tweets are out-of-order (E.g. a tweet from Sep 2012 got into a

\* batch in July 2013). See SEARCH-1750 for more details.

\* This adjust the minStatusID if it is badly out-of-order.

\*/

@VisibleForTesting

protected static long adjustMinStatusId(long minStatusID, Date date) {

long dateTime = date.getTime();

// If the daily batch is for a day before we started using snow flake IDs. Never adjust.

if (!SnowflakeIdParser.isUsableSnowflakeTimestamp(dateTime)) {

return minStatusID;

}

long earliestStartTime = dateTime - MAXIMUM\_OUT\_OF\_ORDER\_TOLERANCE\_MILLIS;

long minStatusTime = SnowflakeIdParser.getTimestampFromTweetId(minStatusID);

if (minStatusTime < earliestStartTime) {

long newMinId = SnowflakeIdParser.generateValidStatusId(earliestStartTime, 0);

LOG.info("Daily batch for " + date + " has badly out of order tweet: " + minStatusID

+ ". The minStatusID for the day this batch is adjusted to " + newMinId);

return newMinId;

} else {

return minStatusID;

}

}

/\*\*

\* Returns a reader that reads tweets from the given directory.

\*

\* @param archiveSegment Determines the timeslice ID of all read tweets.

\* @param tweetsPath The path to the directory where the tweets for this day are stored.

\* @param documentFactory The ThriftIndexingEvent to TweetDocument converter.

\*/

public RecordReader<TweetDocument> getTweetReaders(

ArchiveSegment archiveSegment,

Path tweetsPath,

DocumentFactory<ThriftIndexingEvent> documentFactory) throws IOException {

RecordReader<TweetDocument> tweetDocumentReader =

new TransformingRecordReader<>(

createTweetReader(tweetsPath), new Function<ThriftIndexingEvent, TweetDocument>() {

@Override

public TweetDocument apply(ThriftIndexingEvent event) {

return new TweetDocument(

event.getSortId(),

archiveSegment.getTimeSliceID(),

EarlybirdThriftDocumentUtil.getCreatedAtMs(event.getDocument()),

documentFactory.newDocument(event)

);

}

});

tweetDocumentReader.setExhaustStream(true);

return tweetDocumentReader;

}

private RecordReader<ThriftIndexingEvent> createTweetReader(Path tweetsPath) throws IOException {

if (date.before(START\_DATE\_INCLUSIVE)) {

return new EmptyRecordReader<>();

}

List<RecordReader<ThriftIndexingEvent>> readers = Lists.newArrayList();

FileSystem hdfs = HdfsUtil.getHdfsFileSystem();

try {

Path dayPath = new Path(tweetsPath, ArchiveHDFSUtils.dateToPath(date, "/"));

Path partitionPath =

new Path(dayPath, String.format("p\_%d\_of\_%d", hashPartitionID, numHashPartitions));

PathFilter pathFilter =

Config.environmentIsTest() ? TXT\_DATA\_FILES\_FILTER : LZO\_DATA\_FILES\_FILTER;

FileStatus[] files = hdfs.listStatus(partitionPath, pathFilter);

for (FileStatus fileStatus : files) {

String fileStatusPath = fileStatus.getPath().toString().replaceAll("file:/", "/");

RecordReader<ThriftIndexingEvent> reader = createRecordReaderWithRetries(fileStatusPath);

readers.add(reader);

}

} finally {

IOUtils.closeQuietly(hdfs);

}

if (readers.isEmpty()) {

return new EmptyRecordReader<>();

}

return new MergingSortedRecordReader<>(DESC\_THRIFT\_INDEXING\_EVENT\_COMPARATOR, readers);

}

private RecordReader<ThriftIndexingEvent> createRecordReaderWithRetries(String filePath)

throws IOException {

Predicate<ThriftIndexingEvent> recordFilter = getRecordFilter();

int numTries = 0;

while (true) {

try {

++numTries;

return new LzoThriftBlockFileReader<>(filePath, ThriftIndexingEvent.class, recordFilter);

} catch (IOException e) {

if (numTries < READER\_INIT\_IOEXCEPTION\_RETRIES) {

LOG.warn("Failed to open LzoThriftBlockFileReader for " + filePath + ". Will retry.", e);

} else {

LOG.error("Failed to open LzoThriftBlockFileReader for " + filePath

+ " after too many retries.", e);

throw e;

}

}

}

}

private Predicate<ThriftIndexingEvent> getRecordFilter() {

return Config.environmentIsTest() ? null : input -> {

if (input == null) {

return false;

}

// We only guard against status IDs that are too small, because it is possible

// for a very old tweet to get into today's batch, but not possible for a very

// large ID (a future tweet ID that is not yet published) to get in today's

// batch, unless tweet ID generation messed up.

long statusId = input.getSortId();

boolean keep = statusId >= minStatusID;

if (!keep) {

LOG.debug("Out of order documentId: {} minStatusID: {} Date: {} Path: {}",

statusId, minStatusID, date, path);

OUT\_OF\_ORDER\_STATUSES\_SKIPPED.increment();

}

return keep;

};

}

/\*\*

\* Returns the number of statuses in this batch

\*/

public int getStatusCount() {

return statusCount;

}

/\*\*

\* Was the \_status\_count file was found in this folder.

\*/

public boolean hasStatusCount() {

return hasStatusCountFile;

}

public long getMinStatusID() {

return minStatusID;

}

public long getMaxStatusID() {

return maxStatusID;

}

public Date getDate() {

return date;

}

public Path getPath() {

return path;

}

/\*\*

\* Check whether the partition is

\* . empty and

\* . it is disallowed (empty partition can only happen before 2010)

\* (Empty partition means that the directory is missing when scan happens.)

\*

\* @return true if the partition has no documents and it is not allowed.

\*/

public boolean isDisallowedEmptyPartition() {

return hasStatusCountFile

&& statusCount == 0

&& minStatusID == DailyStatusBatch.EMPTY\_BATCH\_STATUS\_ID

&& maxStatusID == DailyStatusBatch.EMPTY\_BATCH\_STATUS\_ID

&& date.after(getEarliestDenseDay());

}

@Override

public String toString() {

return "PartitionedBatch[hashPartitionId=" + hashPartitionID

+ ",numHashPartitions=" + numHashPartitions

+ ",date=" + date

+ ",path=" + path

+ ",hasStatusCountFile=" + hasStatusCountFile

+ ",statusCount=" + statusCount + "]";

}

private Date getEarliestDenseDay() {

return EarlybirdConfig.getDate("archive\_search\_earliest\_dense\_day");

}

}