package com.twitter.search.earlybird.archive;

import java.io.File;

import java.io.FileNotFoundException;

import java.io.FileWriter;

import java.io.IOException;

import java.util.Calendar;

import java.util.Collection;

import java.util.Date;

import java.util.NavigableMap;

import java.util.concurrent.TimeUnit;

import java.util.concurrent.atomic.AtomicBoolean;

import java.util.regex.Matcher;

import java.util.regex.Pattern;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Preconditions;

import com.google.common.base.Stopwatch;

import com.google.common.collect.Maps;

import org.apache.commons.io.IOUtils;

import org.apache.commons.lang3.time.FastDateFormat;

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.quantity.Amount;

import com.twitter.common.quantity.Time;

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

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

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

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

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

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

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

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

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

/\*\*

\* Provides access to preprocessed statuses (tweets) to be indexed by archive search earlybirds.

\*

\* These tweets can be coming from a scrub gen or from the output of the daily jobs.

\*/

public class DailyStatusBatches {

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

// Maximum time to spend on obtaining daily status batches by computing or loading from HDFS

private static final Amount<Long, Time> MAX\_TIME\_ALLOWED\_DAILY\_STATUS\_BATCHES\_MINUTES =

Amount.of(EarlybirdConfig.getLong("daily\_status\_batches\_max\_initial\_load\_time\_minutes"),

Time.MINUTES);

// Time to wait before trying again when obtaining daily status batches fails

private static final Amount<Long, Time> DAILY\_STATUS\_BATCHES\_WAITING\_TIME\_MINUTES =

Amount.of(EarlybirdConfig.getLong("daily\_status\_batches\_waiting\_time\_minutes"),

Time.MINUTES);

private static final String DAILY\_STATUS\_BATCHES\_SYNC\_PATH =

EarlybirdProperty.ZK\_APP\_ROOT.get() + "/daily\_batches\_sync";

private static final String DAILY\_BATCHES\_ZK\_LOCK = "daily\_batches\_zk\_lock";

private static final Amount<Long, Time> DAILY\_STATUS\_BATCHES\_ZK\_LOCK\_EXPIRATION\_MINUTES =

Amount.of(EarlybirdConfig.getLong("daily\_status\_batches\_zk\_lock\_expiration\_minutes"),

Time.MINUTES);

static final FastDateFormat DATE\_FORMAT = FastDateFormat.getInstance("yyyyMMdd");

// before this date, there was no twitter

private static final Date FIRST\_TWITTER\_DAY = DateUtil.toDate(2006, 2, 1);

private static final String STATUS\_BATCHES\_PREFIX = "status\_batches";

private final String rootDir =

EarlybirdConfig.getString("hdfs\_offline\_segment\_sync\_dir", "top\_archive\_statuses");

private final String buildGen =

EarlybirdConfig.getString("offline\_segment\_build\_gen", "bg\_1");

public static final String STATUS\_SUBDIR\_NAME = "statuses";

public static final String LAYOUT\_SUBDIR\_NAME = "layouts";

public static final String SCRUB\_GEN\_SUFFIX\_PATTERN = "scrubbed/%s";

private static final String INTERMEDIATE\_COUNTS\_SUBDIR\_NAME = "counts";

private static final String SUCCESS\_FILE\_NAME = "\_SUCCESS";

private static final Pattern HASH\_PARTITION\_PATTERN = Pattern.compile("p\_(\\d+)\_of\_(\\d+)");

private static final Date FIRST\_TWEET\_DAY = DateUtil.toDate(2006, 3, 21);

private final Path rootPath = new Path(rootDir);

private final Path buildGenPath = new Path(rootPath, buildGen);

private final Path statusPath = new Path(buildGenPath, STATUS\_SUBDIR\_NAME);

private final NavigableMap<Date, DailyStatusBatch> statusBatches = Maps.newTreeMap();

private Date firstValidDay = null;

private Date lastValidDay = null;

private final ZooKeeperTryLockFactory zkTryLockFactory;

private final Date scrubGenDay;

private long numberOfDaysWithValidScrubGenData;

public DailyStatusBatches(

ZooKeeperTryLockFactory zooKeeperTryLockFactory, Date scrubGenDay) throws IOException {

this.zkTryLockFactory = zooKeeperTryLockFactory;

this.scrubGenDay = scrubGenDay;

FileSystem hdfs = null;

try {

hdfs = HdfsUtil.getHdfsFileSystem();

verifyDirectory(hdfs);

} finally {

IOUtils.closeQuietly(hdfs);

}

}

@VisibleForTesting

public Date getScrubGenDay() {

return scrubGenDay;

}

public Collection<DailyStatusBatch> getStatusBatches() {

return statusBatches.values();

}

/\*\*

\* Reset the states of the directory

\*/

private void resetDirectory() {

statusBatches.clear();

firstValidDay = null;

lastValidDay = null;

}

/\*\*

\* Indicate whether the directory has been initialized

\*/

private boolean isInitialized() {

return lastValidDay != null;

}

/\*\*

\* Load the daily status batches from HDFS; return true if one or more batches could be loaded.

\*\*/

private boolean refreshByLoadingHDFSStatusBatches(final FileSystem fs) throws IOException {

// first find the latest valid end date of statuses

final Date lastValidStatusDay = getLastValidInputDateFromNow(fs);

if (lastValidStatusDay != null) {

if (hasStatusBatchesOnHdfs(fs, lastValidStatusDay)) {

if (loadStatusBatchesFromHdfs(fs, lastValidStatusDay)) {

return true;

}

}

}

resetDirectory();

return false;

}

/\*\*

\* Checks the directory for new data and returns true, if one or more new batches could be loaded.

\*/

public void refresh() throws IOException {

final FileSystem hdfs = HdfsUtil.getHdfsFileSystem();

final Stopwatch stopwatch = Stopwatch.createStarted();

try {

if (!isInitialized()) {

if (initializeDailyStatusBatches(hdfs, stopwatch)) {

LOG.info("Successfully obtained daily status batches after {}", stopwatch);

} else {

String errMsg = "Failed to load or compute daily status batches after "

+ stopwatch.toString();

LOG.error(errMsg);

throw new IOException(errMsg);

}

} else {

loadNewDailyBatches(hdfs);

}

} finally {

IOUtils.closeQuietly(hdfs);

}

}

private boolean initializeDailyStatusBatches(final FileSystem hdfs,

final Stopwatch stopwatch) throws IOException {

long timeSpentOnDailyBatches = 0L;

long maxAllowedTimeMs = MAX\_TIME\_ALLOWED\_DAILY\_STATUS\_BATCHES\_MINUTES.as(Time.MILLISECONDS);

long waitingTimeMs = DAILY\_STATUS\_BATCHES\_WAITING\_TIME\_MINUTES.as(Time.MILLISECONDS);

boolean firstLoop = true;

LOG.info("Starting to load or compute daily status batches for the first time.");

while (timeSpentOnDailyBatches <= maxAllowedTimeMs && !Thread.currentThread().isInterrupted()) {

if (!firstLoop) {

try {

LOG.info("Sleeping " + waitingTimeMs

+ " millis before trying to obtain daily batches again");

Thread.sleep(waitingTimeMs);

} catch (InterruptedException e) {

LOG.warn("Interrupted while waiting to load daily batches", e);

Thread.currentThread().interrupt();

break;

}

}

if (isStatusBatchLoadingEnabled() && refreshByLoadingHDFSStatusBatches(hdfs)) {

LOG.info("Successfully loaded daily status batches after {}", stopwatch);

return true;

}

final AtomicBoolean successRef = new AtomicBoolean(false);

if (computeDailyBatchesWithZKLock(hdfs, successRef, stopwatch)) {

return successRef.get();

}

timeSpentOnDailyBatches = stopwatch.elapsed(TimeUnit.MILLISECONDS);

firstLoop = false;

}

return false;

}

private boolean computeDailyBatchesWithZKLock(final FileSystem hdfs,

final AtomicBoolean successRef,

final Stopwatch stopwatch) throws IOException {

// Using a global lock to coordinate among earlybirds and segment builders so that only

// one instance would hit the HDFS name node to query the daily status directories

TryLock lock = zkTryLockFactory.createTryLock(

DatabaseConfig.getLocalHostname(),

DAILY\_STATUS\_BATCHES\_SYNC\_PATH,

DAILY\_BATCHES\_ZK\_LOCK,

DAILY\_STATUS\_BATCHES\_ZK\_LOCK\_EXPIRATION\_MINUTES);

return lock.tryWithLock(() -> {

LOG.info("Obtained ZK lock to compute daily status batches after {}", stopwatch);

successRef.set(initialLoadDailyBatchInfos(hdfs));

if (successRef.get()) {

LOG.info("Successfully computed daily status batches after {}", stopwatch);

if (isStatusBatchFlushingEnabled()) {

LOG.info("Starting to store daily status batches to HDFS");

if (storeStatusBatchesToHdfs(hdfs, lastValidDay)) {

LOG.info("Successfully stored daily status batches to HDFS");

} else {

LOG.warn("Failed storing daily status batches to HDFS");

}

}

} else {

LOG.info("Failed loading daily status info");

}

});

}

private void verifyDirectory(FileSystem hdfs) throws IOException {

if (!hdfs.exists(rootPath)) {

throw new IOException("Root dir '" + rootPath + "' does not exist.");

}

if (!hdfs.exists(buildGenPath)) {

throw new IOException("Build gen dir '" + buildGenPath + "' does not exist.");

}

if (!hdfs.exists(statusPath)) {

throw new IOException("Status dir '" + statusPath + "' does not exist.");

}

}

private void loadNewDailyBatches(FileSystem hdfs) throws IOException {

Preconditions.checkNotNull(lastValidDay);

Calendar day = Calendar.getInstance();

day.setTime(lastValidDay);

day.add(Calendar.DATE, 1);

while (loadDay(hdfs, day.getTime()) != null) {

lastValidDay = day.getTime();

day.add(Calendar.DATE, 1);

}

}

private boolean initialLoadDailyBatchInfos(FileSystem hdfs) throws IOException {

LOG.info("Starting to build timeslice map from scratch.");

final Date lastValidStatusDay = getLastValidInputDateFromNow(hdfs);

if (lastValidStatusDay == null) {

LOG.warn("No data found in " + statusPath + " and scrubbed path");

return false;

}

int mostRecentYear = DateUtil.getCalendar(lastValidStatusDay).get(Calendar.YEAR);

for (int year = 2006; year <= mostRecentYear; ++year) {

// construct path to avoid hdfs.listStatus() calls

Calendar day = Calendar.getInstance();

day.set(year, Calendar.JANUARY, 1, 0, 0, 0);

day.set(Calendar.MILLISECOND, 0);

Calendar yearEnd = Calendar.getInstance();

yearEnd.set(year, Calendar.DECEMBER, 31, 0, 0, 0);

yearEnd.set(Calendar.MILLISECOND, 0);

if (lastValidDay != null) {

// We're updating.

if (lastValidDay.after(yearEnd.getTime())) {

// This year was already loaded.

continue;

}

if (lastValidDay.after(day.getTime())) {

// Start one day after last valid date.

day.setTime(lastValidDay);

day.add(Calendar.DATE, 1);

}

}

for (; !day.after(yearEnd); day.add(Calendar.DATE, 1)) {

loadDay(hdfs, day.getTime());

}

}

boolean updated = false;

numberOfDaysWithValidScrubGenData = 0;

// Iterate batches in sorted order.

for (DailyStatusBatch batch : statusBatches.values()) {

if (!batch.isValid()) {

break;

}

if (batch.getDate().before(scrubGenDay)) {

numberOfDaysWithValidScrubGenData++;

}

if (firstValidDay == null) {

firstValidDay = batch.getDate();

}

if (lastValidDay == null || lastValidDay.before(batch.getDate())) {

lastValidDay = batch.getDate();

updated = true;

}

}

LOG.info("Number of statusBatches: {}", statusBatches.size());

return updated;

}

private static String filesToString(FileStatus[] files) {

if (files == null) {

return "null";

}

StringBuilder b = new StringBuilder();

for (FileStatus s : files) {

b.append(s.getPath().toString()).append(", ");

}

return b.toString();

}

@VisibleForTesting

protected DailyStatusBatch loadDay(FileSystem hdfs, Date day) throws IOException {

Path dayPath = new Path(getStatusPathToUseForDay(day), ArchiveHDFSUtils.dateToPath(day, "/"));

LOG.debug("Looking for batch in " + dayPath.toString());

DailyStatusBatch result = this.statusBatches.get(day);

if (result != null) {

return result;

}

final FileStatus[] files;

try {

files = hdfs.listStatus(dayPath);

LOG.debug("Files found: " + filesToString(files));

} catch (FileNotFoundException e) {

LOG.debug("loadDay() called, but directory does not exist for day: " + day

+ " in: " + dayPath);

return null;

}

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

for (FileStatus file : files) {

Matcher matcher = HASH\_PARTITION\_PATTERN.matcher(file.getPath().getName());

if (matcher.matches()) {

int numHashPartitions = Integer.parseInt(matcher.group(2));

result = new DailyStatusBatch(

day, numHashPartitions, getStatusPathToUseForDay(day), hdfs);

for (int partitionID = 0; partitionID < numHashPartitions; partitionID++) {

result.addPartition(hdfs, dayPath, partitionID);

}

if (result.isValid()) {

statusBatches.put(day, result);

return result;

} else {

LOG.info("Invalid batch found for day: " + day + ", batch: " + result);

}

} else {

// skip logging the intermediate count subdirectories or \_SUCCESS files.

if (!INTERMEDIATE\_COUNTS\_SUBDIR\_NAME.equals(file.getPath().getName())

&& !SUCCESS\_FILE\_NAME.equals(file.getPath().getName())) {

LOG.warn("Path does not match hash partition pattern: " + file.getPath());

}

}

}

} else {

LOG.warn("No data found for day: " + day + " in: " + dayPath

+ " files null: " + (files == null));

}

return null;

}

/\*\*

\* Determines if this directory has a valid batch for the given day.

\*/

public boolean hasValidBatchForDay(Date day) throws IOException {

FileSystem hdfs = null;

try {

hdfs = HdfsUtil.getHdfsFileSystem();

return hasValidBatchForDay(hdfs, day);

} finally {

IOUtils.closeQuietly(hdfs);

}

}

private boolean hasValidBatchForDay(FileSystem fs, Date day) throws IOException {

DailyStatusBatch batch = loadDay(fs, day);

return batch != null && batch.isValid();

}

@VisibleForTesting

Date getFirstValidDay() {

return firstValidDay;

}

@VisibleForTesting

Date getLastValidDay() {

return lastValidDay;

}

private Date getLastValidInputDateFromNow(FileSystem hdfs) throws IOException {

Calendar cal = Calendar.getInstance();

cal.setTime(new Date()); // current date

return getLastValidInputDate(hdfs, cal);

}

/\*\*

\* Starting from current date, probe backward till we find a valid input Date

\*/

@VisibleForTesting

Date getLastValidInputDate(FileSystem hdfs, Calendar cal) throws IOException {

cal.set(Calendar.MILLISECOND, 0);

cal.set(Calendar.HOUR\_OF\_DAY, 0);

cal.set(Calendar.MINUTE, 0);

cal.set(Calendar.SECOND, 0);

cal.set(Calendar.MILLISECOND, 0);

Date lastValidInputDate = cal.getTime();

LOG.info("Probing backwards for last valid data date from " + lastValidInputDate);

while (lastValidInputDate.after(FIRST\_TWITTER\_DAY)) {

if (hasValidBatchForDay(hdfs, lastValidInputDate)) {

LOG.info("Found latest valid data on date " + lastValidInputDate);

LOG.info(" Used path: {}", getStatusPathToUseForDay(lastValidInputDate));

return lastValidInputDate;

}

cal.add(Calendar.DATE, -1);

lastValidInputDate = cal.getTime();

}

return null;

}

/\*\*

\* Check if the daily status batches are already on HDFS

\*/

@VisibleForTesting

boolean hasStatusBatchesOnHdfs(FileSystem fs, Date lastDataDay) {

String hdfsFileName = getHdfsStatusBatchSyncFileName(lastDataDay);

try {

return fs.exists(new Path(hdfsFileName));

} catch (IOException ex) {

LOG.error("Failed checking status batch file on HDFS: " + hdfsFileName, ex);

return false;

}

}

/\*\*

\* Load the daily status batches from HDFS by first copying the file from HDFS to local disk

\* and then reading from the local disk.

\*

\* @param day the latest day of valid statuses.

\* @return true if the loading is successful.

\*/

@VisibleForTesting

boolean loadStatusBatchesFromHdfs(FileSystem fs, Date day) {

// set the directory state to initial state

resetDirectory();

String fileHdfsPath = getHdfsStatusBatchSyncFileName(day);

String fileLocalPath = getLocalStatusBatchSyncFileName(day);

LOG.info("Using " + fileHdfsPath + " as the HDFS batch summary load path.");

LOG.info("Using " + fileLocalPath + " as the local batch summary sync path.");

LineRecordFileReader lineReader = null;

try {

fs.copyToLocalFile(new Path(fileHdfsPath), new Path(fileLocalPath));

lineReader = new LineRecordFileReader(fileLocalPath);

String batchLine;

while ((batchLine = lineReader.readNext()) != null) {

DailyStatusBatch batch = DailyStatusBatch.deserializeFromJson(batchLine);

if (batch == null) {

LOG.error("Invalid daily status batch constructed from line: " + batchLine);

resetDirectory();

return false;

}

Date date = batch.getDate();

if (firstValidDay == null || firstValidDay.after(date)) {

firstValidDay = date;

}

if (lastValidDay == null || lastValidDay.before(date)) {

lastValidDay = date;

}

statusBatches.put(date, batch);

}

LOG.info("Loaded {} status batches from HDFS: {}",

statusBatches.size(), fileHdfsPath);

LOG.info("First entry: {}", statusBatches.firstEntry().getValue().toString());

LOG.info("Last entry: {}", statusBatches.lastEntry().getValue().toString());

return true;

} catch (IOException ex) {

LOG.error("Failed loading time slices from HDFS: " + fileHdfsPath, ex);

resetDirectory();

return false;

} finally {

if (lineReader != null) {

lineReader.stop();

}

}

}

/\*\*

\* Flush the daily status batches to local disk and then upload to HDFS.

\*/

private boolean storeStatusBatchesToHdfs(FileSystem fs, Date day) {

Preconditions.checkNotNull(lastValidDay);

if (!StatusBatchFlushVersion.CURRENT\_FLUSH\_VERSION.isOfficial()) {

LOG.info("Status batch flush version is not official, no batches will be flushed to HDFS");

return true;

}

String fileLocalPath = getLocalStatusBatchSyncFileName(day);

// Flush to local disk

File outputFile = null;

FileWriter fileWriter = null;

try {

LOG.info("Flushing daily status batches into: " + fileLocalPath);

outputFile = new File(fileLocalPath);

outputFile.getParentFile().mkdirs();

if (!outputFile.getParentFile().exists()) {

LOG.error("Cannot create directory: " + outputFile.getParentFile().toString());

return false;

}

fileWriter = new FileWriter(outputFile, false);

for (Date date : statusBatches.keySet()) {

fileWriter.write(statusBatches.get(date).serializeToJson());

fileWriter.write("\n");

}

fileWriter.flush();

// Upload the file to HDFS

return uploadStatusBatchesToHdfs(fs, day);

} catch (IOException e) {

String fileHdfsPath = getHdfsStatusBatchSyncFileName(day);

LOG.error("Failed storing status batches to HDFS: " + fileHdfsPath, e);

return false;

} finally {

try {

if (fileWriter != null) {

fileWriter.close();

}

} catch (IOException e) {

LOG.error("Error to close fileWrite.", e);

}

if (outputFile != null) {

// Delete the local file

outputFile.delete();

}

}

}

/\*\*

\* Upload the status batches to HDFS.

\*/

@VisibleForTesting

boolean uploadStatusBatchesToHdfs(FileSystem fs, Date day) {

String localFileName = getLocalStatusBatchSyncFileName(day);

String hdfsFileName = getHdfsStatusBatchSyncFileName(day);

LOG.info("Using " + hdfsFileName + " as the HDFS batch summary upload path.");

LOG.info("Using " + localFileName + " as the local batch summary sync path.");

try {

Path hdfsFilePath = new Path(hdfsFileName);

if (fs.exists(hdfsFilePath)) {

LOG.warn("Found status batch file on HDFS: " + hdfsFileName);

return true;

}

String hdfsTempName = getHdfsStatusBatchTempSyncFileName(day);

Path hdfsTempPath = new Path(hdfsTempName);

if (fs.exists(hdfsTempPath)) {

LOG.info("Found existing temporary status batch file on HDFS, removing: " + hdfsTempName);

if (!fs.delete(hdfsTempPath, false)) {

LOG.error("Failed to delete temporary file: " + hdfsTempName);

return false;

}

}

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

if (fs.rename(hdfsTempPath, hdfsFilePath)) {

LOG.debug("Renamed " + hdfsTempName + " on HDFS to: " + hdfsFileName);

return true;

} else {

LOG.error("Failed to rename " + hdfsTempName + " on HDFS to: " + hdfsFileName);

return false;

}

} catch (IOException ex) {

LOG.error("Failed uploading status batch file to HDFS: " + hdfsFileName, ex);

return false;

}

}

private static boolean isStatusBatchFlushingEnabled() {

return EarlybirdProperty.ARCHIVE\_DAILY\_STATUS\_BATCH\_FLUSHING\_ENABLED.get(false);

}

private static boolean isStatusBatchLoadingEnabled() {

return EarlybirdConfig.getBool("archive\_daily\_status\_batch\_loading\_enabled", false);

}

private static String getVersionFileExtension() {

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

}

String getStatusBatchSyncRootDir() {

return EarlybirdConfig.getString("archive\_daily\_status\_batch\_sync\_dir",

"daily\_status\_batches") + "/" + scrubGenSuffix();

}

@VisibleForTesting

String getLocalStatusBatchSyncFileName(Date day) {

return getStatusBatchSyncRootDir() + "/" + STATUS\_BATCHES\_PREFIX + "\_"

+ DATE\_FORMAT.format(day) + getVersionFileExtension();

}

String getHdfsStatusBatchSyncRootDir() {

return EarlybirdConfig.getString("hdfs\_archive\_daily\_status\_batch\_sync\_dir",

"daily\_status\_batches") + "/" + scrubGenSuffix();

}

@VisibleForTesting

String getHdfsStatusBatchSyncFileName(Date day) {

return getHdfsStatusBatchSyncRootDir() + "/" + STATUS\_BATCHES\_PREFIX + "\_"

+ DATE\_FORMAT.format(day) + getVersionFileExtension();

}

private String getHdfsStatusBatchTempSyncFileName(Date day) {

return getHdfsStatusBatchSyncRootDir() + "/" + DatabaseConfig.getLocalHostname() + "\_"

+ STATUS\_BATCHES\_PREFIX + "\_" + DATE\_FORMAT.format(day) + getVersionFileExtension();

}

private String scrubGenSuffix() {

return String.format(SCRUB\_GEN\_SUFFIX\_PATTERN, DATE\_FORMAT.format(scrubGenDay));

}

/\*\*

\* Returns the path to the directory that stores the statuses for the given day.

\*/

public Path getStatusPathToUseForDay(Date day) {

if (!day.before(scrubGenDay)) {

return statusPath;

}

String suffix = scrubGenSuffix();

Preconditions.checkArgument(!suffix.isEmpty());

Path scrubPath = new Path(buildGenPath, suffix);

return new Path(scrubPath, STATUS\_SUBDIR\_NAME);

}

/\*\*

\* Determines if the data for the specified scrub gen was fully built, by checking the number of

\* days for which data was built against the expected number of days extracted from the specified

\* scrub gen date.

\*/

public boolean isScrubGenDataFullyBuilt(FileSystem hdfs) throws IOException {

initialLoadDailyBatchInfos(hdfs);

if (numberOfDaysWithValidScrubGenData == 0) {

LOG.warn("numberOfDaysWithValidScrubGenData is 0");

}

long expectedDays = getDiffBetweenDays(scrubGenDay);

return expectedDays == numberOfDaysWithValidScrubGenData;

}

@VisibleForTesting

long getDiffBetweenDays(Date day) {

long diff = day.getTime() - FIRST\_TWEET\_DAY.getTime();

return TimeUnit.DAYS.convert(diff, TimeUnit.MILLISECONDS);

}

}