package com.twitter.search.earlybird.util;

import java.util.Calendar;

import java.util.Collections;

import java.util.Map;

import java.util.TimeZone;

import java.util.concurrent.atomic.AtomicInteger;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Preconditions;

import com.google.common.collect.Maps;

import org.apache.commons.lang.mutable.MutableInt;

import org.apache.commons.lang.mutable.MutableLong;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

/\*\*

\* This class is used to count how many times a field happens in hourly and daily stats.

\* It is used by TermCountMonitor for iterating all fields in the index.

\*

\* There is one exception that this class is also used to count the number of tweets in the index.

\* Under the situation, the passed in fieldName would be empty string (as TWEET\_COUNT\_KEY).

\*/

public class FieldTermCounter {

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

static final TimeZone TIME\_ZONE = TimeZone.getTimeZone("GMT");

static final String TWEET\_COUNT\_KEY = "";

private final String fieldName;

private final int instanceCounter;

// The first date in format "YYYYMMDDHH" that we want to check counts for.

private final int startCheckHour;

// The last date in format "YYYYMMDDHH" that we want to check counts for.

private final int endCheckHour;

// Smallest number of docs we expect to have for each hour.

private final int hourlyMinCount;

//Smallest number of docs we expect to have for each day.

private final int dailyMinCount;

// Count of tweets for each day, keyed of by the hour in the format "YYYYMMDD".

private final Map<Integer, AtomicInteger> exportedHourlyCounts;

// Count of tweets for each day, keyed of by the day in the format "YYYYMMDD".

private final Map<Integer, MutableLong> dailyCounts;

// Only export hourly stats that are below minimum threshold.

private final Map<String, SearchLongGauge> exportedStats;

private final SearchLongGauge hoursWithNoTweetsStat;

private final SearchLongGauge daysWithNoTweetsStat;

public FieldTermCounter(

String fieldName,

int instanceCounter,

int startCheckHour,

int endCheckHour,

int hourlyMinCount,

int dailyMinCount) {

this.fieldName = fieldName;

this.instanceCounter = instanceCounter;

this.startCheckHour = startCheckHour;

this.endCheckHour = endCheckHour;

this.hourlyMinCount = hourlyMinCount;

this.dailyMinCount = dailyMinCount;

this.exportedHourlyCounts = Maps.newHashMap();

this.dailyCounts = Maps.newHashMap();

this.exportedStats = Maps.newHashMap();

this.hoursWithNoTweetsStat = SearchLongGauge.export(getAggregatedNoTweetStatName(true));

this.daysWithNoTweetsStat = SearchLongGauge.export(getAggregatedNoTweetStatName(false));

}

/\*\*

\* Updates the stats exported by this class based on the new counts provided in the given map.

\*/

public void runWithNewCounts(Map<Integer, MutableInt> newCounts) {

dailyCounts.clear();

// See go/rb/813442/#comment2566569

// 1. Update all existing hours

updateExistingHourlyCounts(newCounts);

// 2. Add and export all new hours

addAndExportNewHourlyCounts(newCounts);

// 3. fill in all the missing hours between know min and max days.

fillMissingHourlyCounts();

// 4. Export as a stat, how many hours don't have any tweets (i.e. <= 0)

exportMissingTweetStats();

}

// Input:

// . the new hourly count map in the current iteration

// . the existing hourly count map before the current iteration

// If the hourly key matches from the new hourly map to the existing hourly count map, update

// the value of the existing hourly count map to the value from the new hourly count map.

private void updateExistingHourlyCounts(Map<Integer, MutableInt> newCounts) {

for (Map.Entry<Integer, AtomicInteger> exportedCount : exportedHourlyCounts.entrySet()) {

Integer date = exportedCount.getKey();

AtomicInteger exportedCountValue = exportedCount.getValue();

MutableInt newCount = newCounts.get(date);

if (newCount == null) {

exportedCountValue.set(0);

} else {

exportedCountValue.set(newCount.intValue());

// clean up so that we don't check this date again when we look for new hours

newCounts.remove(date);

}

}

}

// Input:

// . the new hourly count map in the current iteration

// . the existing hourly count map before the current iteration

// This function is called after the above function of updateExistingHourlyCounts() so that all

// matching key value pairs have been removed from the new hourly count map.

// Move all remaining valid values from the new hourly count map to the existing hourly count

// map.

private void addAndExportNewHourlyCounts(Map<Integer, MutableInt> newCounts) {

for (Map.Entry<Integer, MutableInt> newCount : newCounts.entrySet()) {

Integer hour = newCount.getKey();

MutableInt newCountValue = newCount.getValue();

Preconditions.checkState(!exportedHourlyCounts.containsKey(hour),

"Should have already processed and removed existing hours: " + hour);

AtomicInteger newStat = new AtomicInteger(newCountValue.intValue());

exportedHourlyCounts.put(hour, newStat);

}

}

// Find whether the existing hourly count map has hourly holes. If such holes exist, fill 0

// values so that they can be exported.

private void fillMissingHourlyCounts() {

// Figure out the time range for which we should have tweets in the index. At the very least,

// this range should cover [startCheckHour, endCheckHour) if endCheckHour is set, or

// [startCheckHour, latestHourInTheIndexWithTweets] if endCheckHour is not set (latest tier or

// realtime cluster).

int startHour = startCheckHour;

int endHour = endCheckHour < getHourValue(Calendar.getInstance(TIME\_ZONE)) ? endCheckHour : -1;

for (int next : exportedHourlyCounts.keySet()) {

if (next < startHour) {

startHour = next;

}

if (next > endHour) {

endHour = next;

}

}

Calendar endHourCal = getCalendarValue(endHour);

Calendar hour = getCalendarValue(startHour);

for (; hour.before(endHourCal); hour.add(Calendar.HOUR\_OF\_DAY, 1)) {

int hourValue = getHourValue(hour);

if (!exportedHourlyCounts.containsKey(hourValue)) {

exportedHourlyCounts.put(hourValue, new AtomicInteger(0));

}

}

}

private void exportMissingTweetStats() {

int hoursWithNoTweets = 0;

int daysWithNoTweets = 0;

for (Map.Entry<Integer, AtomicInteger> hourlyCount : exportedHourlyCounts.entrySet()) {

int hour = hourlyCount.getKey();

if ((hour < startCheckHour) || (hour >= endCheckHour)) {

continue;

}

// roll up the days

int day = hour / 100;

MutableLong dayCount = dailyCounts.get(day);

if (dayCount == null) {

dailyCounts.put(day, new MutableLong(hourlyCount.getValue().get()));

} else {

dayCount.setValue(dayCount.longValue() + hourlyCount.getValue().get());

}

AtomicInteger exportedCountValue = hourlyCount.getValue();

if (exportedCountValue.get() <= hourlyMinCount) {

// We do not export hourly too few tweets for index fields as it can 10x the existing

// exported stats.

// We might consider whitelisting some high frequency fields later.

if (isFieldForTweet()) {

String statsName = getStatName(hourlyCount.getKey());

SearchLongGauge stat = SearchLongGauge.export(statsName);

stat.set(exportedCountValue.longValue());

exportedStats.put(statsName, stat);

}

LOG.warn("Found an hour with too few tweets. Field: <{}> Hour: {} count: {}",

fieldName, hour, exportedCountValue);

hoursWithNoTweets++;

}

}

for (Map.Entry<Integer, MutableLong> dailyCount : dailyCounts.entrySet()) {

if (dailyCount.getValue().longValue() <= dailyMinCount) {

LOG.warn("Found a day with too few tweets. Field: <{}> Day: {} count: {}",

fieldName, dailyCount.getKey(), dailyCount.getValue());

daysWithNoTweets++;

}

}

hoursWithNoTweetsStat.set(hoursWithNoTweets);

daysWithNoTweetsStat.set(daysWithNoTweets);

}

// When the fieldName is empty string (as TWEET\_COUNT\_KEY), it means that we are counting the

// number of tweets for the index, not for some specific fields.

private boolean isFieldForTweet() {

return TWEET\_COUNT\_KEY.equals(fieldName);

}

private String getAggregatedNoTweetStatName(boolean hourly) {

if (isFieldForTweet()) {

if (hourly) {

return "hours\_with\_no\_indexed\_tweets\_v\_" + instanceCounter;

} else {

return "days\_with\_no\_indexed\_tweets\_v\_" + instanceCounter;

}

} else {

if (hourly) {

return "hours\_with\_no\_indexed\_fields\_v\_" + fieldName + "\_" + instanceCounter;

} else {

return "days\_with\_no\_indexed\_fields\_v\_" + fieldName + "\_" + instanceCounter;

}

}

}

@VisibleForTesting

String getStatName(Integer date) {

return getStatName(fieldName, instanceCounter, date);

}

@VisibleForTesting

static String getStatName(String field, int instance, Integer date) {

if (TWEET\_COUNT\_KEY.equals(field)) {

return "tweets\_indexed\_on\_hour\_v\_" + instance + "\_" + date;

} else {

return "tweets\_indexed\_on\_hour\_v\_" + instance + "\_" + field + "\_" + date;

}

}

@VisibleForTesting

Map<Integer, AtomicInteger> getExportedCounts() {

return Collections.unmodifiableMap(exportedHourlyCounts);

}

@VisibleForTesting

Map<Integer, MutableLong> getDailyCounts() {

return Collections.unmodifiableMap(dailyCounts);

}

@VisibleForTesting

long getHoursWithNoTweets() {

return hoursWithNoTweetsStat.get();

}

@VisibleForTesting

long getDaysWithNoTweets() {

return daysWithNoTweetsStat.get();

}

@VisibleForTesting

Map<String, SearchLongGauge> getExportedHourlyCountStats() {

return exportedStats;

}

/\*\*

\* Given a unit time in seconds since epoch UTC, will return the day in format "YYYYMMDDHH"

\* as an int.

\*/

@VisibleForTesting

static int getHourValue(Calendar cal, int timeSecs) {

cal.setTimeInMillis(timeSecs \* 1000L);

return getHourValue(cal);

}

static int getHourValue(Calendar cal) {

int year = cal.get(Calendar.YEAR) \* 1000000;

int month = (cal.get(Calendar.MONTH) + 1) \* 10000; // month is 0-based

int day = cal.get(Calendar.DAY\_OF\_MONTH) \* 100;

int hour = cal.get(Calendar.HOUR\_OF\_DAY);

return year + month + day + hour;

}

@VisibleForTesting

static Calendar getCalendarValue(int hour) {

Calendar cal = Calendar.getInstance(TIME\_ZONE);

int year = hour / 1000000;

int month = ((hour / 10000) % 100) - 1; // 0-based

int day = (hour / 100) % 100;

int hr = hour % 100;

cal.setTimeInMillis(0); // reset all time fields

cal.set(year, month, day, hr, 0);

return cal;

}

}