package com.twitter.search.common.relevance.entities;

import java.text.Normalizer;

import java.util.Map;

import java.util.NavigableMap;

import java.util.Set;

import java.util.TreeMap;

import java.util.concurrent.ConcurrentMap;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Preconditions;

import com.google.common.collect.Maps;

import org.apache.commons.lang.StringUtils;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.common.text.transformer.HTMLTagRemovalTransformer;

import com.twitter.common\_internal.text.extractor.EmojiExtractor;

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

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

public final class TwitterMessageUtil {

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

private TwitterMessageUtil() {

}

@VisibleForTesting

static final ConcurrentMap<Field, Counters> COUNTERS\_MAP = Maps.newConcurrentMap();

// We truncate the location string because we used to use a MySQL table to store the geocoding

// information. In the MySQL table, the location string was fix width of 30 characters.

// We have migrated to Manhattan and the location string is no longer limited to 30 character.

// However, in order to correctly lookup location geocode from Manhattan, we still need to

// truncate the location just like we did before.

private static final int MAX\_LOCATION\_LEN = 30;

// Note: we strip tags to index source, as typically source contains <a href=...> tags.

// Sometimes we get a source where stripping fails, as the URL in the tag was

// excessively long. We drop these sources, as there is little reason to index them.

private static final int MAX\_SOURCE\_LEN = 64;

private static HTMLTagRemovalTransformer tagRemovalTransformer = new HTMLTagRemovalTransformer();

private static final String STAT\_PREFIX = "twitter\_message\_";

public enum Field {

FROM\_USER\_DISPLAY\_NAME,

NORMALIZED\_LOCATION,

ORIG\_LOCATION,

ORIG\_SOURCE,

SHARED\_USER\_DISPLAY\_NAME,

SOURCE,

TEXT,

TO\_USER\_SCREEN\_NAME;

public String getNameForStats() {

return name().toLowerCase();

}

}

@VisibleForTesting

static class Counters {

private final SearchRateCounter truncatedCounter;

private final SearchRateCounter tweetsWithStrippedSupplementaryCharsCounter;

private final SearchRateCounter strippedSupplementaryCharsCounter;

private final SearchRateCounter nonStrippedEmojiCharsCounter;

private final SearchRateCounter emojisAtTruncateBoundaryCounter;

Counters(Field field) {

String fieldNameForStats = field.getNameForStats();

truncatedCounter = SearchRateCounter.export(

STAT\_PREFIX + "truncated\_" + fieldNameForStats);

tweetsWithStrippedSupplementaryCharsCounter = SearchRateCounter.export(

STAT\_PREFIX + "tweets\_with\_stripped\_supplementary\_chars\_" + fieldNameForStats);

strippedSupplementaryCharsCounter = SearchRateCounter.export(

STAT\_PREFIX + "stripped\_supplementary\_chars\_" + fieldNameForStats);

nonStrippedEmojiCharsCounter = SearchRateCounter.export(

STAT\_PREFIX + "non\_stripped\_emoji\_chars\_" + fieldNameForStats);

emojisAtTruncateBoundaryCounter = SearchRateCounter.export(

STAT\_PREFIX + "emojis\_at\_truncate\_boundary\_" + fieldNameForStats);

}

SearchRateCounter getTruncatedCounter() {

return truncatedCounter;

}

SearchRateCounter getTweetsWithStrippedSupplementaryCharsCounter() {

return tweetsWithStrippedSupplementaryCharsCounter;

}

SearchRateCounter getStrippedSupplementaryCharsCounter() {

return strippedSupplementaryCharsCounter;

}

SearchRateCounter getNonStrippedEmojiCharsCounter() {

return nonStrippedEmojiCharsCounter;

}

SearchRateCounter getEmojisAtTruncateBoundaryCounter() {

return emojisAtTruncateBoundaryCounter;

}

}

static {

for (Field field : Field.values()) {

COUNTERS\_MAP.put(field, new Counters(field));

}

}

// Note: the monorail enforces a limit of 15 characters for screen names,

// but some users with up to 20 character names were grandfathered-in. To allow

// those users to be searchable, support up to 20 chars.

private static final int MAX\_SCREEN\_NAME\_LEN = 20;

// Note: we expect the current limit to be 10K. Also, all supplementary unicode characters (with

// the exception of emojis, maybe) will be removed and not counted as total length. Added alert

// for text truncation rate as well. SEARCH-9512

private static final int MAX\_TWEET\_TEXT\_LEN = 10000;

@VisibleForTesting

static final SearchRateCounter FILTERED\_NO\_STATUS\_ID =

SearchRateCounter.export(STAT\_PREFIX + "filtered\_no\_status\_id");

@VisibleForTesting

static final SearchRateCounter FILTERED\_NO\_FROM\_USER =

SearchRateCounter.export(STAT\_PREFIX + "filtered\_no\_from\_user");

@VisibleForTesting

static final SearchRateCounter FILTERED\_LONG\_SCREEN\_NAME =

SearchRateCounter.export(STAT\_PREFIX + "filtered\_long\_screen\_name");

@VisibleForTesting

static final SearchRateCounter FILTERED\_NO\_TEXT =

SearchRateCounter.export(STAT\_PREFIX + "filtered\_no\_text");

@VisibleForTesting

static final SearchRateCounter FILTERED\_NO\_DATE =

SearchRateCounter.export(STAT\_PREFIX + "filtered\_no\_date");

@VisibleForTesting

static final SearchRateCounter NULLCAST\_TWEET =

SearchRateCounter.export(STAT\_PREFIX + "filter\_nullcast\_tweet");

@VisibleForTesting

static final SearchRateCounter NULLCAST\_TWEET\_ACCEPTED =

SearchRateCounter.export(STAT\_PREFIX + "nullcast\_tweet\_accepted");

@VisibleForTesting

static final SearchRateCounter INCONSISTENT\_TWEET\_ID\_AND\_CREATED\_AT =

SearchRateCounter.export(STAT\_PREFIX + "inconsistent\_tweet\_id\_and\_created\_at\_ms");

/\*\* Strips the given source from the message with the given ID. \*/

private static String stripSource(String source, Long messageId) {

if (source == null) {

return null;

}

// Always strip emojis from sources: they don't really make sense in this field.

String strippedSource = stripSupplementaryChars(

tagRemovalTransformer.transform(source).toString(), Field.SOURCE, true);

if (strippedSource.length() > MAX\_SOURCE\_LEN) {

LOG.warn("Message "

+ messageId

+ " contains stripped source that exceeds MAX\_SOURCE\_LEN. Removing: "

+ strippedSource);

COUNTERS\_MAP.get(Field.SOURCE).getTruncatedCounter().increment();

return null;

}

return strippedSource;

}

/\*\*

\* Strips and truncates the location of the message with the given ID.

\*

\*/

private static String stripAndTruncateLocation(String location) {

// Always strip emojis from locations: they don't really make sense in this field.

String strippedLocation = stripSupplementaryChars(location, Field.NORMALIZED\_LOCATION, true);

return truncateString(strippedLocation, MAX\_LOCATION\_LEN, Field.NORMALIZED\_LOCATION, true);

}

/\*\*

\* Sets the origSource and strippedSource fields on a TwitterMessage

\*

\*/

public static void setSourceOnMessage(TwitterMessage message, String modifiedDeviceSource) {

// Always strip emojis from sources: they don't really make sense in this field.

message.setOrigSource(stripSupplementaryChars(modifiedDeviceSource, Field.ORIG\_SOURCE, true));

message.setStrippedSource(stripSource(modifiedDeviceSource, message.getId()));

}

/\*\*

\* Sets the origLocation to the stripped location, and sets

\* the truncatedNormalizedLocation to the truncated and normalized location.

\*/

public static void setAndTruncateLocationOnMessage(

TwitterMessage message,

String newOrigLocation) {

// Always strip emojis from locations: they don't really make sense in this field.

message.setOrigLocation(stripSupplementaryChars(newOrigLocation, Field.ORIG\_LOCATION, true));

// Locations in the new locations table require additional normalization. It can also change

// the length of the string, so we must do this before truncation.

if (newOrigLocation != null) {

String normalized =

Normalizer.normalize(newOrigLocation, Normalizer.Form.NFKC).toLowerCase().trim();

message.setTruncatedNormalizedLocation(stripAndTruncateLocation(normalized));

} else {

message.setTruncatedNormalizedLocation(null);

}

}

/\*\*

\* Validates the given TwitterMessage.

\*

\* @param message The message to validate.

\* @param stripEmojisForFields The set of fields for which emojis should be stripped.

\* @param acceptNullcastMessage Determines if this message should be accepted, if it's a nullcast

\* message.

\* @return {@code true} if the given message is valid; {@code false} otherwise.

\*/

public static boolean validateTwitterMessage(

TwitterMessage message,

Set<Field> stripEmojisForFields,

boolean acceptNullcastMessage) {

if (message.getNullcast()) {

NULLCAST\_TWEET.increment();

if (!acceptNullcastMessage) {

LOG.info("Dropping nullcasted message " + message.getId());

return false;

}

NULLCAST\_TWEET\_ACCEPTED.increment();

}

if (!message.getFromUserScreenName().isPresent()

|| StringUtils.isBlank(message.getFromUserScreenName().get())) {

LOG.error("Message " + message.getId() + " contains no from user. Skipping.");

FILTERED\_NO\_FROM\_USER.increment();

return false;

}

String fromUserScreenName = message.getFromUserScreenName().get();

if (fromUserScreenName.length() > MAX\_SCREEN\_NAME\_LEN) {

LOG.warn("Message " + message.getId() + " has a user screen name longer than "

+ MAX\_SCREEN\_NAME\_LEN + " characters: " + message.getFromUserScreenName()

+ ". Skipping.");

FILTERED\_LONG\_SCREEN\_NAME.increment();

return false;

}

// Remove supplementary characters and truncate these text fields.

if (message.getFromUserDisplayName().isPresent()) {

message.setFromUserDisplayName(stripSupplementaryChars(

message.getFromUserDisplayName().get(),

Field.FROM\_USER\_DISPLAY\_NAME,

stripEmojisForFields.contains(Field.FROM\_USER\_DISPLAY\_NAME)));

}

if (message.getToUserScreenName().isPresent()) {

String strippedToUserScreenName = stripSupplementaryChars(

message.getToUserLowercasedScreenName().get(),

Field.TO\_USER\_SCREEN\_NAME,

stripEmojisForFields.contains(Field.TO\_USER\_SCREEN\_NAME));

message.setToUserScreenName(

truncateString(

strippedToUserScreenName,

MAX\_SCREEN\_NAME\_LEN,

Field.TO\_USER\_SCREEN\_NAME,

stripEmojisForFields.contains(Field.TO\_USER\_SCREEN\_NAME)));

}

String strippedText = stripSupplementaryChars(

message.getText(),

Field.TEXT,

stripEmojisForFields.contains(Field.TEXT));

message.setText(truncateString(

strippedText,

MAX\_TWEET\_TEXT\_LEN,

Field.TEXT,

stripEmojisForFields.contains(Field.TEXT)));

if (StringUtils.isBlank(message.getText())) {

FILTERED\_NO\_TEXT.increment();

return false;

}

if (message.getDate() == null) {

LOG.error("Message " + message.getId() + " contains no date. Skipping.");

FILTERED\_NO\_DATE.increment();

return false;

}

if (message.isRetweet()) {

return validateRetweetMessage(message.getRetweetMessage(), stripEmojisForFields);

}

// Track if both the snowflake ID and created at timestamp are consistent.

if (!SnowflakeIdParser.isTweetIDAndCreatedAtConsistent(message.getId(), message.getDate())) {

LOG.error("Found inconsistent tweet ID and created at timestamp: [messageID="

+ message.getId() + "], [messageDate=" + message.getDate() + "].");

INCONSISTENT\_TWEET\_ID\_AND\_CREATED\_AT.increment();

}

return true;

}

private static boolean validateRetweetMessage(

TwitterRetweetMessage message, Set<Field> stripEmojisForFields) {

if (message.getSharedId() == null || message.getRetweetId() == null) {

LOG.error("Retweet Message contains a null twitter id. Skipping.");

FILTERED\_NO\_STATUS\_ID.increment();

return false;

}

if (message.getSharedDate() == null) {

LOG.error("Retweet Message " + message.getRetweetId() + " contains no date. Skipping.");

return false;

}

// Remove supplementary characters from these text fields.

message.setSharedUserDisplayName(stripSupplementaryChars(

message.getSharedUserDisplayName(),

Field.SHARED\_USER\_DISPLAY\_NAME,

stripEmojisForFields.contains(Field.SHARED\_USER\_DISPLAY\_NAME)));

return true;

}

/\*\*

\* Strips non indexable chars from the text.

\*

\* Returns the resulting string, which may be the same object as the text argument when

\* no stripping or truncation is necessary.

\*

\* Non-indexed characters are "supplementary unicode" that are not emojis. Note that

\* supplementary unicode are still characters that seem worth indexing, as many characters

\* in CJK languages are supplementary. However this would make the size of our index

\* explode (~186k supplementary characters exist), so it's not feasible.

\*

\* @param text The text to strip

\* @param field The field this text is from

\* @param stripSupplementaryEmojis Whether or not to strip supplementary emojis. Note that this

\* parameter name isn't 100% accurate. This parameter is meant to replicate behavior prior to

\* adding support for \*not\* stripping supplementary emojis. The prior behavior would turn an

\* emoji such as a keycap "1\uFE0F\u20E3" (http://www.iemoji.com/view/emoji/295/symbols/keycap-1)

\* into just '1'. So the keycap emoji is not completely stripped, only the portion after the '1'.

\*

\*/

@VisibleForTesting

public static String stripSupplementaryChars(

String text,

Field field,

boolean stripSupplementaryEmojis) {

if (text == null || text.isEmpty()) {

return text;

}

// Initialize an empty map so that if we choose not to strip emojis,

// then no emojipositions will be found and we don't need a null

// check before checking if an emoji is at a certain spot.

NavigableMap<Integer, Integer> emojiPositions = new TreeMap<>();

if (!stripSupplementaryEmojis) {

emojiPositions = EmojiExtractor.getEmojiPositions(text);

}

StringBuilder strippedTextBuilder = new StringBuilder();

int sequenceStart = 0;

int i = 0;

while (i < text.length()) {

if (Character.isSupplementaryCodePoint(text.codePointAt(i))) {

// Check if this supplementary character is an emoji

if (!emojiPositions.containsKey(i)) {

// It's not an emoji, or we want to strip emojis, so strip it

// text[i] and text[i + 1] are part of a supplementary code point.

strippedTextBuilder.append(text.substring(sequenceStart, i));

sequenceStart = i + 2; // skip 2 chars

i = sequenceStart;

COUNTERS\_MAP.get(field).getStrippedSupplementaryCharsCounter().increment();

} else {

// It's an emoji, keep it

i += emojiPositions.get(i);

COUNTERS\_MAP.get(field).getNonStrippedEmojiCharsCounter().increment();

}

} else {

++i;

}

}

if (sequenceStart < text.length()) {

strippedTextBuilder.append(text.substring(sequenceStart));

}

String strippedText = strippedTextBuilder.toString();

if (strippedText.length() < text.length()) {

COUNTERS\_MAP.get(field).getTweetsWithStrippedSupplementaryCharsCounter().increment();

}

return strippedText;

}

/\*\*

\* Truncates the given string to the given length.

\*

\* Note that we are truncating based on the # of UTF-16 characters a given emoji takes up.

\* So if a single emoji takes up 4 UTF-16 characters, that counts as 4 for the truncation,

\* not just 1.

\*

\* @param text The text to truncate

\* @param maxLength The maximum length of the string after truncation

\* @param field The field from which this string cames

\* @param splitEmojisAtMaxLength If true, don't worry about emojis and just truncate at maxLength,

\* potentially splitting them. If false, truncate before the emoji if truncating at maxLength

\* would cause the emoji to be split.

\*/

@VisibleForTesting

static String truncateString(

String text,

int maxLength,

Field field,

boolean splitEmojisAtMaxLength) {

Preconditions.checkArgument(maxLength > 0);

if ((text == null) || (text.length() <= maxLength)) {

return text;

}

int truncatePoint = maxLength;

NavigableMap<Integer, Integer> emojiPositions;

// If we want to consider emojis we should not strip on an emoji boundary.

if (!splitEmojisAtMaxLength) {

emojiPositions = EmojiExtractor.getEmojiPositions(text);

// Get the last emoji before maxlength.

Map.Entry<Integer, Integer> lastEmojiBeforeMaxLengthEntry =

emojiPositions.lowerEntry(maxLength);

if (lastEmojiBeforeMaxLengthEntry != null) {

int lowerEmojiEnd = lastEmojiBeforeMaxLengthEntry.getKey()

+ lastEmojiBeforeMaxLengthEntry.getValue();

// If the last emoji would be truncated, truncate before the last emoji.

if (lowerEmojiEnd > truncatePoint) {

truncatePoint = lastEmojiBeforeMaxLengthEntry.getKey();

COUNTERS\_MAP.get(field).getEmojisAtTruncateBoundaryCounter().increment();

}

}

}

COUNTERS\_MAP.get(field).getTruncatedCounter().increment();

return text.substring(0, truncatePoint);

}

}