package com.twitter.search.common.util.earlybird;

import java.util.ArrayList;

import java.util.Collection;

import java.util.Collections;

import java.util.Comparator;

import java.util.HashMap;

import java.util.Iterator;

import java.util.List;

import java.util.Map;

import java.util.Set;

import com.google.common.collect.Lists;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.search.common.constants.thriftjava.ThriftLanguage;

import com.twitter.search.common.logging.DebugMessageBuilder;

import com.twitter.search.common.ranking.thriftjava.ThriftFacetFinalSortOrder;

import com.twitter.search.common.schema.earlybird.EarlybirdFieldConstants.EarlybirdFieldConstant;

import com.twitter.search.earlybird.thrift.EarlybirdResponse;

import com.twitter.search.earlybird.thrift.ThriftFacetCount;

import com.twitter.search.earlybird.thrift.ThriftFacetCountMetadata;

import com.twitter.search.earlybird.thrift.ThriftFacetFieldRequest;

import com.twitter.search.earlybird.thrift.ThriftFacetFieldResults;

import com.twitter.search.earlybird.thrift.ThriftFacetRankingMode;

import com.twitter.search.earlybird.thrift.ThriftFacetRequest;

import com.twitter.search.earlybird.thrift.ThriftFacetResults;

import com.twitter.search.earlybird.thrift.ThriftTermResults;

/\*\*

\* A utility class to provide some functions for facets results processing.

\*/

public final class FacetsResultsUtils {

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

private FacetsResultsUtils() {

}

public static class FacetFieldInfo {

public ThriftFacetFieldRequest fieldRequest;

public int totalCounts;

public Map<String, ThriftFacetCount> topFacets;

public List<Map.Entry<ThriftLanguage, Double>> languageHistogramEntries = Lists.newLinkedList();

}

// Only return top languages in the language histogram which sum up to at least this much

// ratio, here we get first 80 percentiles.

public static final double MIN\_PERCENTAGE\_SUM\_REQUIRED = 0.8;

// if a language ratio is over this number, we already return.

public static final double MIN\_PERCENTAGE = 0.01;

/\*\*

\* Prepare facet fields with empty entries and check if we need termStats for filtering.

\* Returns true if termStats filtering is needed (thus the termStats servie call).

\* @param facetRequest The related facet request.

\* @param facetFieldInfoMap The facet field info map to fill, a map from facet type to the facet

\* fiels results info.

\* @return {@code true} if termstats request is needed afterwards.

\*/

public static boolean prepareFieldInfoMap(

ThriftFacetRequest facetRequest,

final Map<String, FacetsResultsUtils.FacetFieldInfo> facetFieldInfoMap) {

boolean termStatsFilteringMode = false;

for (ThriftFacetFieldRequest fieldRequest : facetRequest.getFacetFields()) {

FacetsResultsUtils.FacetFieldInfo info = new FacetsResultsUtils.FacetFieldInfo();

info.fieldRequest = fieldRequest;

facetFieldInfoMap.put(fieldRequest.getFieldName(), info);

if (fieldRequest.getRankingMode() == ThriftFacetRankingMode.FILTER\_WITH\_TERM\_STATISTICS) {

termStatsFilteringMode = true;

}

}

return termStatsFilteringMode;

}

/\*\*

\* Extract information from one ThriftFacetResults into facetFieldInfoMap and userIDWhitelist.

\* @param facetResults Related facets results.

\* @param facetFieldInfoMap The facets field info map to fill, a map from facet type to the facet

\* fiels results info.

\* @param userIDWhitelist The user whitelist to fill.

\*/

public static void fillFacetFieldInfo(

final ThriftFacetResults facetResults,

final Map<String, FacetsResultsUtils.FacetFieldInfo> facetFieldInfoMap,

final Set<Long> userIDWhitelist) {

for (String facetField : facetResults.getFacetFields().keySet()) {

FacetsResultsUtils.FacetFieldInfo info = facetFieldInfoMap.get(facetField);

if (info.topFacets == null) {

info.topFacets = new HashMap<>();

}

ThriftFacetFieldResults results = facetResults.getFacetFields().get(facetField);

if (results.isSetLanguageHistogram()) {

info.languageHistogramEntries.addAll(results.getLanguageHistogram().entrySet());

}

for (ThriftFacetCount newCount : results.getTopFacets()) {

ThriftFacetCount resultCount = info.topFacets.get(newCount.facetLabel);

if (resultCount == null) {

info.topFacets.put(newCount.facetLabel, new ThriftFacetCount(newCount));

} else {

resultCount.setFacetCount(resultCount.facetCount + newCount.facetCount);

resultCount.setSimpleCount(resultCount.simpleCount + newCount.simpleCount);

resultCount.setWeightedCount(resultCount.weightedCount + newCount.weightedCount);

resultCount.setPenaltyCount(resultCount.penaltyCount + newCount.penaltyCount);

// this could pass the old metadata object back or a new merged one.

resultCount.setMetadata(

mergeFacetMetadata(resultCount.getMetadata(), newCount.getMetadata(),

userIDWhitelist));

}

}

info.totalCounts += results.totalCount;

}

}

/\*\*

\* Merge a metadata into an existing one.

\* @param baseMetadata the metadata to merge into.

\* @param metadataUpdate the new metadata to merge.

\* @param userIDWhitelist user id whitelist to filter user id with.

\* @return The updated metadata.

\*/

public static ThriftFacetCountMetadata mergeFacetMetadata(

final ThriftFacetCountMetadata baseMetadata,

final ThriftFacetCountMetadata metadataUpdate,

final Set<Long> userIDWhitelist) {

ThriftFacetCountMetadata mergedMetadata = baseMetadata;

if (metadataUpdate != null) {

String mergedExplanation = null;

if (mergedMetadata != null) {

if (mergedMetadata.maxTweepCred < metadataUpdate.maxTweepCred) {

mergedMetadata.setMaxTweepCred(metadataUpdate.maxTweepCred);

}

if (mergedMetadata.isSetExplanation()) {

mergedExplanation = mergedMetadata.getExplanation();

if (metadataUpdate.isSetExplanation()) {

mergedExplanation += "\n" + metadataUpdate.getExplanation();

}

} else if (metadataUpdate.isSetExplanation()) {

mergedExplanation = metadataUpdate.getExplanation();

}

if (mergedMetadata.getStatusId() == -1) {

if (LOG.isDebugEnabled()) {

LOG.debug("status id in facet count metadata is -1: " + mergedMetadata);

}

mergedMetadata = metadataUpdate;

} else if (metadataUpdate.getStatusId() != -1

&& metadataUpdate.getStatusId() < mergedMetadata.getStatusId()) {

// keep the oldest tweet, ie. the lowest status ID

mergedMetadata = metadataUpdate;

} else if (metadataUpdate.getStatusId() == mergedMetadata.getStatusId()) {

if (mergedMetadata.getTwitterUserId() == -1) {

// in this case we didn't find the user in a previous partition yet

// only update the user if the status id matches

mergedMetadata.setTwitterUserId(metadataUpdate.getTwitterUserId());

mergedMetadata.setDontFilterUser(metadataUpdate.isDontFilterUser());

}

if (!mergedMetadata.isSetStatusLanguage()) {

mergedMetadata.setStatusLanguage(metadataUpdate.getStatusLanguage());

}

}

if (!mergedMetadata.isSetNativePhotoUrl() && metadataUpdate.isSetNativePhotoUrl()) {

mergedMetadata.setNativePhotoUrl(metadataUpdate.getNativePhotoUrl());

}

} else {

mergedMetadata = metadataUpdate;

}

// this will not set an explanation if neither oldMetadata nor metadataUpdate

// had an explanation

if (mergedExplanation != null) {

mergedMetadata.setExplanation(mergedExplanation);

}

if (userIDWhitelist != null) {

// result must not be null now because of the if above

if (mergedMetadata.getTwitterUserId() != -1 && !mergedMetadata.isDontFilterUser()) {

mergedMetadata.setDontFilterUser(

userIDWhitelist.contains(mergedMetadata.getTwitterUserId()));

}

}

}

return mergedMetadata;

}

/\*\*

\* Appends all twimg results to the image results. Optionally resorts the image results if

\* a comparator is passed in.

\* Also computes the sums of totalCount, totalScore, totalPenalty.

\*/

public static void mergeTwimgResults(ThriftFacetResults facetResults,

Comparator<ThriftFacetCount> optionalSortComparator) {

if (facetResults == null || !facetResults.isSetFacetFields()) {

return;

}

ThriftFacetFieldResults imageResults =

facetResults.getFacetFields().get(EarlybirdFieldConstant.IMAGES\_FACET);

ThriftFacetFieldResults twimgResults =

facetResults.getFacetFields().remove(EarlybirdFieldConstant.TWIMG\_FACET);

if (imageResults == null) {

if (twimgResults != null) {

facetResults.getFacetFields().put(EarlybirdFieldConstant.IMAGES\_FACET, twimgResults);

}

return;

}

if (twimgResults != null) {

imageResults.setTotalCount(imageResults.getTotalCount() + twimgResults.getTotalCount());

imageResults.setTotalPenalty(imageResults.getTotalPenalty() + twimgResults.getTotalPenalty());

imageResults.setTotalScore(imageResults.getTotalScore() + twimgResults.getTotalScore());

for (ThriftFacetCount count : twimgResults.getTopFacets()) {

imageResults.addToTopFacets(count);

}

if (optionalSortComparator != null) {

Collections.sort(imageResults.topFacets, optionalSortComparator);

}

}

}

/\*\*

\* Dedup twimg facets.

\*

\* Twimg facet uses the status ID as the facet label, instead of the twimg URL, a.k.a.

\* native photo URL. It is possible to have the same twimg URL appearing in two different

\* facet label (RT style retweet? copy & paste the twimg URL?). Therefore, to dedup twimg

\* facet correctly, we need to look at ThriftFacetCount.metadata.nativePhotoUrl

\*

\* @param dedupSet A set holding the native URLs from the twimg facetFieldResults. By having

\* the caller passing in the set, it allows the caller to dedup the facet

\* across different ThriftFacetFieldResults.

\* @param facetFieldResults The twimg facet field results to be debupped

\* @param debugMessageBuilder

\*/

public static void dedupTwimgFacet(Set<String> dedupSet,

ThriftFacetFieldResults facetFieldResults,

DebugMessageBuilder debugMessageBuilder) {

if (facetFieldResults == null || facetFieldResults.getTopFacets() == null) {

return;

}

Iterator<ThriftFacetCount> iterator = facetFieldResults.getTopFacetsIterator();

while (iterator.hasNext()) {

ThriftFacetCount count = iterator.next();

if (count.isSetMetadata() && count.getMetadata().isSetNativePhotoUrl()) {

String nativeUrl = count.getMetadata().getNativePhotoUrl();

if (dedupSet.contains(nativeUrl)) {

iterator.remove();

debugMessageBuilder.detailed("dedupTwimgFacet removed %s", nativeUrl);

} else {

dedupSet.add(nativeUrl);

}

}

}

}

private static final class LanguageCount {

private final ThriftLanguage lang;

private final double count;

private LanguageCount(ThriftLanguage lang, double count) {

this.lang = lang;

this.count = count;

}

}

/\*\*

\* Calculate the top languages and store them in the results.

\*/

public static void fillTopLanguages(FacetsResultsUtils.FacetFieldInfo info,

final ThriftFacetFieldResults results) {

double sumForLanguage = 0.0;

double[] sums = new double[ThriftLanguage.values().length];

for (Map.Entry<ThriftLanguage, Double> entry : info.languageHistogramEntries) {

sumForLanguage += entry.getValue();

if (entry.getKey() == null) {

// EB might be setting null key for unknown language. SEARCH-1294

continue;

}

sums[entry.getKey().getValue()] += entry.getValue();

}

if (sumForLanguage == 0.0) {

return;

}

List<LanguageCount> langCounts = new ArrayList<>(ThriftLanguage.values().length);

for (int i = 0; i < sums.length; i++) {

if (sums[i] > 0.0) {

// ThriftLanguage.findByValue() might return null, which should fall back to UNKNOWN.

ThriftLanguage lang = ThriftLanguage.findByValue(i);

lang = lang == null ? ThriftLanguage.UNKNOWN : lang;

langCounts.add(new LanguageCount(lang, sums[i]));

}

}

Collections.sort(langCounts, (left, right) -> Double.compare(right.count, left.count));

double percentageSum = 0.0;

Map<ThriftLanguage, Double> languageHistogramMap =

new HashMap<>(langCounts.size());

int numAdded = 0;

for (LanguageCount langCount : langCounts) {

if (langCount.count == 0.0) {

break;

}

double percentage = langCount.count / sumForLanguage;

if (percentageSum > MIN\_PERCENTAGE\_SUM\_REQUIRED

&& percentage < MIN\_PERCENTAGE && numAdded >= 3) {

break;

}

languageHistogramMap.put(langCount.lang, percentage);

percentageSum += percentage;

numAdded++;

}

results.setLanguageHistogram(languageHistogramMap);

}

/\*\*

\* Replace "p.twimg.com/" part of the native photo (twimg) URL with "pbs.twimg.com/media/".

\* We need to do this because of blobstore and it's suppose to be a temporary measure. This

\* code should be removed once we verified that all native photo URL being sent to Search

\* are prefixed with "pbs.twimg.com/media/" and no native photo URL in our index contains

\* "p.twimg.com/"

\*

\* Please see SEARCH-783 and EVENTS-539 for more details.

\*

\* @param response response containing the facet results

\*/

public static void fixNativePhotoUrl(EarlybirdResponse response) {

if (response == null

|| !response.isSetFacetResults()

|| !response.getFacetResults().isSetFacetFields()) {

return;

}

for (Map.Entry<String, ThriftFacetFieldResults> facetMapEntry

: response.getFacetResults().getFacetFields().entrySet()) {

final String facetResultField = facetMapEntry.getKey();

if (EarlybirdFieldConstant.TWIMG\_FACET.equals(facetResultField)

|| EarlybirdFieldConstant.IMAGES\_FACET.equals(facetResultField)) {

ThriftFacetFieldResults facetFieldResults = facetMapEntry.getValue();

for (ThriftFacetCount facetCount : facetFieldResults.getTopFacets()) {

replacePhotoUrl(facetCount.getMetadata());

}

}

}

}

/\*\*

\* Replace "p.twimg.com/" part of the native photo (twimg) URL with "pbs.twimg.com/media/".

\* We need to do this because of blobstore and it's suppose to be a temporary measure. This

\* code should be removed once we verified that all native photo URL being sent to Search

\* are prefixed with "pbs.twimg.com/media/" and no native photo URL in our index contains

\* "p.twimg.com/"

\*

\* Please see SEARCH-783 and EVENTS-539 for more details.

\*

\* @param termResultsCollection collection of ThriftTermResults containing the native photo URL

\*/

public static void fixNativePhotoUrl(Collection<ThriftTermResults> termResultsCollection) {

if (termResultsCollection == null) {

return;

}

for (ThriftTermResults termResults : termResultsCollection) {

if (!termResults.isSetMetadata()) {

continue;

}

replacePhotoUrl(termResults.getMetadata());

}

}

/\*\*

\* Helper function for fixNativePhotoUrl()

\*/

private static void replacePhotoUrl(ThriftFacetCountMetadata metadata) {

if (metadata != null

&& metadata.isSetNativePhotoUrl()) {

String nativePhotoUrl = metadata.getNativePhotoUrl();

nativePhotoUrl = nativePhotoUrl.replace("://p.twimg.com/", "://pbs.twimg.com/media/");

metadata.setNativePhotoUrl(nativePhotoUrl);

}

}

/\*\*

\* Deepcopy of an EarlybirdResponse without explanation

\*/

public static EarlybirdResponse deepCopyWithoutExplanation(EarlybirdResponse facetsResponse) {

if (facetsResponse == null) {

return null;

} else if (!facetsResponse.isSetFacetResults()

|| facetsResponse.getFacetResults().getFacetFieldsSize() == 0) {

return facetsResponse.deepCopy();

}

EarlybirdResponse copy = facetsResponse.deepCopy();

for (Map.Entry<String, ThriftFacetFieldResults> entry

: copy.getFacetResults().getFacetFields().entrySet()) {

if (entry.getValue().getTopFacetsSize() > 0) {

for (ThriftFacetCount fc : entry.getValue().getTopFacets()) {

fc.getMetadata().unsetExplanation();

}

}

}

return copy;

}

/\*\*

\* Returns a comparator used to compare facet counts by calling

\* getFacetCountComparator(ThriftFacetFinalSortOrder). The sort order is determined by

\* the facetRankingOptions on the facet request.

\*/

public static Comparator<ThriftFacetCount> getFacetCountComparator(

ThriftFacetRequest facetRequest) {

ThriftFacetFinalSortOrder sortOrder = ThriftFacetFinalSortOrder.SCORE;

if (facetRequest.isSetFacetRankingOptions()

&& facetRequest.getFacetRankingOptions().isSetFinalSortOrder()) {

sortOrder = facetRequest.getFacetRankingOptions().getFinalSortOrder();

}

return getFacetCountComparator(sortOrder);

}

/\*\*

\* Returns a comparator using the specified order.

\*/

public static Comparator<ThriftFacetCount> getFacetCountComparator(

ThriftFacetFinalSortOrder sortOrder) {

switch (sortOrder) {

case SIMPLE\_COUNT: return SIMPLE\_COUNT\_COMPARATOR;

case SCORE: return SCORE\_COMPARATOR;

case CREATED\_AT: return CREATED\_AT\_COMPARATOR;

case WEIGHTED\_COUNT: return WEIGHTED\_COUNT\_COMPARATOR;

default: return SCORE\_COMPARATOR;

}

}

private static final Comparator<ThriftFacetCount> SIMPLE\_COUNT\_COMPARATOR =

(count1, count2) -> {

if (count1.simpleCount > count2.simpleCount) {

return 1;

} else if (count1.simpleCount < count2.simpleCount) {

return -1;

}

return count1.facetLabel.compareTo(count2.facetLabel);

};

private static final Comparator<ThriftFacetCount> WEIGHTED\_COUNT\_COMPARATOR =

(count1, count2) -> {

if (count1.weightedCount > count2.weightedCount) {

return 1;

} else if (count1.weightedCount < count2.weightedCount) {

return -1;

}

return SIMPLE\_COUNT\_COMPARATOR.compare(count1, count2);

};

private static final Comparator<ThriftFacetCount> SCORE\_COMPARATOR =

(count1, count2) -> {

if (count1.score > count2.score) {

return 1;

} else if (count1.score < count2.score) {

return -1;

}

return SIMPLE\_COUNT\_COMPARATOR.compare(count1, count2);

};

private static final Comparator<ThriftFacetCount> CREATED\_AT\_COMPARATOR =

(count1, count2) -> {

if (count1.isSetMetadata() && count1.getMetadata().isSetCreated\_at()

&& count2.isSetMetadata() && count2.getMetadata().isSetCreated\_at()) {

// more recent items have higher created\_at values

if (count1.getMetadata().getCreated\_at() > count2.getMetadata().getCreated\_at()) {

return 1;

} else if (count1.getMetadata().getCreated\_at() < count2.getMetadata().getCreated\_at()) {

return -1;

}

}

return SCORE\_COMPARATOR.compare(count1, count2);

};

}