package com.twitter.search.earlybird\_root.mergers;

import java.util.ArrayList;

import java.util.Collection;

import java.util.Collections;

import java.util.List;

import java.util.Map;

import javax.annotation.Nonnull;

import javax.annotation.Nullable;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Preconditions;

import com.google.common.collect.Lists;

import com.google.common.collect.Maps;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

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

import com.twitter.search.common.util.earlybird.FacetsResultsUtils;

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

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

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

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

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

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

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

/\*\*

\* Takes multiple successful EarlybirdResponses and merges them.

\*/

public class ThriftTermResultsMerger {

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

private static final SearchCounter BIN\_ID\_GAP\_COUNTER =

SearchCounter.export("thrift\_term\_results\_merger\_found\_gap\_in\_bin\_ids");

private static final SearchCounter MIN\_COMPLETE\_BIN\_ID\_ADJUSTED\_NULL =

SearchCounter.export("thrift\_term\_results\_merger\_min\_complete\_bin\_id\_adjusted\_null");

private static final SearchCounter MIN\_COMPLETE\_BIN\_ID\_NULL\_WITHOUT\_BINS =

SearchCounter.export("thrift\_term\_results\_merger\_min\_complete\_bin\_id\_null\_without\_bins");

private static final SearchCounter MIN\_COMPLETE\_BIN\_ID\_OUT\_OF\_RANGE =

SearchCounter.export("thrift\_term\_results\_merger\_min\_complete\_bin\_id\_out\_of\_range");

private static final SearchCounter RESPONSE\_WITHOUT\_DRIVING\_QUERY\_HIT =

SearchCounter.export("response\_without\_driving\_query\_hit");

private static final ThriftTermRequest GLOBAL\_COUNT\_REQUEST =

new ThriftTermRequest().setFieldName("").setTerm("");

/\*\*

\* Sorted list of the most recent (and contiguous) numBins binIds across all responses.

\* Expected to be an empty list if this request did not ask for histograms, or if it

\* did ask for histograms for 0 numBins.

\*/

@Nonnull

private final List<Integer> mostRecentBinIds;

/\*\*

\* The first binId in the {@link #mostRecentBinIds} list. This value is not meant to be used in

\* case mostRecentBinIds is an empty list.

\*/

private final int firstBinId;

/\*\*

\* For each unique ThriftTermRequest, stores an array of the total counts for all the binIds

\* that we will return, summed up across all earlybird responses.

\*

\* The values in each totalCounts array correspond to the binIds in the

\* {@link #mostRecentBinIds} list.

\*

\* Key: thrift term request.

\* Value: array of the total counts summed up across all earlybird responses for the key's

\* term request, corresponding to the binIds in {@link #mostRecentBinIds}.

\*/

private final Map<ThriftTermRequest, int[]> mergedTermRequestTotalCounts = Maps.newHashMap();

/\*\*

\* The set of all unique binIds that we are merging.

\*/

private final Map<ThriftTermRequest, ThriftTermResults> termResultsMap = Maps.newHashMap();

private final ThriftHistogramSettings histogramSettings;

/\*\*

\* Only relevant for merging responses with histogram settings.

\* This will be null either if (1) the request is not asking for histograms at all, or if

\* (2) numBins was set to 0 (and no bin can be considered complete).

\* If not null, the minCompleteBinId will be computed as the max over all merged responses'

\* minCompleteBinId's.

\*/

@Nullable

private final Integer minCompleteBinId;

/\*\*

\* Create merger with collections of results to merge

\*/

public ThriftTermResultsMerger(Collection<EarlybirdResponse> termStatsResults,

ThriftHistogramSettings histogramSettings) {

this.histogramSettings = histogramSettings;

Collection<EarlybirdResponse> filteredTermStatsResults =

filterOutEmptyEarlybirdResponses(termStatsResults);

this.mostRecentBinIds = findMostRecentBinIds(histogramSettings, filteredTermStatsResults);

this.firstBinId = mostRecentBinIds.isEmpty()

? Integer.MAX\_VALUE // Should not be used if mostRecentBinIds is empty.

: mostRecentBinIds.get(0);

List<Integer> minCompleteBinIds =

Lists.newArrayListWithCapacity(filteredTermStatsResults.size());

for (EarlybirdResponse response : filteredTermStatsResults) {

Preconditions.checkState(response.getResponseCode() == EarlybirdResponseCode.SUCCESS,

"Unsuccessful responses should not be given to ThriftTermResultsMerger.");

Preconditions.checkState(response.getTermStatisticsResults() != null,

"Response given to ThriftTermResultsMerger has no termStatisticsResults.");

ThriftTermStatisticsResults termStatisticsResults = response.getTermStatisticsResults();

List<Integer> binIds = termStatisticsResults.getBinIds();

for (Map.Entry<ThriftTermRequest, ThriftTermResults> entry

: termStatisticsResults.getTermResults().entrySet()) {

ThriftTermRequest termRequest = entry.getKey();

ThriftTermResults termResults = entry.getValue();

adjustTotalCount(termResults, binIds);

addTotalCountData(termRequest, termResults);

if (histogramSettings != null) {

Preconditions.checkState(termStatisticsResults.isSetBinIds());

addHistogramData(termRequest, termResults, termStatisticsResults.getBinIds());

}

}

if (histogramSettings != null) {

addMinCompleteBinId(minCompleteBinIds, response);

}

}

minCompleteBinId = minCompleteBinIds.isEmpty() ? null : Collections.max(minCompleteBinIds);

}

/\*\*

\* Take out any earlybird responses that we know did not match anything relevant to the query,

\* and may have erroneous binIds.

\*/

private Collection<EarlybirdResponse> filterOutEmptyEarlybirdResponses(

Collection<EarlybirdResponse> termStatsResults) {

List<EarlybirdResponse> emptyResponses = Lists.newArrayList();

List<EarlybirdResponse> nonEmptyResponses = Lists.newArrayList();

for (EarlybirdResponse response : termStatsResults) {

// Guard against erroneously merging and returning 0 counts when we actually have data to

// return from other partitions.

// When a query doesn't match anything at all on an earlybird, the binIds that are returned

// do not correspond at all to the actual query, and are just based on the data range on the

// earlybird itself.

// We can identify these responses as (1) being non-early terminated, and (2) having 0

// hits processed.

if (isTermStatResponseEmpty(response)) {

emptyResponses.add(response);

} else {

nonEmptyResponses.add(response);

}

}

// If all responses were "empty", we will just use those to merge into a new set of empty

// responses, using the binIds provided.

return nonEmptyResponses.isEmpty() ? emptyResponses : nonEmptyResponses;

}

private boolean isTermStatResponseEmpty(EarlybirdResponse response) {

return response.isSetSearchResults()

&& (response.getSearchResults().getNumHitsProcessed() == 0

|| drivingQueryHasNoHits(response))

&& response.isSetEarlyTerminationInfo()

&& !response.getEarlyTerminationInfo().isEarlyTerminated();

}

/\*\*

\* If the global count bins are all 0, then we know the driving query has no hits.

\* This check is added as a short term solution for SEARCH-5476. This short term fix requires

\* the client to set the includeGlobalCounts to kick in.

\*/

private boolean drivingQueryHasNoHits(EarlybirdResponse response) {

ThriftTermStatisticsResults termStatisticsResults = response.getTermStatisticsResults();

if (termStatisticsResults == null || termStatisticsResults.getTermResults() == null) {

// If there's no term stats response, be conservative and return false.

return false;

} else {

ThriftTermResults globalCounts =

termStatisticsResults.getTermResults().get(GLOBAL\_COUNT\_REQUEST);

if (globalCounts == null) {

// We cannot tell if driving query has no hits, be conservative and return false.

return false;

} else {

for (Integer i : globalCounts.getHistogramBins()) {

if (i > 0) {

return false;

}

}

RESPONSE\_WITHOUT\_DRIVING\_QUERY\_HIT.increment();

return true;

}

}

}

private static List<Integer> findMostRecentBinIds(

ThriftHistogramSettings histogramSettings,

Collection<EarlybirdResponse> filteredTermStatsResults) {

Integer largestFirstBinId = null;

List<Integer> binIdsToUse = null;

if (histogramSettings != null) {

int numBins = histogramSettings.getNumBins();

for (EarlybirdResponse response : filteredTermStatsResults) {

ThriftTermStatisticsResults termStatisticsResults = response.getTermStatisticsResults();

Preconditions.checkState(termStatisticsResults.getBinIds().size() == numBins,

"expected all results to have the same numBins. "

+ "request numBins: %s, response numBins: %s",

numBins, termStatisticsResults.getBinIds().size());

if (termStatisticsResults.getBinIds().size() > 0) {

Integer firstBinId = termStatisticsResults.getBinIds().get(0);

if (largestFirstBinId == null

|| largestFirstBinId.intValue() < firstBinId.intValue()) {

largestFirstBinId = firstBinId;

binIdsToUse = termStatisticsResults.getBinIds();

}

}

}

}

return binIdsToUse == null

? Collections.<Integer>emptyList()

// Just in case, make a copy of the binIds so that we don't reuse the same list from one

// of the responses we're merging.

: Lists.newArrayList(binIdsToUse);

}

private void addMinCompleteBinId(List<Integer> minCompleteBinIds,

EarlybirdResponse response) {

Preconditions.checkNotNull(histogramSettings);

ThriftTermStatisticsResults termStatisticsResults = response.getTermStatisticsResults();

if (termStatisticsResults.isSetMinCompleteBinId()) {

// This is the base case. Early terminated or not, this is the proper minCompleteBinId

// that we're told to use for this response.

minCompleteBinIds.add(termStatisticsResults.getMinCompleteBinId());

} else if (termStatisticsResults.getBinIds().size() > 0) {

// This is the case where no bins were complete. For the purposes of merging, we need to

// mark all the binIds in this response as non-complete by marking the "max(binId)+1" as the

// last complete bin.

// When returning the merged response, we still have a guard for the resulting

// minCompleteBinId being outside of the binIds range, and will set the returned

// minCompleteBinId value to null, if this response's binIds end up being used as the most

// recent ones, and we need to signify that none of the bins are complete.

int binSize = termStatisticsResults.getBinIds().size();

Integer maxBinId = termStatisticsResults.getBinIds().get(binSize - 1);

minCompleteBinIds.add(maxBinId + 1);

LOG.debug("Adjusting null minCompleteBinId for response: {}, histogramSettings {}",

response, histogramSettings);

MIN\_COMPLETE\_BIN\_ID\_ADJUSTED\_NULL.increment();

} else {

// This should only happen in the case where numBins is set to 0.

Preconditions.checkState(histogramSettings.getNumBins() == 0,

"Expected numBins set to 0. response: %s", response);

Preconditions.checkState(minCompleteBinIds.isEmpty(),

"minCompleteBinIds: %s", minCompleteBinIds);

LOG.debug("Got null minCompleteBinId with no bins for response: {}, histogramSettings {}",

response, histogramSettings);

MIN\_COMPLETE\_BIN\_ID\_NULL\_WITHOUT\_BINS.increment();

}

}

private void addTotalCountData(ThriftTermRequest request, ThriftTermResults results) {

ThriftTermResults termResults = termResultsMap.get(request);

if (termResults == null) {

termResultsMap.put(request, results);

} else {

termResults.setTotalCount(termResults.getTotalCount() + results.getTotalCount());

if (termResults.isSetMetadata()) {

termResults.setMetadata(

FacetsResultsUtils.mergeFacetMetadata(termResults.getMetadata(),

results.getMetadata(), null));

}

}

}

/\*\*

\* Set results.totalCount to the sum of hits in only the bins that will be returned in

\* the merged response.

\*/

private void adjustTotalCount(ThriftTermResults results, List<Integer> binIds) {

int adjustedTotalCount = 0;

List<Integer> histogramBins = results.getHistogramBins();

if ((binIds != null) && (histogramBins != null)) {

Preconditions.checkState(

histogramBins.size() == binIds.size(),

"Expected ThriftTermResults to have the same number of histogramBins as binIds set in "

+ " ThriftTermStatisticsResults. ThriftTermResults.histogramBins: %s, "

+ " ThriftTermStatisticsResults.binIds: %s.",

histogramBins, binIds);

for (int i = 0; i < binIds.size(); ++i) {

if (binIds.get(i) >= firstBinId) {

adjustedTotalCount += histogramBins.get(i);

}

}

}

results.setTotalCount(adjustedTotalCount);

}

private void addHistogramData(ThriftTermRequest request,

ThriftTermResults results,

List<Integer> binIds) {

int[] requestTotalCounts = mergedTermRequestTotalCounts.get(request);

if (requestTotalCounts == null) {

requestTotalCounts = new int[mostRecentBinIds.size()];

mergedTermRequestTotalCounts.put(request, requestTotalCounts);

}

// Only consider these results if they fall into the mostRecentBinIds range.

//

// The list of returned binIds is expected to be both sorted (in ascending order), and

// contiguous, which allows us to use firstBinId to check if it overlaps with the

// mostRecentBinIds range.

if (binIds.size() > 0 && binIds.get(binIds.size() - 1) >= firstBinId) {

int firstBinIndex;

if (binIds.get(0) == firstBinId) {

// This should be the common case when all partitions have the same binIds,

// no need to do a binary search.

firstBinIndex = 0;

} else {

// The firstBinId must be in the binIds range. We can find it using binary search since

// binIds are sorted.

firstBinIndex = Collections.binarySearch(binIds, firstBinId);

Preconditions.checkState(firstBinIndex >= 0,

"Expected to find firstBinId (%s) in the result binIds: %s, "

+ "histogramSettings: %s, termRequest: %s",

firstBinId, binIds, histogramSettings, request);

}

// Skip binIds that are before the smallest binId that we will use in the merged results.

for (int i = firstBinIndex; i < binIds.size(); i++) {

final Integer currentBinValue = results.getHistogramBins().get(i);

requestTotalCounts[i - firstBinIndex] += currentBinValue.intValue();

}

}

}

/\*\*

\* Return a new ThriftTermStatisticsResults with the total counts merged, and if enabled,

\* histogram bins merged.

\*/

public ThriftTermStatisticsResults merge() {

ThriftTermStatisticsResults results = new ThriftTermStatisticsResults(termResultsMap);

if (histogramSettings != null) {

mergeHistogramBins(results);

}

return results;

}

/\*\*

\* Takes multiple histogram results and merges them so:

\* 1) Counts for the same binId (represents the time) and term are summed

\* 2) All results are re-indexed to use the most recent bins found from the union of all bins

\*/

private void mergeHistogramBins(ThriftTermStatisticsResults mergedResults) {

mergedResults.setBinIds(mostRecentBinIds);

mergedResults.setHistogramSettings(histogramSettings);

setMinCompleteBinId(mergedResults);

useMostRecentBinsForEachThriftTermResults();

}

private void setMinCompleteBinId(ThriftTermStatisticsResults mergedResults) {

if (mostRecentBinIds.isEmpty()) {

Preconditions.checkState(minCompleteBinId == null);

// This is the case where the requested numBins is set to 0. We don't have any binIds,

// and the minCompleteBinId has to be unset.

LOG.debug("Empty binIds returned for mergedResults: {}", mergedResults);

} else {

Preconditions.checkNotNull(minCompleteBinId);

Integer maxBinId = mostRecentBinIds.get(mostRecentBinIds.size() - 1);

if (minCompleteBinId <= maxBinId) {

mergedResults.setMinCompleteBinId(minCompleteBinId);

} else {

// Leaving the minCompleteBinId unset as it is outside the range of the returned binIds.

LOG.debug("Computed minCompleteBinId: {} is out of maxBinId: {} for mergedResults: {}",

minCompleteBinId, mergedResults);

MIN\_COMPLETE\_BIN\_ID\_OUT\_OF\_RANGE.increment();

}

}

}

/\*\*

\* Check that the binIds we are using are contiguous. Increment the provided stat if we find

\* a gap, as we don't expect to find any.

\* See: SEARCH-4362

\*

\* @param sortedBinIds most recent numBins sorted binIds.

\* @param binIdGapCounter stat to increment if we see a gap in the binId range.

\*/

@VisibleForTesting

static void checkForBinIdGaps(List<Integer> sortedBinIds, SearchCounter binIdGapCounter) {

for (int i = sortedBinIds.size() - 1; i > 0; i--) {

final Integer currentBinId = sortedBinIds.get(i);

final Integer previousBinId = sortedBinIds.get(i - 1);

if (previousBinId < currentBinId - 1) {

binIdGapCounter.increment();

break;

}

}

}

/\*\*

\* Returns a view containing only the last N items from the list

\*/

private static <E> List<E> takeLastN(List<E> lst, int n) {

Preconditions.checkArgument(n <= lst.size(),

"Attempting to take more elements than the list has. List size: %s, n: %s", lst.size(), n);

return lst.subList(lst.size() - n, lst.size());

}

private void useMostRecentBinsForEachThriftTermResults() {

for (Map.Entry<ThriftTermRequest, ThriftTermResults> entry : termResultsMap.entrySet()) {

ThriftTermRequest request = entry.getKey();

ThriftTermResults results = entry.getValue();

List<Integer> histogramBins = Lists.newArrayList();

results.setHistogramBins(histogramBins);

int[] requestTotalCounts = mergedTermRequestTotalCounts.get(request);

Preconditions.checkNotNull(requestTotalCounts);

for (int totalCount : requestTotalCounts) {

histogramBins.add(totalCount);

}

}

}

/\*\*

\* Merges search stats from several earlybird responses and puts them in

\* {@link ThriftSearchResults} structure.

\*

\* @param responses earlybird responses to merge the search stats from

\* @return merged search stats inside of {@link ThriftSearchResults} structure

\*/

public static ThriftSearchResults mergeSearchStats(Collection<EarlybirdResponse> responses) {

int numHitsProcessed = 0;

int numPartitionsEarlyTerminated = 0;

for (EarlybirdResponse response : responses) {

ThriftSearchResults searchResults = response.getSearchResults();

if (searchResults != null) {

numHitsProcessed += searchResults.getNumHitsProcessed();

numPartitionsEarlyTerminated += searchResults.getNumPartitionsEarlyTerminated();

}

}

ThriftSearchResults searchResults = new ThriftSearchResults(new ArrayList<>());

searchResults.setNumHitsProcessed(numHitsProcessed);

searchResults.setNumPartitionsEarlyTerminated(numPartitionsEarlyTerminated);

return searchResults;

}

}