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

import java.util.Collections;

import java.util.List;

import javax.annotation.Nullable;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Preconditions;

import com.google.common.cache.CacheBuilder;

import com.google.common.cache.CacheLoader;

import com.google.common.cache.LoadingCache;

import com.google.common.collect.Lists;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.common.collections.Pair;

import com.twitter.common.quantity.Amount;

import com.twitter.common.quantity.Time;

import com.twitter.common.util.Clock;

import com.twitter.search.common.futures.Futures;

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

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

import com.twitter.search.common.query.thriftjava.EarlyTerminationInfo;

import com.twitter.search.common.relevance.utils.ResultComparators;

import com.twitter.search.common.search.EarlyTerminationState;

import com.twitter.search.common.util.FinagleUtil;

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

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

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

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

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

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

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

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

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

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

import com.twitter.search.earlybird\_root.common.EarlybirdFeatureSchemaMerger;

import com.twitter.search.earlybird\_root.common.EarlybirdRequestContext;

import com.twitter.search.earlybird\_root.common.EarlybirdServiceResponse;

import com.twitter.util.Function;

import com.twitter.util.Function0;

import com.twitter.util.Future;

/\*\* Utility functions for merging recency and relevance results. \*/

public class SuperRootResponseMerger {

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

private static final String ALL\_STATS\_PREFIX = "superroot\_response\_merger\_";

private static final SearchCounter FULL\_ARCHIVE\_MIN\_ID\_GREATER\_THAN\_REALTIME\_MIN\_ID =

SearchCounter.export("full\_archive\_min\_id\_greater\_than\_realtime\_min\_id");

private static final String ERROR\_FORMAT = "%s%s\_errors\_from\_cluster\_%s\_%s";

private final ThriftSearchRankingMode rankingMode;

private final EarlybirdFeatureSchemaMerger featureSchemaMerger;

private final String featureStatPrefix;

private final Clock clock;

private final String rankingModeStatPrefix;

private final SearchCounter mergedResponseSearchResultsNotSet;

private final SearchCounter invalidMinStatusId;

private final SearchCounter invalidMaxStatusId;

private final SearchCounter noMinIds;

private final SearchCounter noMaxIds;

private final SearchCounter mergedResponses;

private final SearchCounter mergedResponsesWithExactDups;

private final LoadingCache<Pair<ThriftTweetSource, ThriftTweetSource>, SearchCounter> dupsStats;

private static final EarlybirdResponse EMPTY\_RESPONSE =

new EarlybirdResponse(EarlybirdResponseCode.SUCCESS, 0)

.setSearchResults(new ThriftSearchResults()

.setResults(Lists.<ThriftSearchResult>newArrayList()));

/\*\*

\* Creates a new SuperRootResponseMerger instance.

\* @param rankingMode The ranking mode to use when merging results.

\* @param featureSchemaMerger The merger that can merge feature schema from different tiers.

\* @param clock The clock that will be used to merge results.

\*/

public SuperRootResponseMerger(ThriftSearchRankingMode rankingMode,

EarlybirdFeatureSchemaMerger featureSchemaMerger,

Clock clock) {

this.rankingModeStatPrefix = rankingMode.name().toLowerCase();

this.rankingMode = rankingMode;

this.featureSchemaMerger = featureSchemaMerger;

this.clock = clock;

this.featureStatPrefix = "superroot\_" + rankingMode.name().toLowerCase();

mergedResponseSearchResultsNotSet = SearchCounter.export(

ALL\_STATS\_PREFIX + rankingModeStatPrefix + "\_merged\_response\_search\_results\_not\_set");

invalidMinStatusId =

SearchCounter.export(ALL\_STATS\_PREFIX + rankingModeStatPrefix + "\_invalid\_min\_status\_id");

invalidMaxStatusId =

SearchCounter.export(ALL\_STATS\_PREFIX + rankingModeStatPrefix + "\_invalid\_max\_status\_id");

noMinIds = SearchCounter.export(ALL\_STATS\_PREFIX + rankingModeStatPrefix + "\_no\_min\_ids");

noMaxIds = SearchCounter.export(ALL\_STATS\_PREFIX + rankingModeStatPrefix + "\_no\_max\_ids");

mergedResponses = SearchCounter.export(ALL\_STATS\_PREFIX + rankingModeStatPrefix

+ "\_merged\_responses");

mergedResponsesWithExactDups =

SearchCounter.export(ALL\_STATS\_PREFIX + rankingModeStatPrefix

+ "\_merged\_responses\_with\_exact\_dups");

dupsStats = CacheBuilder.newBuilder()

.build(new CacheLoader<Pair<ThriftTweetSource, ThriftTweetSource>, SearchCounter>() {

@Override

public SearchCounter load(Pair<ThriftTweetSource, ThriftTweetSource> key) {

return SearchCounter.export(

ALL\_STATS\_PREFIX + rankingModeStatPrefix + "\_merged\_responses\_with\_exact\_dups\_"

+ key.getFirst().name() + "\_" + key.getSecond().name());

}

});

}

private void incrErrorCount(String cluster, @Nullable EarlybirdResponse response) {

String cause;

if (response != null) {

cause = response.getResponseCode().name().toLowerCase();

} else {

cause = "null\_response";

}

String statName = String.format(

ERROR\_FORMAT, ALL\_STATS\_PREFIX, rankingModeStatPrefix, cluster, cause

);

SearchCounter.export(statName).increment();

}

/\*\*

\* Merges the given response futures.

\*

\* @param earlybirdRequestContext The earlybird request.

\* @param realtimeResponseFuture The response from the realtime cluster.

\* @param protectedResponseFuture The response from the protected cluster.

\* @param fullArchiveResponseFuture The response from the full archive cluster.

\* @return A future with the merged results.

\*/

public Future<EarlybirdResponse> mergeResponseFutures(

final EarlybirdRequestContext earlybirdRequestContext,

final Future<EarlybirdServiceResponse> realtimeResponseFuture,

final Future<EarlybirdServiceResponse> protectedResponseFuture,

final Future<EarlybirdServiceResponse> fullArchiveResponseFuture) {

Future<EarlybirdResponse> mergedResponseFuture = Futures.map(

realtimeResponseFuture, protectedResponseFuture, fullArchiveResponseFuture,

new Function0<EarlybirdResponse>() {

@Override

public EarlybirdResponse apply() {

// If the realtime response is not valid, return an error response.

// Also, the realtime service should always be called.

EarlybirdServiceResponse realtimeResponse = Futures.get(realtimeResponseFuture);

if (realtimeResponse.getServiceState().serviceWasRequested()

&& (!realtimeResponse.getServiceState().serviceWasCalled()

|| !EarlybirdResponseMergeUtil.isValidResponse(

realtimeResponse.getResponse()))) {

incrErrorCount("realtime", realtimeResponse.getResponse());

return EarlybirdResponseMergeUtil.transformInvalidResponse(

realtimeResponse.getResponse(), "realtime");

}

// If we have a protected response and it's not valid, return an error response.

EarlybirdServiceResponse protectedResponse = Futures.get(protectedResponseFuture);

if (protectedResponse.getServiceState().serviceWasCalled()) {

if (!EarlybirdResponseMergeUtil.isValidResponse(protectedResponse.getResponse())) {

incrErrorCount("protected", protectedResponse.getResponse());

return EarlybirdResponseMergeUtil.transformInvalidResponse(

protectedResponse.getResponse(), "protected");

}

}

// If we have a full archive response, check if it's valid.

EarlybirdServiceResponse fullArchiveResponse = Futures.get(fullArchiveResponseFuture);

boolean archiveHasError =

fullArchiveResponse.getServiceState().serviceWasCalled()

&& !EarlybirdResponseMergeUtil.isValidResponse(fullArchiveResponse.getResponse());

// Merge the responses.

EarlybirdResponse mergedResponse = mergeResponses(

earlybirdRequestContext,

realtimeResponse.getResponse(),

protectedResponse.getResponse(),

fullArchiveResponse.getResponse());

// If the realtime clusters didn't return any results, and the full archive cluster

// returned an error response, return an error merged response.

if (archiveHasError && !EarlybirdResponseUtil.hasResults(mergedResponse)) {

incrErrorCount("full\_archive", fullArchiveResponse.getResponse());

return EarlybirdResponseMergeUtil.failedEarlybirdResponse(

fullArchiveResponse.getResponse().getResponseCode(),

"realtime clusters had no results and archive cluster response had error");

}

// Corner case: the realtime response could have exactly numRequested results, and could

// be exhausted (not early-terminated). In this case, the request should not have been

// sent to the full archive cluster.

// - If the full archive cluster is not available, or was not requested, then we don't

// need to change anything.

// - If the full archive cluster is available and was requested (but wasn't hit

// because we found enough results in the realtime cluster), then we should set the

// early-termination flag on the merged response, to indicate that we potentially

// have more results for this query in our index.

if ((fullArchiveResponse.getServiceState()

== EarlybirdServiceResponse.ServiceState.SERVICE\_NOT\_CALLED)

&& !EarlybirdResponseUtil.isEarlyTerminated(realtimeResponse.getResponse())) {

EarlyTerminationInfo earlyTerminationInfo = new EarlyTerminationInfo(true);

earlyTerminationInfo.setEarlyTerminationReason(

EarlyTerminationState.TERMINATED\_NUM\_RESULTS\_EXCEEDED.getTerminationReason());

mergedResponse.setEarlyTerminationInfo(earlyTerminationInfo);

}

// If we've exhausted all clusters, set the minSearchedStatusID to 0.

if (!EarlybirdResponseUtil.isEarlyTerminated(mergedResponse)) {

mergedResponse.getSearchResults().setMinSearchedStatusID(0);

}

return mergedResponse;

}

});

// Handle all merging exceptions.

return handleResponseException(mergedResponseFuture,

"Exception thrown while merging responses.");

}

/\*\*

\* Merge the results in the given responses.

\*

\* @param earlybirdRequestContext The earlybird request context.

\* @param realtimeResponse The response from the realtime cluster.

\* @param protectedResponse The response from the protected cluster.

\* @param fullArchiveResponse The response from the full archive cluster.

\* @return The merged response.

\*/

private EarlybirdResponse mergeResponses(

EarlybirdRequestContext earlybirdRequestContext,

@Nullable EarlybirdResponse realtimeResponse,

@Nullable EarlybirdResponse protectedResponse,

@Nullable EarlybirdResponse fullArchiveResponse) {

EarlybirdRequest request = earlybirdRequestContext.getRequest();

ThriftSearchQuery searchQuery = request.getSearchQuery();

int numResultsRequested;

if (request.isSetNumResultsToReturnAtRoot()) {

numResultsRequested = request.getNumResultsToReturnAtRoot();

} else {

numResultsRequested = searchQuery.getNumResults();

}

Preconditions.checkState(numResultsRequested > 0);

EarlybirdResponse mergedResponse = EMPTY\_RESPONSE.deepCopy();

if ((realtimeResponse != null)

&& (realtimeResponse.getResponseCode() != EarlybirdResponseCode.TIER\_SKIPPED)) {

mergedResponse = realtimeResponse.deepCopy();

}

if (!mergedResponse.isSetSearchResults()) {

mergedResponseSearchResultsNotSet.increment();

mergedResponse.setSearchResults(

new ThriftSearchResults(Lists.<ThriftSearchResult>newArrayList()));

}

// If either the realtime or the full archive response is early-terminated, we want the merged

// response to be early-terminated too. The early-termination flag from the realtime response

// carries over to the merged response, because mergedResponse is just a deep copy of the

// realtime response. So we only need to check the early-termination flag of the full archive

// response.

if ((fullArchiveResponse != null)

&& EarlybirdResponseUtil.isEarlyTerminated(fullArchiveResponse)) {

mergedResponse.setEarlyTerminationInfo(fullArchiveResponse.getEarlyTerminationInfo());

}

// If realtime has empty results and protected has some results then we copy the early

// termination information if that is present

if (protectedResponse != null

&& mergedResponse.getSearchResults().getResults().isEmpty()

&& !protectedResponse.getSearchResults().getResults().isEmpty()

&& EarlybirdResponseUtil.isEarlyTerminated(protectedResponse)) {

mergedResponse.setEarlyTerminationInfo(protectedResponse.getEarlyTerminationInfo());

}

// Merge the results.

List<ThriftSearchResult> mergedResults = mergeResults(

numResultsRequested, realtimeResponse, protectedResponse, fullArchiveResponse);

// Trim the merged results if necessary.

boolean resultsTrimmed = false;

if (mergedResults.size() > numResultsRequested

&& !(searchQuery.isSetRelevanceOptions()

&& searchQuery.getRelevanceOptions().isReturnAllResults())) {

// If we have more results than requested, trim the result list and re-adjust

// minSearchedStatusID.

mergedResults = mergedResults.subList(0, numResultsRequested);

// Mark early termination in merged response

if (!EarlybirdResponseUtil.isEarlyTerminated(mergedResponse)) {

EarlyTerminationInfo earlyTerminationInfo = new EarlyTerminationInfo(true);

earlyTerminationInfo.setEarlyTerminationReason(

EarlyTerminationState.TERMINATED\_NUM\_RESULTS\_EXCEEDED.getTerminationReason());

mergedResponse.setEarlyTerminationInfo(earlyTerminationInfo);

}

resultsTrimmed = true;

}

mergedResponse.getSearchResults().setResults(mergedResults);

featureSchemaMerger.mergeFeatureSchemaAcrossClusters(

earlybirdRequestContext,

mergedResponse,

featureStatPrefix,

realtimeResponse,

protectedResponse,

fullArchiveResponse);

// Set the minSearchedStatusID and maxSearchedStatusID fields on the merged response.

setMinSearchedStatusId(mergedResponse, realtimeResponse, protectedResponse, fullArchiveResponse,

resultsTrimmed);

setMaxSearchedStatusId(mergedResponse, realtimeResponse, protectedResponse,

fullArchiveResponse);

int numRealtimeSearchedSegments =

(realtimeResponse != null && realtimeResponse.isSetNumSearchedSegments())

? realtimeResponse.getNumSearchedSegments()

: 0;

int numProtectedSearchedSegments =

(protectedResponse != null && protectedResponse.isSetNumSearchedSegments())

? protectedResponse.getNumSearchedSegments()

: 0;

int numArchiveSearchedSegments =

(fullArchiveResponse != null && fullArchiveResponse.isSetNumSearchedSegments())

? fullArchiveResponse.getNumSearchedSegments()

: 0;

mergedResponse.setNumSearchedSegments(

numRealtimeSearchedSegments + numProtectedSearchedSegments + numArchiveSearchedSegments);

if (earlybirdRequestContext.getRequest().getDebugMode() > 0) {

mergedResponse.setDebugString(

mergeClusterDebugStrings(realtimeResponse, protectedResponse, fullArchiveResponse));

}

return mergedResponse;

}

/\*\*

\* Merges the given responses.

\*

\* @param numResults the number of results requested

\* @param realtimeResponse the response from the realtime response

\* @param protectedResponse the response from the protected response

\* @param fullArchiveResponse the response from the full archive response

\* @return the list of merged results

\*/

private List<ThriftSearchResult> mergeResults(int numResults,

@Nullable EarlybirdResponse realtimeResponse,

@Nullable EarlybirdResponse protectedResponse,

@Nullable EarlybirdResponse fullArchiveResponse) {

mergedResponses.increment();

// We first merge the results from the two realtime clusters, Realtime cluster and

// Realtime Protected Tweets cluster

List<ThriftSearchResult> mergedResults = mergePublicAndProtectedRealtimeResults(

numResults,

realtimeResponse,

protectedResponse,

fullArchiveResponse,

clock);

EarlybirdResponseMergeUtil.addResultsToList(mergedResults, fullArchiveResponse,

ThriftTweetSource.FULL\_ARCHIVE\_CLUSTER);

List<ThriftSearchResult> distinctMergedResults =

EarlybirdResponseMergeUtil.distinctByStatusId(mergedResults, dupsStats);

if (mergedResults != distinctMergedResults) {

mergedResponsesWithExactDups.increment();

}

if (rankingMode == ThriftSearchRankingMode.RELEVANCE

|| rankingMode == ThriftSearchRankingMode.TOPTWEETS) {

distinctMergedResults.sort(ResultComparators.SCORE\_COMPARATOR);

} else {

distinctMergedResults.sort(ResultComparators.ID\_COMPARATOR);

}

return distinctMergedResults;

}

/\*\*

\* Method for merging tweets from protected and realtime clusters

\* - realtime, guaranteed newer than any archive tweets

\* - protected, also realtime, but with a potentially larger window (optional)

\* - archive, public, guaranteed older than any public realtime tweets (optional, used for

\* id limits, \*not added to results\*)

\* It adds the ThriftSearchResults from protected tweets to the realtimeResponse

\*

\* Algorithm diagram: (with newer tweets at the top)

\* ------------------------------------ <--- protected maxSearchedStatusID

\* |C:Newest protected realtime tweets|

\* | (does not exist if realtime |

\* | maxID >= protected maxID) |

\*

\* | ------------------------ | <--- 60 seconds ago

\* |D:Newer protected realtime tweets |

\* | (does not exist if realtime |

\* | maxID >= 60 seconds ago) |

\* ---------- | ------------------------ | <--- public realtime maxSearchedStatusID

\* |A:Public| |E:Automatically valid protected |

\* |realtime| |realtime tweets |

\* ---------- | ------------------------ | <--- public realtime minSearchedStatusID

\* | |

\* ---------- | E if archive is present | <--- public archive maxSearchedStatusID

\* ---------- | E if archive is present | <--- public archive maxSearchedStatusID

\* |B:Public| | F is archive is not present |

\* |archive | | |

\* ---------- | ------------------------ | <--- public archive minSearchedStatusID

\* |F:Older protected realtime tweets |

\* | (does not exist if protected |

\* | minID >= public minID) |

\* ------------------------------------ <--- protected minSearchedStatusID

\* Step 1: Select tweets from groups A, and E. If this is enough, return them

\* Step 2: Select tweets from groups A, E, and F. If this is enough, return them

\* Step 3: Select tweets from groups A, D, E, and F and return them

\*

\* There are two primary tradeoffs, both of which favor public tweets:

\* (1) Benefit: While public indexing latency is < 60s, auto-updating never misses public tweets

\* Cost: Absence of public tweets may delay protected tweets from being searchable for 60s

\* (2) Benefit: No failure or delay from the protected cluster will affect realtime results

\* Cost: If the protected cluster indexes more slowly, auto-update may miss its tweets

\*

\* @param fullArchiveTweets - used solely for generating anchor points, not merged in.

\*/

@VisibleForTesting

static List<ThriftSearchResult> mergePublicAndProtectedRealtimeResults(

int numRequested,

EarlybirdResponse realtimeTweets,

EarlybirdResponse realtimeProtectedTweets,

@Nullable EarlybirdResponse fullArchiveTweets,

Clock clock) {

// See which results will actually be used

boolean isRealtimeUsable = EarlybirdResponseUtil.hasResults(realtimeTweets);

boolean isArchiveUsable = EarlybirdResponseUtil.hasResults(fullArchiveTweets);

boolean isProtectedUsable = EarlybirdResponseUtil.hasResults(realtimeProtectedTweets);

long minId = Long.MIN\_VALUE;

long maxId = Long.MAX\_VALUE;

if (isRealtimeUsable) {

// Determine the actual upper/lower bounds on the tweet id

if (realtimeTweets.getSearchResults().isSetMinSearchedStatusID()) {

minId = realtimeTweets.getSearchResults().getMinSearchedStatusID();

}

if (realtimeTweets.getSearchResults().isSetMaxSearchedStatusID()) {

maxId = realtimeTweets.getSearchResults().getMaxSearchedStatusID();

}

int justRight = realtimeTweets.getSearchResults().getResultsSize();

if (isArchiveUsable) {

justRight += fullArchiveTweets.getSearchResults().getResultsSize();

if (fullArchiveTweets.getSearchResults().isSetMinSearchedStatusID()) {

long fullArchiveMinId = fullArchiveTweets.getSearchResults().getMinSearchedStatusID();

if (fullArchiveMinId <= minId) {

minId = fullArchiveMinId;

} else {

FULL\_ARCHIVE\_MIN\_ID\_GREATER\_THAN\_REALTIME\_MIN\_ID.increment();

}

}

}

if (isProtectedUsable) {

for (ThriftSearchResult result : realtimeProtectedTweets.getSearchResults().getResults()) {

if (result.getId() >= minId && result.getId() <= maxId) {

justRight++;

}

}

}

if (justRight < numRequested) {

// Since this is only used as an upper bound, old (pre-2010) ids are still handled correctly

maxId = Math.max(

maxId,

SnowflakeIdParser.generateValidStatusId(

clock.nowMillis() - Amount.of(60, Time.SECONDS).as(Time.MILLISECONDS), 0));

}

}

List<ThriftSearchResult> mergedSearchResults = Lists.newArrayListWithCapacity(numRequested \* 2);

// Add valid tweets in order of priority: protected, then realtime

// Only add results that are within range (that check only matters for protected)

if (isProtectedUsable) {

EarlybirdResponseMergeUtil.markWithTweetSource(

realtimeProtectedTweets.getSearchResults().getResults(),

ThriftTweetSource.REALTIME\_PROTECTED\_CLUSTER);

for (ThriftSearchResult result : realtimeProtectedTweets.getSearchResults().getResults()) {

if (result.getId() <= maxId && result.getId() >= minId) {

mergedSearchResults.add(result);

}

}

}

if (isRealtimeUsable) {

EarlybirdResponseMergeUtil.addResultsToList(

mergedSearchResults, realtimeTweets, ThriftTweetSource.REALTIME\_CLUSTER);

}

// Set the minSearchedStatusID and maxSearchedStatusID on the protected response to the

// minId and maxId that were used to trim the protected results.

// This is needed in order to correctly set these IDs on the merged response.

ThriftSearchResults protectedResults =

EarlybirdResponseUtil.getResults(realtimeProtectedTweets);

if ((protectedResults != null)

&& protectedResults.isSetMinSearchedStatusID()

&& (protectedResults.getMinSearchedStatusID() < minId)) {

protectedResults.setMinSearchedStatusID(minId);

}

if ((protectedResults != null)

&& protectedResults.isSetMaxSearchedStatusID()

&& (protectedResults.getMaxSearchedStatusID() > maxId)) {

realtimeProtectedTweets.getSearchResults().setMaxSearchedStatusID(maxId);

}

return mergedSearchResults;

}

/\*\*

\* Merges the debug strings of the given cluster responses.

\*

\* @param realtimeResponse The response from the realtime cluster.

\* @param protectedResponse The response from the protected cluster.

\* @param fullArchiveResponse The response from the full archive cluster.

\* @return The merged debug string.

\*/

public static String mergeClusterDebugStrings(@Nullable EarlybirdResponse realtimeResponse,

@Nullable EarlybirdResponse protectedResponse,

@Nullable EarlybirdResponse fullArchiveResponse) {

StringBuilder sb = new StringBuilder();

if ((realtimeResponse != null) && realtimeResponse.isSetDebugString()) {

sb.append("Realtime response: ").append(realtimeResponse.getDebugString());

}

if ((protectedResponse != null) && protectedResponse.isSetDebugString()) {

if (sb.length() > 0) {

sb.append("\n");

}

sb.append("Protected response: ").append(protectedResponse.getDebugString());

}

if ((fullArchiveResponse != null) && fullArchiveResponse.isSetDebugString()) {

if (sb.length() > 0) {

sb.append("\n");

}

sb.append("Full archive response: ").append(fullArchiveResponse.getDebugString());

}

if (sb.length() == 0) {

return null;

}

return sb.toString();

}

/\*\*

\* Sets the minSearchedStatusID field on the merged response.

\*

\* @param mergedResponse The merged response.

\* @param fullArchiveResponse The full archive response.

\* @param resultsTrimmed Whether the merged response results were trimmed.

\*/

private void setMinSearchedStatusId(EarlybirdResponse mergedResponse,

EarlybirdResponse realtimeResponse,

EarlybirdResponse protectedResponse,

EarlybirdResponse fullArchiveResponse,

boolean resultsTrimmed) {

Preconditions.checkNotNull(mergedResponse.getSearchResults());

if (resultsTrimmed) {

// We got more results that we asked for and we trimmed them.

// Set minSearchedStatusID to the ID of the oldest result.

ThriftSearchResults searchResults = mergedResponse.getSearchResults();

if (searchResults.getResultsSize() > 0) {

List<ThriftSearchResult> results = searchResults.getResults();

long lastResultId = results.get(results.size() - 1).getId();

searchResults.setMinSearchedStatusID(lastResultId);

}

return;

}

// We did not get more results that we asked for. Get the min of the minSearchedStatusIDs of

// the merged responses.

List<Long> minIDs = Lists.newArrayList();

if (fullArchiveResponse != null

&& fullArchiveResponse.isSetSearchResults()

&& fullArchiveResponse.getSearchResults().isSetMinSearchedStatusID()) {

minIDs.add(fullArchiveResponse.getSearchResults().getMinSearchedStatusID());

if (mergedResponse.getSearchResults().isSetMinSearchedStatusID()

&& mergedResponse.getSearchResults().getMinSearchedStatusID()

< fullArchiveResponse.getSearchResults().getMinSearchedStatusID()) {

invalidMinStatusId.increment();

}

}

if (protectedResponse != null

&& !EarlybirdResponseUtil.hasResults(realtimeResponse)

&& EarlybirdResponseUtil.hasResults(protectedResponse)

&& protectedResponse.getSearchResults().isSetMinSearchedStatusID()) {

minIDs.add(protectedResponse.getSearchResults().getMinSearchedStatusID());

}

if (mergedResponse.getSearchResults().isSetMinSearchedStatusID()) {

minIDs.add(mergedResponse.getSearchResults().getMinSearchedStatusID());

}

if (!minIDs.isEmpty()) {

mergedResponse.getSearchResults().setMinSearchedStatusID(Collections.min(minIDs));

} else {

noMinIds.increment();

}

}

/\*\*

\* Sets the maxSearchedStatusID field on the merged response.

\*

\* @param mergedResponse The merged response.

\* @param fullArchiveResponse The full archive response.

\*/

private void setMaxSearchedStatusId(EarlybirdResponse mergedResponse,

EarlybirdResponse realtimeResponse,

EarlybirdResponse protectedResponse,

EarlybirdResponse fullArchiveResponse) {

Preconditions.checkNotNull(mergedResponse.getSearchResults());

List<Long> maxIDs = Lists.newArrayList();

if (fullArchiveResponse != null

&& fullArchiveResponse.isSetSearchResults()

&& fullArchiveResponse.getSearchResults().isSetMaxSearchedStatusID()) {

maxIDs.add(fullArchiveResponse.getSearchResults().getMaxSearchedStatusID());

if (mergedResponse.getSearchResults().isSetMaxSearchedStatusID()

&& fullArchiveResponse.getSearchResults().getMaxSearchedStatusID()

> mergedResponse.getSearchResults().getMaxSearchedStatusID()) {

invalidMaxStatusId.increment();

}

}

if (protectedResponse != null

&& !EarlybirdResponseUtil.hasResults(realtimeResponse)

&& EarlybirdResponseUtil.hasResults(protectedResponse)

&& protectedResponse.getSearchResults().isSetMaxSearchedStatusID()) {

maxIDs.add(protectedResponse.getSearchResults().getMaxSearchedStatusID());

}

if (mergedResponse.getSearchResults().isSetMaxSearchedStatusID()) {

maxIDs.add(mergedResponse.getSearchResults().getMaxSearchedStatusID());

}

ThriftSearchResults searchResults = mergedResponse.getSearchResults();

if (searchResults.getResultsSize() > 0) {

List<ThriftSearchResult> results = searchResults.getResults();

maxIDs.add(results.get(0).getId());

}

if (!maxIDs.isEmpty()) {

mergedResponse.getSearchResults().setMaxSearchedStatusID(Collections.max(maxIDs));

} else {

noMaxIds.increment();

}

}

/\*\*

\* Handles exceptions thrown while merging responses. Timeout exceptions are converted to

\* SERVER\_TIMEOUT\_ERROR responses. All other exceptions are converted to PERSISTENT\_ERROR

\* responses.

\*/

private Future<EarlybirdResponse> handleResponseException(

Future<EarlybirdResponse> responseFuture, final String debugMsg) {

return responseFuture.handle(

new Function<Throwable, EarlybirdResponse>() {

@Override

public EarlybirdResponse apply(Throwable t) {

EarlybirdResponseCode responseCode = EarlybirdResponseCode.PERSISTENT\_ERROR;

if (FinagleUtil.isTimeoutException(t)) {

responseCode = EarlybirdResponseCode.SERVER\_TIMEOUT\_ERROR;

}

EarlybirdResponse response = new EarlybirdResponse(responseCode, 0);

response.setDebugString(debugMsg + "\n" + t);

return response;

}

});

}

}