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

import java.util.ArrayList;

import java.util.List;

import javax.inject.Inject;

import javax.inject.Named;

import com.google.common.base.Preconditions;

import com.google.common.collect.ImmutableList;

import com.google.common.collect.Lists;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.finagle.Service;

import com.twitter.search.common.decider.SearchDecider;

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

import com.twitter.search.earlybird.config.ServingRange;

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

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

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

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

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

import com.twitter.search.earlybird\_root.filters.EarlybirdTimeRangeFilter;

import com.twitter.search.earlybird\_root.filters.ServingRangeProvider;

import com.twitter.search.earlybird\_root.mergers.EarlybirdResponseMerger;

import com.twitter.search.earlybird\_root.mergers.SuperRootResponseMerger;

import com.twitter.search.earlybird\_root.mergers.TermStatisticsResponseMerger;

import com.twitter.search.earlybird\_root.mergers.TierResponseAccumulator;

import com.twitter.util.Function;

import com.twitter.util.Future;

import static com.twitter.search.common.util.earlybird.TermStatisticsUtil.determineBinSize;

/\*\*

\* For TermStats traffic SuperRoot hits both realtime and archive in parallel, and then merges

\* the results.

\*/

public class TermStatsRequestRouter extends RequestRouter {

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

private static final String SUPERROOT\_SKIP\_FULL\_ARCHIVE\_CLUSTER\_FOR\_TERM\_STATS\_REQUESTS =

"superroot\_skip\_full\_archive\_cluster\_for\_term\_stats\_requests";

private final Service<EarlybirdRequestContext, EarlybirdResponse> realtimeService;

private final Service<EarlybirdRequestContext, EarlybirdResponse> fullArchiveService;

private final SearchDecider decider;

private final ServingRangeProvider realtimeServingRangeProvider;

@Inject

public TermStatsRequestRouter(

@Named(InjectionNames.REALTIME)

Service<EarlybirdRequestContext, EarlybirdResponse> realtime,

@Named(TermStatsRequestRouterModule.REALTIME\_TIME\_RANGE\_FILTER)

EarlybirdTimeRangeFilter realtimeTimeRangeFilter,

@Named(InjectionNames.FULL\_ARCHIVE)

Service<EarlybirdRequestContext, EarlybirdResponse> fullArchive,

@Named(TermStatsRequestRouterModule.FULL\_ARCHIVE\_TIME\_RANGE\_FILTER)

EarlybirdTimeRangeFilter fullArchiveTimeRangeFilter,

SearchDecider decider) {

LOG.info("Instantiating a TermStatsRequestRouter");

this.realtimeService = realtimeTimeRangeFilter

.andThen(realtime);

this.fullArchiveService = fullArchiveTimeRangeFilter

.andThen(fullArchive);

this.decider = decider;

this.realtimeServingRangeProvider = realtimeTimeRangeFilter.getServingRangeProvider();

}

/\*\*

\* Hit both realtime and full-archive clusters then merges term stat request.

\*/

@Override

public Future<EarlybirdResponse> route(EarlybirdRequestContext requestContext) {

List<RequestResponse> requestResponses = new ArrayList<>();

Future<EarlybirdResponse> realtimeResponseFuture = realtimeService.apply(requestContext);

this.saveRequestResponse(requestResponses, "realtime", requestContext, realtimeResponseFuture);

Future<EarlybirdResponse> archiveResponseFuture =

requestContext.getRequest().isGetOlderResults()

&& !decider.isAvailable(SUPERROOT\_SKIP\_FULL\_ARCHIVE\_CLUSTER\_FOR\_TERM\_STATS\_REQUESTS)

? fullArchiveService.apply(requestContext)

: Future.value(emptyResponse());

this.saveRequestResponse(requestResponses, "archive", requestContext, archiveResponseFuture);

Future<EarlybirdResponse> mergedResponse =

merge(realtimeResponseFuture, archiveResponseFuture, requestContext);

return this.maybeAttachSentRequestsToDebugInfo(

requestResponses,

requestContext,

mergedResponse

);

}

/\*\*

\* Merge responses from realtime and full archive clusters.

\*/

private Future<EarlybirdResponse> merge(

final Future<EarlybirdResponse> realtimeResponseFuture,

final Future<EarlybirdResponse> archiveResponseFuture,

final EarlybirdRequestContext requestContext) {

return realtimeResponseFuture.flatMap(

new Function<EarlybirdResponse, Future<EarlybirdResponse>>() {

@Override

public Future<EarlybirdResponse> apply(final EarlybirdResponse realtimeResponse) {

if (!EarlybirdResponseUtil.isSuccessfulResponse(realtimeResponse)) {

return Future.value(realtimeResponse);

}

return archiveResponseFuture.flatMap(

new Function<EarlybirdResponse, Future<EarlybirdResponse>>() {

@Override

public Future<EarlybirdResponse> apply(EarlybirdResponse archiveResponse) {

if (!EarlybirdResponseUtil.isSuccessfulResponse(archiveResponse)) {

return Future.value(

mergeWithUnsuccessfulArchiveResponse(

requestContext, realtimeResponse, archiveResponse));

}

List<Future<EarlybirdResponse>> responses =

ImmutableList.<Future<EarlybirdResponse>>builder()

.add(realtimeResponseFuture)

.add(archiveResponseFuture)

.build();

EarlybirdResponseMerger merger = new TermStatisticsResponseMerger(

requestContext, responses, new TierResponseAccumulator());

return merger.merge().map(new Function<EarlybirdResponse, EarlybirdResponse>() {

@Override

public EarlybirdResponse apply(EarlybirdResponse mergedResponse) {

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

mergedResponse.setDebugString(

SuperRootResponseMerger.mergeClusterDebugStrings(

realtimeResponse, null, archiveResponse));

}

return mergedResponse;

}

});

}

});

}

});

}

private EarlybirdResponse mergeWithUnsuccessfulArchiveResponse(

EarlybirdRequestContext requestContext,

EarlybirdResponse realtimeResponse,

EarlybirdResponse archiveResponse) {

// If the realtime cluster was skipped, and the full archive returned an error

// response, return the full archive response.

if (isTierSkippedResponse(realtimeResponse)) {

return archiveResponse;

}

// If the realtime response has results and the full archive cluster returned an error

// response, we return the realtime response. If the client needs more results, it can paginate,

// and on the next request it will get the error response from the full archive cluster.

if (realtimeResponse.isSetTermStatisticsResults()

&& !realtimeResponse.getTermStatisticsResults().getTermResults().isEmpty()) {

realtimeResponse.setDebugString(

"Full archive cluster returned an error response ("

+ archiveResponse.getResponseCode() + "). "

+ SuperRootResponseMerger.mergeClusterDebugStrings(

realtimeResponse, null, archiveResponse));

return updateMinCompleteBinId(requestContext, realtimeResponse);

}

// If the realtime response has no results, and the full archive cluster returned an error

// response, return a PERSISTENT\_ERROR response, and merge the debug strings from the two

// responses.

EarlybirdResponse mergedResponse =

new EarlybirdResponse(EarlybirdResponseCode.PERSISTENT\_ERROR, 0);

mergedResponse.setDebugString(

"Full archive cluster returned an error response ("

+ archiveResponse.getResponseCode()

+ "), and the realtime response had no results. "

+ SuperRootResponseMerger.mergeClusterDebugStrings(

realtimeResponse, null, archiveResponse));

return mergedResponse;

}

/\*\*

\* If we get a completed realtime response but a failed archive response, the minCompleteBinId we

\* return will be incorrect -- the realtime minCompleteBinId is assumed to be the oldest bin

\* returned, rather than the bin that intersects the realtime serving boundary. In these cases, we

\* need to move the minCompleteBinId forward.

\* <p>

\* Note that we cannot always set the minCompleteBinId for the realtime results to the bin

\* intersecting the realtime serving boundary: somewhere in the guts of the merging logic, we set

\* the minCompleteBinId of the merged response to the max of the minCompleteBinIds of the original

\* responses. :-(

\*/

private EarlybirdResponse updateMinCompleteBinId(

EarlybirdRequestContext requestContext, EarlybirdResponse realtimeResponse) {

Preconditions.checkArgument(

realtimeResponse.getTermStatisticsResults().isSetMinCompleteBinId());

int roundedServingRange = roundServingRangeUpToNearestBinId(requestContext, realtimeResponse);

int minCompleteBinId = Math.max(

roundedServingRange,

realtimeResponse.getTermStatisticsResults().getMinCompleteBinId());

realtimeResponse.getTermStatisticsResults().setMinCompleteBinId(minCompleteBinId);

return realtimeResponse;

}

private static EarlybirdResponse emptyResponse() {

return new EarlybirdResponse(EarlybirdResponseCode.SUCCESS, 0)

.setSearchResults(new ThriftSearchResults()

.setResults(Lists.newArrayList()))

.setDebugString("Full archive cluster not requested or not available.");

}

private static boolean isTierSkippedResponse(EarlybirdResponse response) {

return response.getResponseCode() == EarlybirdResponseCode.TIER\_SKIPPED;

}

/\*\*

\* Given a termstats request/response pair, round the serving range for the appropriate cluster up

\* to the nearest binId at the appropriate resolution.

\*/

private int roundServingRangeUpToNearestBinId(

EarlybirdRequestContext request, EarlybirdResponse response) {

ServingRange servingRange = realtimeServingRangeProvider.getServingRange(

request, request.useOverrideTierConfig());

long servingRangeStartSecs = servingRange.getServingRangeSinceTimeSecondsFromEpoch();

int binSize = determineBinSize(response.getTermStatisticsResults().getHistogramSettings());

return (int) Math.ceil((double) servingRangeStartSecs / binSize);

}

}