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

import java.util.Collections;

import java.util.List;

import java.util.concurrent.TimeUnit;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Preconditions;

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

import com.twitter.search.common.schema.earlybird.EarlybirdCluster;

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

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

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

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

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

import com.twitter.util.Future;

/\*\*

\* A RecencyResponseMerger that prioritizes not losing results during pagination.

\* As of now, this merger is used by Gnip to make sure that scrolling returns all results.

\*

\* The logic used for merging partitions is a bit tricky, because on one hand, we want to make sure

\* that we do miss results on the next pagination request; on the other hand, we want to return as

\* many results as we can, and we want to set the minSearchedStatusID of the merged response as low

\* as we can, in order to minimize the number of pagination requests.

\*

\* The merging logic is:

\*

\* Realtime cluster:

\* 1. merge results from all partitions

\* 2. if at least one partition response is early-terminated, set earlyTerminated = true

\* on the merged response

\* 3. set trimmingMinId = max(minSearchedStatusIDs of all partition responses)

\* 4. trim all results to trimmingMinId

\* 5. set minSearchedStatusID on the merged response to trimmingMinId

\* 6. if we have more than numRequested results:

\* - keep only the newest numRequested results

\* - set minSearchedStatusID of the merged response to the lowest tweet ID in the response

\* 7. if at least one partition response is not early-terminated, set

\* tierBottomId = max(minSearchedStatusIDs of all non-early-terminated responses)

\* (otherwise, set tierBottomId to some undefined value: -1, Long.MAX\_VALUE, etc.)

\* 8. if minSearchedStatusID of the merged response is the same as tierBottomId,

\* clear the early-termination flag on the merged response

\*

\* The logic in steps 7 and 8 can be a little tricky to understand. They basically say: when we've

\* exhausted the "least deep" partition in the realtime cluster, it's time to move to the full

\* archive cluster (if we keep going past the "least deep" partition, we might miss results).

\*

\* Full archive cluster:

\* 1. merge results from all partitions

\* 2. if at least one partition response is early-terminated, set earlyTerminated = true

\* on the merged response

\* 3. set trimmingMinId to:

\* - max(minSearchedStatusIDs of early-terminated responses), if at least one partition response

\* is early-terminated

\* - min(minSearchedStatusIDs of all responses), if all partition responses are not

\* early-terminated

\* 4. trim all results to trimmingMinId

\* 5. set minSearchedStatusID of the merged response to trimmingMinId

\* 6. if we have more than numRequested results:

\* - keep only the newest numRequested results

\* - set minSearchedStatusID of the merged response to the lowest tweet ID in the response

\*

\* The logic in step 3 can be a little tricky to understand. On one hand, if we always set

\* trimmingMinId to the highest minSearchedStatusID, then some tweets at the very bottom of some

\* partitions will never be returned. Consider the case:

\*

\* partition 1 has tweets 10, 8, 6

\* partition 2 has tweets 9, 7, 5

\*

\* In this case, we would always trim all results to minId = 6, and tweet 5 would never be returned.

\*

\* On the other hand, if we always set trimmingMinId to the lowest minSearchedStatusID, then we

\* might miss tweets from partitions that early-terminated. Consider the case:

\*

\* partition 1 has tweets 10, 5, 3, 1 that match our query

\* partition 2 has tweets 9, 8, 7, 6, 2 that match our query

\*

\* If we ask for 3 results, than partition 1 will return tweets 10, 5, 3, and partition 2 will

\* return tweets 9, 8, 7. If we set trimmingMinId = min(minSearchedStatusIDs), then the next

\* pagination request will have [max\_id = 2], and we will miss tweet 6.

\*

\* So the intuition here is that if we have an early-terminated response, we cannot set

\* trimmingMinId to something lower than the minSearchedStatusID returned by that partition

\* (otherwise we might miss results from that partition). However, if we've exhausted all

\* partitions, then it's OK to not trim any result, because tiers do not intersect, so we will not

\* miss any result from the next tier once we get there.

\*/

public class StrictRecencyResponseMerger extends RecencyResponseMerger {

private static final SearchTimerStats STRICT\_RECENCY\_TIMER\_AVG =

SearchTimerStats.export("merge\_recency\_strict", TimeUnit.NANOSECONDS, false, true);

@VisibleForTesting

static final EarlyTerminationTrimmingStats PARTITION\_MERGING\_EARLY\_TERMINATION\_TRIMMING\_STATS =

new EarlyTerminationTrimmingStats("strict\_recency\_partition\_merging");

@VisibleForTesting

static final EarlyTerminationTrimmingStats TIER\_MERGING\_EARLY\_TERMINATION\_TRIMMING\_STATS =

new EarlyTerminationTrimmingStats("strict\_recency\_tier\_merging");

private final EarlybirdCluster cluster;

public StrictRecencyResponseMerger(EarlybirdRequestContext requestContext,

List<Future<EarlybirdResponse>> responses,

ResponseAccumulator mode,

EarlybirdFeatureSchemaMerger featureSchemaMerger,

EarlybirdCluster cluster) {

super(requestContext, responses, mode, featureSchemaMerger);

this.cluster = cluster;

}

@Override

protected SearchTimerStats getMergedResponseTimer() {

return STRICT\_RECENCY\_TIMER\_AVG;

}

/\*\*

\* Unlike {@link com.twitter.search.earlybird\_root.mergers.RecencyResponseMerger}, this method

\* takes a much simpler approach by just taking the max of the maxSearchedStatusIds.

\*

\* Also, when no maxSearchedStatusId is available at all, Long.MIN\_VALUE is used instead of

\* Long.MAX\_VALUE. This ensures that we don't return any result in these cases.

\*/

@Override

protected long findMaxFullySearchedStatusID() {

return accumulatedResponses.getMaxIds().isEmpty()

? Long.MIN\_VALUE : Collections.max(accumulatedResponses.getMaxIds());

}

/\*\*

\* This method is subtly different from the base class version: when no minSearchedStatusId is

\* available at all, Long.MAX\_VALUE is used instead of Long.MIN\_VALUE. This ensures that we

\* don't return any result in these cases.

\*/

@Override

protected long findMinFullySearchedStatusID() {

List<Long> minIds = accumulatedResponses.getMinIds();

if (minIds.isEmpty()) {

return Long.MAX\_VALUE;

}

if (accumulatedResponses.isMergingPartitionsWithinATier()) {

return getTrimmingMinId();

}

// When merging tiers, the min ID should be the smallest among the min IDs.

return Collections.min(minIds);

}

@Override

protected TrimStats trimResults(

ThriftSearchResults searchResults, long mergedMin, long mergedMax) {

if (!searchResults.isSetResults() || searchResults.getResultsSize() == 0) {

// no results, no trimming needed

return TrimStats.EMPTY\_STATS;

}

TrimStats trimStats = new TrimStats();

trimExactDups(searchResults, trimStats);

filterResultsByMergedMinMaxIds(searchResults, mergedMax, mergedMin, trimStats);

int numResults = computeNumResultsToKeep();

if (searchResults.getResultsSize() > numResults) {

trimStats.setResultsTruncatedFromTailCount(searchResults.getResultsSize() - numResults);

searchResults.setResults(searchResults.getResults().subList(0, numResults));

}

return trimStats;

}

/\*\*

\* This method is different from the base class version because when minResultId is bigger

\* than currentMergedMin, we always take minResultId.

\* If we don't do this, we would lose results.

\*

\* Illustration with an example. Assuming we are outside of the lag threshold.

\* Num results requested: 3

\* Response 1: min: 100 max: 900 results: 400, 500, 600

\* Response 2: min: 300 max: 700 results: 350, 450, 550

\*

\* Merged results: 600, 550, 500

\* Merged max: 900

\* Merged min: we could take 300 (minId), or take 500 (minResultId).

\*

\* If we take minId, and use 300 as the pagination cursor, we'd lose results

\* 350 and 450 when we paginate. So we have to take minResultId here.

\*/

@Override

protected void setMergedMinSearchedStatusId(

ThriftSearchResults searchResults,

long currentMergedMin,

boolean resultsWereTrimmed) {

if (accumulatedResponses.getMinIds().isEmpty()) {

return;

}

long minId = currentMergedMin;

if (resultsWereTrimmed

&& (searchResults != null)

&& searchResults.isSetResults()

&& (searchResults.getResultsSize() > 0)) {

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

minId = results.get(results.size() - 1).getId();

}

searchResults.setMinSearchedStatusID(minId);

}

@Override

protected boolean clearEarlyTerminationIfReachingTierBottom(EarlybirdResponse mergedResponse) {

if (EarlybirdCluster.isArchive(cluster)) {

// We don't need to worry about the tier bottom when merging partition responses in the full

// archive cluster: if all partitions were exhausted and we didn't trim the results, then

// the early-terminated flag on the merged response will be false. If at least one partition

// is early-terminated, or we trimmed some results, then the ealry-terminated flag on the

// merged response will be true, and we should continue getting results from this tier before

// we move to the next one.

return false;

}

ThriftSearchResults searchResults = mergedResponse.getSearchResults();

if (searchResults.getMinSearchedStatusID() == getTierBottomId()) {

mergedResponse.getEarlyTerminationInfo().setEarlyTerminated(false);

mergedResponse.getEarlyTerminationInfo().unsetMergedEarlyTerminationReasons();

responseMessageBuilder.debugVerbose(

"Set earlytermination to false because minSearchedStatusId is tier bottom");

return true;

}

return false;

}

@Override

protected boolean shouldEarlyTerminateWhenEnoughTrimmedResults() {

return false;

}

@Override

protected final EarlyTerminationTrimmingStats getEarlyTerminationTrimmingStatsForPartitions() {

return PARTITION\_MERGING\_EARLY\_TERMINATION\_TRIMMING\_STATS;

}

@Override

protected final EarlyTerminationTrimmingStats getEarlyTerminationTrimmingStatsForTiers() {

return TIER\_MERGING\_EARLY\_TERMINATION\_TRIMMING\_STATS;

}

/\*\* Determines the bottom of the realtime cluster, based on the partition responses. \*/

private long getTierBottomId() {

Preconditions.checkState(!EarlybirdCluster.isArchive(cluster));

long tierBottomId = -1;

for (EarlybirdResponse response : accumulatedResponses.getSuccessResponses()) {

if (!isEarlyTerminated(response)

&& response.isSetSearchResults()

&& response.getSearchResults().isSetMinSearchedStatusID()

&& (response.getSearchResults().getMinSearchedStatusID() > tierBottomId)) {

tierBottomId = response.getSearchResults().getMinSearchedStatusID();

}

}

return tierBottomId;

}

/\*\* Determines the minId to which all results should be trimmed. \*/

private long getTrimmingMinId() {

List<Long> minIds = accumulatedResponses.getMinIds();

Preconditions.checkArgument(!minIds.isEmpty());

if (!EarlybirdCluster.isArchive(cluster)) {

return Collections.max(minIds);

}

long maxOfEarlyTerminatedMins = -1;

long minOfAllMins = Long.MAX\_VALUE;

for (EarlybirdResponse response : accumulatedResponses.getSuccessResponses()) {

if (response.isSetSearchResults()

&& response.getSearchResults().isSetMinSearchedStatusID()) {

long minId = response.getSearchResults().getMinSearchedStatusID();

minOfAllMins = Math.min(minOfAllMins, minId);

if (isEarlyTerminated(response)) {

maxOfEarlyTerminatedMins = Math.max(maxOfEarlyTerminatedMins, minId);

}

}

}

if (maxOfEarlyTerminatedMins >= 0) {

return maxOfEarlyTerminatedMins;

} else {

return minOfAllMins;

}

}

/\*\* Determines if the given earlybird response is early terminated. \*/

private boolean isEarlyTerminated(EarlybirdResponse response) {

return response.isSetEarlyTerminationInfo()

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

}

}