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

import java.util.Collections;

import java.util.List;

import java.util.Map;

import java.util.concurrent.TimeUnit;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Preconditions;

import com.google.common.collect.ImmutableMap;

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.metrics.SearchTimerStats;

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.earlybird.thrift.EarlybirdResponse;

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

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

import com.twitter.search.earlybird\_root.collectors.RecencyMergeCollector;

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

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

import com.twitter.util.Future;

import static com.twitter.search.earlybird\_root.mergers.RecencyResponseMerger

.EarlyTerminationTrimmingStats.Type.ALREADY\_EARLY\_TERMINATED;

import static com.twitter.search.earlybird\_root.mergers.RecencyResponseMerger

.EarlyTerminationTrimmingStats.Type.FILTERED;

import static com.twitter.search.earlybird\_root.mergers.RecencyResponseMerger

.EarlyTerminationTrimmingStats.Type.FILTERED\_AND\_TRUNCATED;

import static com.twitter.search.earlybird\_root.mergers.RecencyResponseMerger

.EarlyTerminationTrimmingStats.Type.NOT\_EARLY\_TERMINATED;

import static com.twitter.search.earlybird\_root.mergers.RecencyResponseMerger

.EarlyTerminationTrimmingStats.Type.TERMINATED\_GOT\_EXACT\_NUM\_RESULTS;

import static com.twitter.search.earlybird\_root.mergers.RecencyResponseMerger

.EarlyTerminationTrimmingStats.Type.TRUNCATED;

/\*\*

\* Merger class to merge recency search EarlybirdResponse objects.

\*/

public class RecencyResponseMerger extends EarlybirdResponseMerger {

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

private static final SearchTimerStats RECENCY\_TIMER =

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

@VisibleForTesting

static final String TERMINATED\_COLLECTED\_ENOUGH\_RESULTS =

"terminated\_collected\_enough\_results";

// Allowed replication lag relative to all replicas. Replication lag exceeding

// this amount may result in some tweets from the replica not returned in search.

private static final long ALLOWED\_REPLICATION\_LAG\_MS = 10000;

private static final double SUCCESSFUL\_RESPONSE\_THRESHOLD = 0.9;

@VisibleForTesting

static final SearchCounter RECENCY\_ZERO\_RESULT\_COUNT\_AFTER\_FILTERING\_MAX\_MIN\_IDS =

SearchCounter.export("merger\_recency\_zero\_result\_count\_after\_filtering\_max\_min\_ids");

@VisibleForTesting

static final SearchCounter RECENCY\_TRIMMED\_TOO\_MANY\_RESULTS\_COUNT =

SearchCounter.export("merger\_recency\_trimmed\_too\_many\_results\_count");

private static final SearchCounter RECENCY\_TIER\_MERGE\_EARLY\_TERMINATED\_WITH\_NOT\_ENOUGH\_RESULTS =

SearchCounter.export("merger\_recency\_tier\_merge\_early\_terminated\_with\_not\_enough\_results");

private static final SearchCounter RECENCY\_CLEARED\_EARLY\_TERMINATION\_COUNT =

SearchCounter.export("merger\_recency\_cleared\_early\_termination\_count");

/\*\*

\* Results were truncated because merged results exceeded the requested numResults.

\*/

@VisibleForTesting

static final String MERGING\_EARLY\_TERMINATION\_REASON\_TRUNCATED =

"root\_merging\_truncated\_results";

/\*\*

\* Results that were were filtered smaller than merged minSearchedStatusId were filtered out.

\*/

@VisibleForTesting

static final String MERGING\_EARLY\_TERMINATION\_REASON\_FILTERED =

"root\_merging\_filtered\_results";

@VisibleForTesting

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

new EarlyTerminationTrimmingStats("recency\_partition\_merging");

@VisibleForTesting

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

new EarlyTerminationTrimmingStats("recency\_tier\_merging");

@VisibleForTesting

static class EarlyTerminationTrimmingStats {

enum Type {

/\*\*

\* The whole result was not terminated at all.

\*/

NOT\_EARLY\_TERMINATED,

/\*\*

\* Was terminated before we did any trimming.

\*/

ALREADY\_EARLY\_TERMINATED,

/\*\*

\* Was not terminated when merged, but results were filtered due to min/max ranges.

\*/

FILTERED,

/\*\*

\* Was not terminated when merged, but results were truncated.

\*/

TRUNCATED,

/\*\*

\* Was not terminated when merged, but results were filtered due to min/max ranges and

\* truncated.

\*/

FILTERED\_AND\_TRUNCATED,

/\*\*

\* When the search asks for X result, and we get exactly X results back, without trimming

\* or truncating on the tail side (min\_id side), we still mark the search as early terminated.

\* This is because later tiers possibly has more results.

\*/

TERMINATED\_GOT\_EXACT\_NUM\_RESULTS,

}

/\*\*

\* A counter tracking merged responses for each {@link EarlyTerminationTrimmingStats.Type}

\* define above.

\*/

private final ImmutableMap<Type, SearchCounter> searchCounterMap;

EarlyTerminationTrimmingStats(String prefix) {

Map<Type, SearchCounter> tempMap = Maps.newEnumMap(Type.class);

tempMap.put(NOT\_EARLY\_TERMINATED,

SearchCounter.export(prefix + "\_not\_early\_terminated\_after\_merging"));

tempMap.put(ALREADY\_EARLY\_TERMINATED,

SearchCounter.export(prefix + "\_early\_terminated\_before\_merge\_trimming"));

tempMap.put(TRUNCATED,

SearchCounter.export(prefix + "\_early\_terminated\_after\_merging\_truncated"));

tempMap.put(FILTERED,

SearchCounter.export(prefix + "\_early\_terminated\_after\_merging\_filtered"));

tempMap.put(FILTERED\_AND\_TRUNCATED,

SearchCounter.export(prefix + "\_early\_terminated\_after\_merging\_filtered\_and\_truncated"));

tempMap.put(TERMINATED\_GOT\_EXACT\_NUM\_RESULTS,

SearchCounter.export(prefix + "\_early\_terminated\_after\_merging\_got\_exact\_num\_results"));

searchCounterMap = Maps.immutableEnumMap(tempMap);

}

public SearchCounter getCounterFor(Type type) {

return searchCounterMap.get(type);

}

}

private final EarlybirdFeatureSchemaMerger featureSchemaMerger;

public RecencyResponseMerger(EarlybirdRequestContext requestContext,

List<Future<EarlybirdResponse>> responses,

ResponseAccumulator mode,

EarlybirdFeatureSchemaMerger featureSchemaMerger) {

super(requestContext, responses, mode);

this.featureSchemaMerger = featureSchemaMerger;

}

@Override

protected double getDefaultSuccessResponseThreshold() {

return SUCCESSFUL\_RESPONSE\_THRESHOLD;

}

@Override

protected SearchTimerStats getMergedResponseTimer() {

return RECENCY\_TIMER;

}

@Override

protected EarlybirdResponse internalMerge(EarlybirdResponse mergedResponse) {

// The merged maxSearchedStatusId and minSearchedStatusId

long maxId = findMaxFullySearchedStatusID();

long minId = findMinFullySearchedStatusID();

RecencyMergeCollector collector = new RecencyMergeCollector(responses.size());

int totalResultSize = addResponsesToCollector(collector);

ThriftSearchResults searchResults = collector.getAllSearchResults();

TrimStats trimStats = trimResults(searchResults, minId, maxId);

setMergedMaxSearchedStatusId(searchResults, maxId);

setMergedMinSearchedStatusId(

searchResults, minId, trimStats.getResultsTruncatedFromTailCount() > 0);

mergedResponse.setSearchResults(searchResults);

// Override some components of the response as appropriate to real-time.

searchResults.setHitCounts(aggregateHitCountMap());

if (accumulatedResponses.isMergingPartitionsWithinATier()

&& clearEarlyTerminationIfReachingTierBottom(mergedResponse)) {

RECENCY\_CLEARED\_EARLY\_TERMINATION\_COUNT.increment();

} else {

setEarlyTerminationForTrimmedResults(mergedResponse, trimStats);

}

responseMessageBuilder.debugVerbose("Hits: %s %s", totalResultSize, trimStats);

responseMessageBuilder.debugVerbose(

"Hash Partitioned Earlybird call completed successfully: %s", mergedResponse);

featureSchemaMerger.collectAndSetFeatureSchemaInResponse(

searchResults,

requestContext,

"merger\_recency\_tier",

accumulatedResponses.getSuccessResponses());

return mergedResponse;

}

/\*\*

\* When we reached tier bottom, pagination can stop working even though we haven't got

\* all results. e.g.

\* Results from partition 1: [101 91 81], minSearchedStatusId is 81

\* Results from Partition 2: [102 92], minSearchedStatusId is 92, not early terminated.

\*

\* After merge, we get [102, 101, 92], with minResultId == 92. Since results from

\* partition 2 is not early terminated, 92 is the tier bottom here. Since results are

\* filtered, early termination for merged result is set to true, so blender will call again,

\* with maxDocId == 91. This time we get result:

\* Results from partition 1: [91 81], minSearchedStatusId is 81

\* Results from partition 2: [], minSearchedStatusId is still 92

\* After merge we get [] and minSearchedStatusId is still 92. No progress can be made on

\* pagination and clients get stuck.

\*

\* So in this case, we clear the early termination flag to tell blender there is no more

\* result in this tier. Tweets below tier bottom will be missed, but that also happens

\* without this step, as the next pagination call will return empty results anyway.

\* So even if there is NOT overlap between tiers, this is still better.

\*

\* Return true if early termination is cleared due to this, otherwise return false.

\* To be safe, we do nothing here to keep existing behavior and only override it in

\* StrictRecencyResponseMerger.

\*/

protected boolean clearEarlyTerminationIfReachingTierBottom(EarlybirdResponse mergedResponse) {

return false;

}

/\*\*

\* Determines if the merged response should be early-terminated when it has exactly as many

\* trimmed results as requested, as is not early-terminated because of other reasons.

\*/

protected boolean shouldEarlyTerminateWhenEnoughTrimmedResults() {

return true;

}

/\*\*

\* If the end results were trimmed in any way, reflect that in the response as a query that was

\* early terminated. A response can be either (1) truncated because we merged more results than

\* what was asked for with numResults, or (2) we filtered results that were smaller than the

\* merged minSearchedStatusId.

\*

\* @param mergedResponse the merged response.

\* @param trimStats trim stats for this merge.

\*/

private void setEarlyTerminationForTrimmedResults(

EarlybirdResponse mergedResponse,

TrimStats trimStats) {

responseMessageBuilder.debugVerbose("Checking for merge trimming, trimStats %s", trimStats);

EarlyTerminationTrimmingStats stats = getEarlyTerminationTrimmingStats();

EarlyTerminationInfo earlyTerminationInfo = mergedResponse.getEarlyTerminationInfo();

Preconditions.checkNotNull(earlyTerminationInfo);

if (!earlyTerminationInfo.isEarlyTerminated()) {

if (trimStats.getMinIdFilterCount() > 0 || trimStats.getResultsTruncatedFromTailCount() > 0) {

responseMessageBuilder.debugVerbose("Setting early termination, trimStats: %s, results: %s",

trimStats, mergedResponse);

earlyTerminationInfo.setEarlyTerminated(true);

addEarlyTerminationReasons(earlyTerminationInfo, trimStats);

if (trimStats.getMinIdFilterCount() > 0

&& trimStats.getResultsTruncatedFromTailCount() > 0) {

stats.getCounterFor(FILTERED\_AND\_TRUNCATED).increment();

} else if (trimStats.getMinIdFilterCount() > 0) {

stats.getCounterFor(FILTERED).increment();

} else if (trimStats.getResultsTruncatedFromTailCount() > 0) {

stats.getCounterFor(TRUNCATED).increment();

} else {

Preconditions.checkState(false, "Invalid TrimStats: %s", trimStats);

}

} else if ((computeNumResultsToKeep() == mergedResponse.getSearchResults().getResultsSize())

&& shouldEarlyTerminateWhenEnoughTrimmedResults()) {

earlyTerminationInfo.setEarlyTerminated(true);

earlyTerminationInfo.addToMergedEarlyTerminationReasons(

TERMINATED\_COLLECTED\_ENOUGH\_RESULTS);

stats.getCounterFor(TERMINATED\_GOT\_EXACT\_NUM\_RESULTS).increment();

} else {

stats.getCounterFor(NOT\_EARLY\_TERMINATED).increment();

}

} else {

stats.getCounterFor(ALREADY\_EARLY\_TERMINATED).increment();

// Even if the results were already marked as early terminated, we can add additional

// reasons for debugging (if the merged results were filtered or truncated).

addEarlyTerminationReasons(earlyTerminationInfo, trimStats);

}

}

private void addEarlyTerminationReasons(

EarlyTerminationInfo earlyTerminationInfo,

TrimStats trimStats) {

if (trimStats.getMinIdFilterCount() > 0) {

earlyTerminationInfo.addToMergedEarlyTerminationReasons(

MERGING\_EARLY\_TERMINATION\_REASON\_FILTERED);

}

if (trimStats.getResultsTruncatedFromTailCount() > 0) {

earlyTerminationInfo.addToMergedEarlyTerminationReasons(

MERGING\_EARLY\_TERMINATION\_REASON\_TRUNCATED);

}

}

private EarlyTerminationTrimmingStats getEarlyTerminationTrimmingStats() {

if (accumulatedResponses.isMergingPartitionsWithinATier()) {

return getEarlyTerminationTrimmingStatsForPartitions();

} else {

return getEarlyTerminationTrimmingStatsForTiers();

}

}

protected EarlyTerminationTrimmingStats getEarlyTerminationTrimmingStatsForPartitions() {

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

}

protected EarlyTerminationTrimmingStats getEarlyTerminationTrimmingStatsForTiers() {

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

}

/\*\*

\* If we get enough results, no need to go on.

\* If one of the partitions early terminated, we can't go on or else there could be a gap.

\*/

@Override

public boolean shouldEarlyTerminateTierMerge(int totalResultsFromSuccessfulShards,

boolean foundEarlyTermination) {

int resultsRequested = computeNumResultsToKeep();

boolean shouldEarlyTerminate = foundEarlyTermination

|| totalResultsFromSuccessfulShards >= resultsRequested;

if (shouldEarlyTerminate && totalResultsFromSuccessfulShards < resultsRequested) {

RECENCY\_TIER\_MERGE\_EARLY\_TERMINATED\_WITH\_NOT\_ENOUGH\_RESULTS.increment();

}

return shouldEarlyTerminate;

}

/\*\*

\* Find the min status id that has been \_completely\_ searched across all partitions. The

\* largest min status id across all partitions.

\*

\* @return the min searched status id found

\*/

protected long findMinFullySearchedStatusID() {

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

if (minIds.isEmpty()) {

return Long.MIN\_VALUE;

}

if (accumulatedResponses.isMergingPartitionsWithinATier()) {

// When merging partitions, the min ID should be the largest among the min IDs.

return Collections.max(accumulatedResponses.getMinIds());

} else {

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

return Collections.min(accumulatedResponses.getMinIds());

}

}

/\*\*

\* Find the max status id that has been \_completely\_ searched across all partitions. The

\* smallest max status id across all partitions.

\*

\* This is where we reconcile replication lag by selecting the oldest maxid from the

\* partitions searched.

\*

\* @return the max searched status id found

\*/

protected long findMaxFullySearchedStatusID() {

List<Long> maxIDs = accumulatedResponses.getMaxIds();

if (maxIDs.isEmpty()) {

return Long.MAX\_VALUE;

}

Collections.sort(maxIDs);

final long newest = maxIDs.get(maxIDs.size() - 1);

final long newestTimestamp = SnowflakeIdParser.getTimestampFromTweetId(newest);

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

long oldest = maxIDs.get(i);

long oldestTimestamp = SnowflakeIdParser.getTimestampFromTweetId(oldest);

long deltaMs = newestTimestamp - oldestTimestamp;

if (i == 0) {

LOG.debug("Max delta is {}", deltaMs);

}

if (deltaMs < ALLOWED\_REPLICATION\_LAG\_MS) {

if (i != 0) {

LOG.debug("{} partition replicas lagging more than {} ms", i, ALLOWED\_REPLICATION\_LAG\_MS);

}

return oldest;

}

}

// Can't get here - by this point oldest == newest, and delta is 0.

return newest;

}

/\*\*

\* Trim the ThriftSearchResults if we have enough results, to return the first

\* 'computeNumResultsToKeep()' number of results.

\*

\* If we don't have enough results after trimming, this function will first try to back fill

\* older results, then newer results

\*

\* @param searchResults ThriftSearchResults that hold the to be trimmed List<ThriftSearchResult>

\* @return TrimStats containing statistics about how many results being removed

\*/

protected TrimStats trimResults(

ThriftSearchResults searchResults,

long mergedMin,

long mergedMax) {

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

// no results, no trimming needed

return TrimStats.EMPTY\_STATS;

}

if (requestContext.getRequest().getSearchQuery().isSetSearchStatusIds()) {

// Not a normal search, no trimming needed

return TrimStats.EMPTY\_STATS;

}

TrimStats trimStats = new TrimStats();

trimExactDups(searchResults, trimStats);

int numResultsRequested = computeNumResultsToKeep();

if (shouldSkipTrimmingWhenNotEnoughResults(searchResults, numResultsRequested)) {

//////////////////////////////////////////////////////////

// We don't have enough results, let's not do trimming

//////////////////////////////////////////////////////////

return trimStats;

}

if (accumulatedResponses.isMergingPartitionsWithinATier()) {

trimResultsBasedSearchedRange(

searchResults, trimStats, numResultsRequested, mergedMin, mergedMax);

}

// Respect "computeNumResultsToKeep()" here, only keep "computeNumResultsToKeep()" results.

truncateResults(searchResults, trimStats);

return trimStats;

}

/\*\*

\* When there's not enough results, we don't remove results based on the searched range.

\* This has a tradeoff: with this, we don't reduce our recall when we already don't have enough

\* results. However, with this, we can lose results while paginating because we return results

\* outside of the valid searched range.

\*/

protected boolean shouldSkipTrimmingWhenNotEnoughResults(

ThriftSearchResults searchResults, int numResultsRequested) {

return searchResults.getResultsSize() <= numResultsRequested;

}

/\*\*

\* Trim results based on search range. The search range [x, y] is determined by:

\* x is the maximun of the minimun search IDs;

\* y is the minimun of the maximum search IDs.

\*

\* Ids out side of this range are removed.

\* If we do not get enough results after the removal, we add IDs back until we get enough results.

\* We first add IDs back from the older side back. If there's still not enough results,

\* we start adding IDs from the newer side back.

\*/

private void trimResultsBasedSearchedRange(ThriftSearchResults searchResults,

TrimStats trimStats,

int numResultsRequested,

long mergedMin,

long mergedMax) {

///////////////////////////////////////////////////////////////////

// we have more results than requested, let's do some trimming

///////////////////////////////////////////////////////////////////

// Save the original results before trimming

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

filterResultsByMergedMinMaxIds(searchResults, mergedMax, mergedMin, trimStats);

// This does happen. It is hard to say what we should do here so we just return the original

// result here.

if (searchResults.getResultsSize() == 0) {

RECENCY\_ZERO\_RESULT\_COUNT\_AFTER\_FILTERING\_MAX\_MIN\_IDS.increment();

searchResults.setResults(originalResults);

// Clean up min/mix filtered count, since we're bringing back whatever we just filtered.

trimStats.clearMaxIdFilterCount();

trimStats.clearMinIdFilterCount();

if (LOG.isDebugEnabled() || responseMessageBuilder.isDebugMode()) {

String errMsg = "No trimming is done as filtered results is empty. "

+ "maxId=" + mergedMax + ",minId=" + mergedMin;

LOG.debug(errMsg);

responseMessageBuilder.append(errMsg + "\n");

}

} else {

// oops! we're trimming too many results. Let's put some back

if (searchResults.getResultsSize() < numResultsRequested) {

RECENCY\_TRIMMED\_TOO\_MANY\_RESULTS\_COUNT.increment();

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

long firstTrimmedResultId = trimmedResults.get(0).getId();

long lastTrimmedResultId = trimmedResults.get(trimmedResults.size() - 1).getId();

// First, try to back fill with older results

int i = 0;

for (; i < originalResults.size(); ++i) {

ThriftSearchResult result = originalResults.get(i);

if (result.getId() < lastTrimmedResultId) {

trimmedResults.add(result);

trimStats.decreaseMinIdFilterCount();

if (trimmedResults.size() >= numResultsRequested) {

break;

}

}

}

// still not enough results? back fill with newer results

// find the oldest of the newer results

if (trimmedResults.size() < numResultsRequested) {

// still not enough results? back fill with newer results

// find the oldest of the newer results

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

ThriftSearchResult result = originalResults.get(i);

if (result.getId() > firstTrimmedResultId) {

trimmedResults.add(result);

trimStats.decreaseMaxIdFilterCount();

if (trimmedResults.size() >= numResultsRequested) {

break;

}

}

}

// newer results were added to the back of the list, re-sort

Collections.sort(trimmedResults, ResultComparators.ID\_COMPARATOR);

}

}

}

}

protected void setMergedMinSearchedStatusId(

ThriftSearchResults searchResults,

long currentMergedMin,

boolean resultsWereTrimmed) {

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

return;

}

long merged;

if (searchResults == null

|| !searchResults.isSetResults()

|| searchResults.getResultsSize() == 0) {

merged = currentMergedMin;

} else {

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

long firstResultId = results.get(0).getId();

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

merged = Math.min(firstResultId, lastResultId);

if (!resultsWereTrimmed) {

// If the results were trimmed, we want to set minSearchedStatusID to the smallest

// tweet ID in the response. Otherwise, we want to take the min between that, and

// the current minSearchedStatusID.

merged = Math.min(merged, currentMergedMin);

}

}

searchResults.setMinSearchedStatusID(merged);

}

private void setMergedMaxSearchedStatusId(

ThriftSearchResults searchResults,

long currentMergedMax) {

if (accumulatedResponses.getMaxIds().isEmpty()) {

return;

}

long merged;

if (searchResults == null

|| !searchResults.isSetResults()

|| searchResults.getResultsSize() == 0) {

merged = currentMergedMax;

} else {

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

long firstResultId = results.get(0).getId();

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

long maxResultId = Math.max(firstResultId, lastResultId);

merged = Math.max(maxResultId, currentMergedMax);

}

searchResults.setMaxSearchedStatusID(merged);

}

protected static void filterResultsByMergedMinMaxIds(

ThriftSearchResults results, long maxStatusId, long minStatusId, TrimStats trimStats) {

List<ThriftSearchResult> trimedResults =

Lists.newArrayListWithCapacity(results.getResultsSize());

for (ThriftSearchResult result : results.getResults()) {

long statusId = result.getId();

if (statusId > maxStatusId) {

trimStats.increaseMaxIdFilterCount();

} else if (statusId < minStatusId) {

trimStats.increaseMinIdFilterCount();

} else {

trimedResults.add(result);

}

}

results.setResults(trimedResults);

}

}