package com.twitter.search.earlybird.search.relevance.collectors;

import java.io.IOException;

import com.google.common.base.Preconditions;

import com.twitter.common.util.Clock;

import com.twitter.common\_internal.collections.RandomAccessPriorityQueue;

import com.twitter.search.common.relevance.features.TweetIntegerShingleSignature;

import com.twitter.search.common.schema.base.ImmutableSchemaInterface;

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

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

import com.twitter.search.earlybird.common.userupdates.UserTable;

import com.twitter.search.earlybird.search.relevance.RelevanceHit;

import com.twitter.search.earlybird.search.relevance.RelevanceSearchRequestInfo;

import com.twitter.search.earlybird.search.relevance.RelevanceSearchResults;

import com.twitter.search.earlybird.search.relevance.scoring.ScoringFunction;

import com.twitter.search.earlybird.stats.EarlybirdSearcherStats;

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

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

/\*\*

\* RelevanceTopCollector is a results collector that collects the top numResults by

\* score, filtering out duplicates.

\*/

public class RelevanceTopCollector extends AbstractRelevanceCollector {

// Search results are collected in a min-heap.

protected final RandomAccessPriorityQueue<RelevanceHit, TweetIntegerShingleSignature> minQueue;

// Number of hits actually added to the min queue after dupe filtering and skipping.

// Less than or equal to numHitsProcessed.

protected int numHitsCollected;

// The 'top' of the min heap, or, the lowest scored document in the heap.

private RelevanceHit pqTop;

private float lowestScore = ScoringFunction.SKIP\_HIT;

private final boolean isFilterDupes;

public RelevanceTopCollector(

ImmutableSchemaInterface schema,

RelevanceSearchRequestInfo searchRequestInfo,

ScoringFunction scoringFunction,

EarlybirdSearcherStats searcherStats,

EarlybirdCluster cluster,

UserTable userTable,

Clock clock,

int requestDebugMode) {

super(schema, searchRequestInfo, scoringFunction, searcherStats, cluster, userTable, clock,

requestDebugMode);

this.minQueue = new RandomAccessPriorityQueue<RelevanceHit, TweetIntegerShingleSignature>(

searchRequestInfo.getNumResultsRequested(), RelevanceHit.PQ\_COMPARATOR\_BY\_SCORE) {

@Override

protected RelevanceHit getSentinelObject() {

return new RelevanceHit(); // default relevance constructor would create a hit with the

// lowest score possible.

}

};

this.pqTop = minQueue.top();

this.isFilterDupes = getSearchRequestInfo().getRelevanceOptions().isFilterDups();

}

protected void collectWithScoreInternal(

long tweetID,

long timeSliceID,

float score,

ThriftSearchResultMetadata metadata) {

// This collector cannot handle these scores:

assert !Float.isNaN(score);

if (score <= lowestScore) {

// Since docs are returned in-order (i.e., increasing doc Id), a document

// with equal score to pqTop.score cannot compete since HitQueue favors

// documents with lower doc Ids. Therefore reject those docs too.

// IMPORTANT: docs skipped by the scoring function will have scores set

// to ScoringFunction.SKIP\_HIT, meaning they will not be collected.

return;

}

boolean dupFound = false;

Preconditions.checkState(metadata.isSetSignature(),

"The signature should be set at metadata collection time, but it is null. "

+ "Tweet id = %s, metadata = %s",

tweetID,

metadata);

int signatureInt = metadata.getSignature();

final TweetIntegerShingleSignature signature =

TweetIntegerShingleSignature.deserialize(signatureInt);

if (isFilterDupes) {

// update duplicate if any

if (signatureInt != TweetIntegerShingleSignature.DEFAULT\_NO\_SIGNATURE) {

dupFound = minQueue.incrementElement(

signature,

element -> {

if (score > element.getScore()) {

element.update(timeSliceID, tweetID, signature, metadata);

}

}

);

}

}

if (!dupFound) {

numHitsCollected++;

// if we didn't find a duplicate element to update then we add it now as a new element to the

// pq

pqTop = minQueue.updateTop(top -> top.update(timeSliceID, tweetID, signature, metadata));

lowestScore = pqTop.getScore();

}

}

@Override

protected void doCollectWithScore(final long tweetID, final float score) throws IOException {

ThriftSearchResultMetadata metadata = collectMetadata();

scoringFunction.populateResultMetadataBasedOnScoringData(

searchRequestInfo.getSearchQuery().getResultMetadataOptions(),

metadata,

scoringFunction.getScoringDataForCurrentDocument());

collectWithScoreInternal(tweetID, currTimeSliceID, score, metadata);

}

@Override

public EarlyTerminationState innerShouldCollectMore() {

// Note that numHitsCollected here might be less than num results collected in the

// TwitterEarlyTerminationCollector, if we hit dups or there are very low scores.

if (numHitsCollected >= getMaxHitsToProcess()) {

return setEarlyTerminationState(EarlyTerminationState.TERMINATED\_MAX\_HITS\_EXCEEDED);

}

return EarlyTerminationState.COLLECTING;

}

@Override

protected RelevanceSearchResults doGetRelevanceResults() throws IOException {

return getRelevanceResultsInternal();

}

protected RelevanceSearchResults getRelevanceResultsInternal() {

return resultsFromQueue(minQueue, getSearchRequestInfo().getNumResultsRequested(),

getRelevanceStats());

}

private static RelevanceSearchResults resultsFromQueue(

RandomAccessPriorityQueue<RelevanceHit, TweetIntegerShingleSignature> pq,

int desiredNumResults,

ThriftSearchResultsRelevanceStats relevanceStats) {

// trim first in case we didn't fill up the queue to not get any sentinel values here

int numResults = pq.trim();

if (numResults > desiredNumResults) {

for (int i = 0; i < numResults - desiredNumResults; i++) {

pq.pop();

}

numResults = desiredNumResults;

}

RelevanceSearchResults results = new RelevanceSearchResults(numResults);

// insert hits in decreasing order by score

for (int i = numResults - 1; i >= 0; i--) {

RelevanceHit hit = pq.pop();

results.setHit(hit, i);

}

results.setRelevanceStats(relevanceStats);

results.setNumHits(numResults);

return results;

}

}