package com.twitter.search.earlybird.search;

import java.io.IOException;

import java.util.Comparator;

import java.util.HashSet;

import java.util.Set;

import java.util.SortedSet;

import java.util.TreeSet;

import com.google.common.annotations.VisibleForTesting;

import org.apache.commons.lang.mutable.MutableInt;

import org.apache.lucene.index.IndexReader;

import org.apache.lucene.index.NumericDocValues;

import org.apache.lucene.index.Term;

import org.apache.lucene.search.Query;

import org.apache.lucene.search.ScoreMode;

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

import com.twitter.search.common.schema.earlybird.EarlybirdFieldConstants.EarlybirdFieldConstant;

import com.twitter.search.common.search.TwitterIndexSearcher;

import com.twitter.search.common.util.analysis.LongTermAttributeImpl;

import com.twitter.search.core.earlybird.index.EarlybirdIndexSegmentAtomicReader;

public class AntiGamingFilter {

private interface Acceptor {

boolean accept(int internalDocID) throws IOException;

}

private NumericDocValues userReputation;

private NumericDocValues fromUserIDs;

private final Query luceneQuery;

private boolean termsExtracted = false;

private final Set<Term> queryTerms;

// we ignore these user ids for anti-gaming filtering, because they were explicitly queried for

private Set<Long> segmentUserIDWhitelist = null;

// we gather the whitelisted userIDs from all segments here

private Set<Long> globalUserIDWhitelist = null;

/\*\*

\* Used to track the number of occurrences of a particular user.

\*/

private static final class UserCount

implements RandomAccessPriorityQueue.SignatureProvider<Long> {

private long userID;

private int count;

@Override

public Long getSignature() {

return userID;

}

@Override

public void clear() {

userID = 0;

count = 0;

}

}

private static final Comparator<UserCount> USER\_COUNT\_COMPARATOR =

(d1, d2) -> d1.count == d2.count ? Long.compare(d1.userID, d2.userID) : d1.count - d2.count;

private final RandomAccessPriorityQueue<UserCount, Long> priorityQueue =

new RandomAccessPriorityQueue<UserCount, Long>(1024, USER\_COUNT\_COMPARATOR) {

@Override

protected UserCount getSentinelObject() {

return new UserCount();

}

};

private final Acceptor acceptor;

private final int maxHitsPerUser;

/\*\*

\* Creates an AntiGamingFilter that either accepts or rejects tweets from all users.

\* This method should only be called in tests.

\*

\* @param alwaysValue Determines if tweets should always be accepted or rejected.

\* @return An AntiGamingFilter that either accepts or rejects tweets from all users.

\*/

@VisibleForTesting

public static AntiGamingFilter newMock(boolean alwaysValue) {

return new AntiGamingFilter(alwaysValue) {

@Override

public void startSegment(EarlybirdIndexSegmentAtomicReader reader) {

}

};

}

private AntiGamingFilter(boolean alwaysValue) {

acceptor = internalDocID -> alwaysValue;

maxHitsPerUser = Integer.MAX\_VALUE;

termsExtracted = true;

luceneQuery = null;

queryTerms = null;

}

public AntiGamingFilter(int maxHitsPerUser, int maxTweepCred, Query luceneQuery) {

this.maxHitsPerUser = maxHitsPerUser;

this.luceneQuery = luceneQuery;

if (maxTweepCred != -1) {

this.acceptor = internalDocID -> {

long userReputationVal =

userReputation.advanceExact(internalDocID) ? userReputation.longValue() : 0L;

return ((byte) userReputationVal > maxTweepCred) || acceptUser(internalDocID);

};

} else {

this.acceptor = this::acceptUser;

}

this.queryTerms = new HashSet<>();

}

public Set<Long> getUserIDWhitelist() {

return globalUserIDWhitelist;

}

private boolean acceptUser(int internalDocID) throws IOException {

final long fromUserID = getUserId(internalDocID);

final MutableInt freq = new MutableInt();

// try to increment UserCount for an user already exist in the priority queue.

boolean incremented = priorityQueue.incrementElement(

fromUserID, element -> freq.setValue(++element.count));

// If not incremented, it means the user node does not exist in the priority queue yet.

if (!incremented) {

priorityQueue.updateTop(element -> {

element.userID = fromUserID;

element.count = 1;

freq.setValue(element.count);

});

}

if (freq.intValue() <= maxHitsPerUser) {

return true;

} else if (segmentUserIDWhitelist == null) {

return false;

}

return segmentUserIDWhitelist.contains(fromUserID);

}

/\*\*

\* Initializes this filter with the new feature source. This method should be called every time an

\* earlybird searcher starts searching in a new segment.

\*

\* @param reader The reader for the new segment.

\*/

public void startSegment(EarlybirdIndexSegmentAtomicReader reader) throws IOException {

if (!termsExtracted) {

extractTerms(reader);

}

fromUserIDs =

reader.getNumericDocValues(EarlybirdFieldConstant.FROM\_USER\_ID\_CSF.getFieldName());

// fill the id whitelist for the current segment. initialize lazily.

segmentUserIDWhitelist = null;

SortedSet<Integer> sortedFromUserDocIds = new TreeSet<>();

for (Term t : queryTerms) {

if (t.field().equals(EarlybirdFieldConstant.FROM\_USER\_ID\_FIELD.getFieldName())) {

// Add the operand of the from\_user\_id operator to the whitelist

long fromUserID = LongTermAttributeImpl.copyBytesRefToLong(t.bytes());

addUserToWhitelists(fromUserID);

} else if (t.field().equals(EarlybirdFieldConstant.FROM\_USER\_FIELD.getFieldName())) {

// For a [from X] filter, we need to find a document that has the from\_user field set to X,

// and then we need to get the value of the from\_user\_id field for that document and add it

// to the whitelist. We can get the from\_user\_id value from the fromUserIDs NumericDocValues

// instance, but we need to traverse it in increasing order of doc IDs. So we add a doc ID

// for each term to a sorted set for now, and then we traverse it in increasing doc ID order

// and add the from\_user\_id values for those docs to the whitelist.

int firstInternalDocID = reader.getNewestDocID(t);

if (firstInternalDocID != EarlybirdIndexSegmentAtomicReader.TERM\_NOT\_FOUND) {

sortedFromUserDocIds.add(firstInternalDocID);

}

}

}

for (int fromUserDocId : sortedFromUserDocIds) {

addUserToWhitelists(getUserId(fromUserDocId));

}

userReputation =

reader.getNumericDocValues(EarlybirdFieldConstant.USER\_REPUTATION.getFieldName());

// Reset the fromUserIDs NumericDocValues so that the acceptor can use it to iterate over docs.

fromUserIDs =

reader.getNumericDocValues(EarlybirdFieldConstant.FROM\_USER\_ID\_CSF.getFieldName());

}

private void extractTerms(IndexReader reader) throws IOException {

Query query = luceneQuery;

for (Query rewrittenQuery = query.rewrite(reader); rewrittenQuery != query;

rewrittenQuery = query.rewrite(reader)) {

query = rewrittenQuery;

}

// Create a new TwitterIndexSearcher instance here instead of an IndexSearcher instance, to use

// the TwitterIndexSearcher.collectionStatistics() implementation.

query.createWeight(new TwitterIndexSearcher(reader), ScoreMode.COMPLETE, 1.0f)

.extractTerms(queryTerms);

termsExtracted = true;

}

public boolean accept(int internalDocID) throws IOException {

return acceptor.accept(internalDocID);

}

private void addUserToWhitelists(long userID) {

if (this.segmentUserIDWhitelist == null) {

this.segmentUserIDWhitelist = new HashSet<>();

}

if (this.globalUserIDWhitelist == null) {

this.globalUserIDWhitelist = new HashSet<>();

}

this.segmentUserIDWhitelist.add(userID);

this.globalUserIDWhitelist.add(userID);

}

@VisibleForTesting

protected long getUserId(int internalDocId) throws IOException {

return fromUserIDs.advanceExact(internalDocId) ? fromUserIDs.longValue() : 0L;

}

}