package com.twitter.search.earlybird.queryparser;

import java.util.ArrayList;

import java.util.IdentityHashMap;

import com.google.common.base.Preconditions;

import com.twitter.search.common.util.text.HighFrequencyTermPairs;

import com.twitter.search.queryparser.query.BooleanQuery;

import com.twitter.search.queryparser.query.Conjunction;

import com.twitter.search.queryparser.query.Disjunction;

import com.twitter.search.queryparser.query.Operator;

import com.twitter.search.queryparser.query.Phrase;

import com.twitter.search.queryparser.query.Query;

import com.twitter.search.queryparser.query.QueryParserException;

import com.twitter.search.queryparser.query.QueryVisitor;

import com.twitter.search.queryparser.query.SpecialTerm;

import com.twitter.search.queryparser.query.Term;

import com.twitter.search.queryparser.query.annotation.Annotation;

/\*\*

\* Iterates over the Query, populating information of an ArrayList of HighFrequencyTermQueryGroup so that

\* HighFrequencyTermPairRewriteVisitor can rewrite the query to use hf term pairs. Returns the

\* (approximate) number of high frequency terms it has detected. Iff that number is greater than 1

\* it MAY be able to rewrite the query to use the hf\_term\_pairs field.

\*

\* The key to HF Term Pair rewriting is understanding which nodes can be combined. This extractor

\* accomplishes this job by grouping nodes of the query together. All positive children of a

\* conjunction are grouped together, and all negative children of a disjunction are grouped

\* together. The end result is a tree of groups, where every child of a single group will have the

\* opposite value of isPositive of the parent group.

\*

\* I'll try to break it down a bit further. Let's assume "a" and "b" are hf terms, and '

\* "[hf\_term\_pair a b]" represents querying their co-occurence.

\* Query (\* a b not\_hf) can become (\* [hf\_term\_pair a b] not\_hf)

\* Query (+ -a -b -not\_hf) can become (+ -[hf\_term\_pair a b] -not\_hf)

\* These two rules represent the bulk of the rewrites that this class makes.

\*

\* We also keep track of another form of rewrite. A member of a group can be paired up with a member

\* of any of its parent groups as long as both groups have the same isPositive value. This

\* operation mimics boolean distribution. As this is probably better explained with an example:

\* Query (\* a (+ not\_hf (\* b not\_hf2))) can become (\* a (+ not\_hf (\* [hf\_term\_pair a b ] not\_hf2)))

\* Query (+ -a (\* not\_hf (+ -b not\_hf2))) can become (+ -a (\* not\_hf (+ -[hf\_term\_pair a b] not\_hf2)))

\*/

public class HighFrequencyTermPairExtractor extends QueryVisitor<Integer> {

private final ArrayList<HighFrequencyTermQueryGroup> groupList;

private final IdentityHashMap<Query, Integer> groupIds;

public HighFrequencyTermPairExtractor(ArrayList<HighFrequencyTermQueryGroup> groupList,

IdentityHashMap<Query, Integer> groupIds) {

Preconditions.checkNotNull(groupList);

Preconditions.checkArgument(groupList.isEmpty());

this.groupList = groupList;

this.groupIds = groupIds;

}

@Override

public Integer visit(Disjunction disjunction) throws QueryParserException {

return visit((BooleanQuery) disjunction);

}

@Override

public Integer visit(Conjunction conjunction) throws QueryParserException {

return visit((BooleanQuery) conjunction);

}

/\*\*

\* All positive children under a conjunction (negative children under disjunction) belong in the

\* same group as booleanQuery. All other children belong in their own, separate, new groups.

\* @param booleanQuery

\* @return Number of high frequency terms seen by this node and its children

\* @throws QueryParserException

\*/

private Integer visit(BooleanQuery booleanQuery) throws QueryParserException {

HighFrequencyTermQueryGroup group = getGroupForQuery(booleanQuery);

int numHits = 0;

for (Query node : booleanQuery.getChildren()) {

boolean neg = node.mustNotOccur();

if (node.isTypeOf(Query.QueryType.DISJUNCTION)) {

// Disjunctions, being negative conjunctions, are inherently negative nodes. In terms of

// being in a positive or negative group, we must flip their Occur value.

neg = !neg;

}

if (booleanQuery.isTypeOf(Query.QueryType.DISJUNCTION) && node.mustOccur()) {

// Potential Example: (\* a (+ +b not\_c)) => (\* (+ +b not\_c) [hf\_term\_pair a b 0.05])

// Implementation is too difficult and would make this rewriter even MORE complicated for

// a rarely used query. For now, we ignore it completely. We might gain some benefit in the

// future if we decide to create a new extractor and rewriter and rewrite this subquery, and

// that wouldn't complicate things too much.

continue;

}

if (booleanQuery.isTypeOf(Query.QueryType.CONJUNCTION) != neg) { // Add node to current group

groupIds.put(node, group.groupIdx);

group.numMembers++;

} else { // Create a new group

HighFrequencyTermQueryGroup newGroup =

new HighFrequencyTermQueryGroup(groupList.size(), group.groupIdx, !group.isPositive);

newGroup.numMembers++;

groupIds.put(node, newGroup.groupIdx);

groupList.add(newGroup);

}

numHits += node.accept(this);

}

return numHits;

}

@Override

public Integer visit(Phrase phrase) throws QueryParserException {

HighFrequencyTermQueryGroup group = getGroupForQuery(phrase);

int numFound = 0;

if (!phrase.hasAnnotationType(Annotation.Type.OPTIONAL)) {

boolean canBeRewritten = false;

// Special case: phrases with exactly 2 terms that are both high frequency can be

// rewritten. In all other cases terms will be treated as pre-used hf term phrases.

if (!phrase.hasAnnotations() && phrase.size() == 2

&& HighFrequencyTermPairs.HF\_TERM\_SET.contains(phrase.getTerms().get(0))

&& HighFrequencyTermPairs.HF\_TERM\_SET.contains(phrase.getTerms().get(1))) {

canBeRewritten = true;

}

// Special case: do not treat phrase containing :prox annotation as a real phrase.

boolean proximityPhrase = phrase.hasAnnotationType(Annotation.Type.PROXIMITY);

String lastHFToken = null;

for (String token : phrase.getTerms()) {

if (HighFrequencyTermPairs.HF\_TERM\_SET.contains(token)) {

group.preusedHFTokens.add(token);

if (group.distributiveToken == null) {

group.distributiveToken = token;

}

if (lastHFToken != null && !proximityPhrase) {

if (canBeRewritten) {

group.hfPhrases.add(lastHFToken + " " + token);

} else {

group.preusedHFPhrases.add(lastHFToken + " " + token);

}

}

lastHFToken = token;

numFound++;

} else {

lastHFToken = null;

}

}

}

return numFound;

}

@Override

public Integer visit(Term term) throws QueryParserException {

if (groupList.isEmpty()) { // Shortcut for 1 term queries.

return 0;

}

HighFrequencyTermQueryGroup group = getGroupForQuery(term);

if (!term.hasAnnotationType(Annotation.Type.OPTIONAL)

&& HighFrequencyTermPairs.HF\_TERM\_SET.contains(term.getValue())) {

if (!term.hasAnnotations()) {

group.hfTokens.add(term.getValue());

} else { // Should not remove the annotated term.

group.preusedHFTokens.add(term.getValue());

}

if (group.distributiveToken == null) {

group.distributiveToken = term.getValue();

}

return 1;

}

return 0;

}

@Override

public Integer visit(Operator operator) throws QueryParserException {

return 0;

}

@Override

public Integer visit(SpecialTerm special) throws QueryParserException {

return 0;

}

/\*\*

\* Uses the query's visitor data as an index and returns the group it belongs to. If groupList is

\* empty, create a new group and set this group's visitor data to be index 0.

\* @param query

\* @return the group which query belongs to.

\*/

private HighFrequencyTermQueryGroup getGroupForQuery(Query query) {

if (groupList.isEmpty()) {

boolean pos = !query.mustNotOccur();

if (query instanceof Disjunction) {

pos = !pos;

}

HighFrequencyTermQueryGroup group = new HighFrequencyTermQueryGroup(0, pos);

group.numMembers++;

groupList.add(group);

groupIds.put(query, 0);

}

return groupList.get(groupIds.get(query));

}

}