package com.twitter.search.earlybird.queryparser;

import java.util.ArrayList;

import java.util.IdentityHashMap;

import java.util.List;

import java.util.Set;

import javax.annotation.Nullable;

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.SearchRateCounter;

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

import com.twitter.search.earlybird.common.config.EarlybirdConfig;

import com.twitter.search.queryparser.parser.SerializedQueryParser;

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.QueryNodeUtils;

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.search.SearchOperator;

/\*\*

\* Iterates over the Query, modifying it to include high frequency term pairs, replacing

\* singular high frequency terms where possible.

\*

\* Assumes that this will be used IMMEDIATELY after using HighFrequencyTermPairExtractor

\*

\* There are two primary functions of this visitor:

\* 1. Append hf\_term\_pairs to each group's root node.

\* 2. Remove all unnecessary term queries (unnecessary as they are captured by an hf\_term\_pair)

\*

\* Every time the visitor finishes visiting a node, HighFrequencyTermQueryGroup.numVisits will be

\* incremented for that node's group. When numVisits == numChildren, we know we have just finished

\* processing the root of the group. At this point, we must append relevant hf\_term\_pairs to this

\* node.

\*/

public class HighFrequencyTermPairRewriteVisitor extends QueryVisitor<Query> {

private static final Logger LOG = LoggerFactory.getLogger(

HighFrequencyTermPairRewriteVisitor.class);

private static final SearchRateCounter SEARCH\_HF\_PAIR\_COUNTER =

SearchRateCounter.export("hf\_pair\_rewrite");

private final ArrayList<HighFrequencyTermQueryGroup> groupList;

private final IdentityHashMap<Query, Integer> groupIds;

private final boolean allowNegativeOrRewrite;

/\*\*

\* Creates a new HighFrequencyTermPairRewriteVisitor. Should be used only IMMEDIATELY after using

\* a HighFrequencyTermPairExtractor

\* @param groupList The groups extracted using HighFrequencyTermPairExtractor

\* @param groupIds the mapping from query to the HF term query group

\*/

public HighFrequencyTermPairRewriteVisitor(ArrayList<HighFrequencyTermQueryGroup> groupList,

IdentityHashMap<Query, Integer> groupIds) {

this(groupList, groupIds, true);

}

/\*\*

\* Creates a new HighFrequencyTermPairRewriteVisitor. Should be used only IMMEDIATELY after using

\* a HighFrequencyTermPairExtractor

\* @param groupList The groups extracted using HighFrequencyTermPairExtractor

\* @param groupIds the mapping from query to the HF term query group

\* @param allowNegativeOrRewrite whether to allow rewrite for 'or (-terms)'

\*/

public HighFrequencyTermPairRewriteVisitor(ArrayList<HighFrequencyTermQueryGroup> groupList,

IdentityHashMap<Query, Integer> groupIds,

boolean allowNegativeOrRewrite) {

this.groupList = groupList;

this.groupIds = groupIds;

this.allowNegativeOrRewrite = allowNegativeOrRewrite;

}

/\*\*

\* This method logs successful rewrites, and protects against unsuccessful ones by

\* catching all exceptions and restoring the previous query.

\*/

public static Query safeRewrite(Query safeQuery, boolean allowNegativeOrRewrite)

throws QueryParserException {

Query query = safeQuery;

ArrayList<HighFrequencyTermQueryGroup> groups = Lists.newArrayList();

IdentityHashMap<Query, Integer> groupIds = Maps.newIdentityHashMap();

// Step 1: extract high frequency term pairs and phrases.

try {

int hfTermsFound = query.accept(new HighFrequencyTermPairExtractor(groups, groupIds));

if (hfTermsFound < 2) {

return query;

}

} catch (Exception e) {

LOG.error("Exception while extracting high frequency term pairs", e);

return query;

}

// Step 2: rewrite (safely).

String original = query.serialize();

try {

query = query.accept(

new HighFrequencyTermPairRewriteVisitor(groups, groupIds, allowNegativeOrRewrite))

.simplify();

String rewrite = query.serialize();

if (LOG.isDebugEnabled()) {

LOG.debug("Optimized query: " + original + " -> " + rewrite);

}

SEARCH\_HF\_PAIR\_COUNTER.increment();

return query;

} catch (Exception e) {

LOG.error("Exception rewriting high frequency term pairs", e);

return new SerializedQueryParser(EarlybirdConfig.getPenguinVersion()).parse(original);

}

}

/\*\*

\* The rewritten query to use the hf\_term\_pair operators.

\*

\* @param disjunction query node which must have been previously visited by

\* HighFrequencyTermPairExtractor and not had its visitor data cleared.

\*/

@Override

public Query visit(Disjunction disjunction) throws QueryParserException {

return visit((BooleanQuery) disjunction);

}

/\*\*

\* The rewritten query to use the hf\_term\_pair operators.

\*

\* @param conjunction query node which must have been previously visited by

\* HighFrequencyTermPairExtractor and not had its visitor data cleared.

\*/

@Override

public Query visit(Conjunction conjunction) throws QueryParserException {

return visit((BooleanQuery) conjunction);

}

/\*\*

\* Applies this visitor to a BooleanQuery.

\*/

public Query visit(BooleanQuery booleanQuery) throws QueryParserException {

HighFrequencyTermQueryGroup group = groupList.get(groupIds.get(booleanQuery));

queryPreprocess(group);

ArrayList<Query> children = Lists.newArrayList();

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

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.

children.add(node);

continue;

}

Query child = node.accept(this);

if (child != null) {

children.add(child);

}

}

Query newBooleanQuery = booleanQuery.newBuilder().setChildren(children).build();

return queryPostprocess(newBooleanQuery, group);

}

/\*\*

\* The rewritten query to use the hf\_term\_pair operators.

\*

\* @param phraseToVisit query node which must have been previously visited by

\* HighFrequencyTermPairExtractor and not had its visitor data cleared.

\*/

@Override

public Query visit(Phrase phraseToVisit) throws QueryParserException {

Phrase phrase = phraseToVisit;

HighFrequencyTermQueryGroup group = groupList.get(groupIds.get(phrase));

queryPreprocess(group);

// Remove all high frequency phrases from the query that do not have any annotations.

// This will cause phrase de-duping, which we probably don't care about.

if (!hasAnnotations(phrase) && (

group.hfPhrases.contains(phrase.getPhraseValue())

|| group.preusedHFPhrases.contains(phrase.getPhraseValue()))) {

// This term will be appended to the end of the query in the form of a pair.

phrase = null;

}

return queryPostprocess(phrase, group);

}

/\*\*

\* The rewritten query to use the hf\_term\_pair operators.

\*

\* @param termToVisit query node which must have been previously visited by

\* HighFrequencyTermPairExtractor and not had its visitor data cleared.

\*/

@Override

public Query visit(Term termToVisit) throws QueryParserException {

Term term = termToVisit;

HighFrequencyTermQueryGroup group = groupList.get(groupIds.get(term));

queryPreprocess(group);

// Remove all high frequency terms from the query that do not have any annotations. This will

// do term de-duping within a group, which may effect scoring, but since these are high df

// terms, they don't have much of an impact anyways.

if (!hasAnnotations(term)

&& (group.preusedHFTokens.contains(term.getValue())

|| group.hfTokens.contains(term.getValue()))) {

// This term will be appended to the end of the query in the form of a pair.

term = null;

}

return queryPostprocess(term, group);

}

/\*\*

\* The rewritten query to use the hf\_term\_pair operators.

\*

\* @param operator query node which must have been previously visited by

\* HighFrequencyTermPairExtractor and not had its visitor data cleared.

\*/

@Override

public Query visit(Operator operator) throws QueryParserException {

HighFrequencyTermQueryGroup group = groupList.get(groupIds.get(operator));

queryPreprocess(group);

return queryPostprocess(operator, group);

}

/\*\*

\* The rewritten query to use the hf\_term\_pair operators.

\*

\* @param special query node which must have been previously visited by

\* HighFrequencyTermPairExtractor and not had its visitor data cleared.

\*/

@Override

public Query visit(SpecialTerm special) throws QueryParserException {

HighFrequencyTermQueryGroup group = groupList.get(groupIds.get(special));

queryPreprocess(group);

return queryPostprocess(special, group);

}

/\*\*

\* Before visiting a node's children, we must process its group's distributiveToken. This way, a

\* node only has to check its grandparent group for a distributiveToken instead of recursing all

\* of the way up to the root of the tree.

\*/

private void queryPreprocess(HighFrequencyTermQueryGroup group) {

if (group.distributiveToken == null) {

group.distributiveToken = getAncestorDistributiveToken(group);

}

}

/\*\*

\* If the query isn't the root of the group, returns the query. Otherwise, if the query's

\* group has at most one hf term, return the query. Otherwise, returns the query with hf\_term\_pair

\* operators created from the group's hf terms appended to it.

\*/

private Query queryPostprocess(@Nullable Query query, HighFrequencyTermQueryGroup group)

throws QueryParserException {

group.numVisits++;

if (group.numMembers == group.numVisits

&& (!group.hfTokens.isEmpty() || !group.preusedHFTokens.isEmpty()

|| group.hasPhrases())) {

group.removePreusedTokens();

String ancestorDistributiveToken = getAncestorDistributiveToken(group);

// Need at least 2 tokens to perform a pair rewrite. Try to get one

// additional token from ancestors, and if that fails, from phrases.

if ((group.hfTokens.size() + group.preusedHFTokens.size()) == 1

&& ancestorDistributiveToken != null) {

group.preusedHFTokens.add(ancestorDistributiveToken);

}

if ((group.hfTokens.size() + group.preusedHFTokens.size()) == 1) {

String tokenFromPhrase = group.getTokenFromPhrase();

if (tokenFromPhrase != null) {

group.preusedHFTokens.add(tokenFromPhrase);

}

}

return appendPairs(query, group);

}

return query;

}

/\*\*

\* Returns the distributiveToken of group's grandparent.

\*/

private String getAncestorDistributiveToken(HighFrequencyTermQueryGroup group) {

String ancestorDistributiveToken = null;

if (group.parentGroupIdx >= 0 && groupList.get(group.parentGroupIdx).parentGroupIdx >= 0) {

ancestorDistributiveToken =

groupList.get(groupList.get(group.parentGroupIdx).parentGroupIdx).distributiveToken;

}

return ancestorDistributiveToken;

}

/\*\*

\* Returns the hf\_term\_pair operators created using the hf terms of the group appended to query.

\*

\* @param query The query which the new hf\_term\_pair operators will be appended to.

\* @param group The group which this query belongs to.

\* @return The hf\_term\_pair operators created using the hf terms of the group appended to query.

\*/

private Query appendPairs(@Nullable Query query, HighFrequencyTermQueryGroup group)

throws QueryParserException {

BooleanQuery query2 = createQueryFromGroup(group);

// If either of the queries are null, do not have to worry about combining them.

if (query2 == null) {

return query;

} else if (query == null) {

return query2;

}

Query newQuery;

if (query.isTypeOf(Query.QueryType.CONJUNCTION)

|| query.isTypeOf(Query.QueryType.DISJUNCTION)) {

// Adding children in this way is safer when its query is a conjunction or disjunction

// ex. Other way: (+ +de -la -the) => (+ (+ +de -la -the) -[hf\_term\_pair la the 0.005])

// This way: (+ +de -la -the) => (+ +de -la -the -[hf\_term\_pair la the 0.005])

return ((BooleanQuery.Builder) query.newBuilder()).addChildren(query2.getChildren()).build();

} else if (!group.isPositive) {

// In lucene, [+ (-term1, -term2, ...)] has non-deterministic behavior and the rewrite is not

// efficient from query execution perspective. So, we will not do this rewrite if it is

// configured that way.

if (!allowNegativeOrRewrite) {

return query;

}

// Negate both queries to combine, and the append as a conjunction, followed by negating

// whole query. Equivalent to appending as a disjunction.

newQuery = QueryNodeUtils.appendAsConjunction(

query.negate(),

query2.negate()

);

newQuery = newQuery.makeMustNot();

} else {

newQuery = QueryNodeUtils.appendAsConjunction(query, query2);

newQuery = newQuery.makeDefault();

}

return newQuery;

}

/\*\*

\* Creates a conjunction of term\_pairs using the sets of hf terms in HighFrequencyTermQueryGroup

\* group. If !group.isPositive, will return a disjunction of negated pairs. If there aren't enough

\* hfTokens, will return null.

\*/

private BooleanQuery createQueryFromGroup(HighFrequencyTermQueryGroup group)

throws QueryParserException {

if (!group.hfTokens.isEmpty() || group.preusedHFTokens.size() > 1 || group.hasPhrases()) {

List<Query> terms = createTermPairsForGroup(group.hfTokens,

group.preusedHFTokens,

group.hfPhrases,

group.preusedHFPhrases);

if (group.isPositive) {

return new Conjunction(terms);

} else {

return new Disjunction(Lists.transform(terms, QueryNodeUtils.NEGATE\_QUERY));

}

}

return null;

}

/\*\*

\* Creates HF\_TERM\_PAIR terms out of hfTokens and optHFTokens. Attempts to create the minimal

\* amount of tokens necessary. optHFToken pairs should be given a weight of 0.0 and not be scored,

\* as they are likely already included in the query in a phrase or an annotated term.

\* @param hfTokens

\* @param optHFTokens

\* @return A list of hf\_term\_pair operators.

\*/

private List<Query> createTermPairsForGroup(Set<String> hfTokens,

Set<String> optHFTokens,

Set<String> hfPhrases,

Set<String> optHFPhrases) {

// Handle sets with only one token.

if (optHFTokens.size() == 1 && hfTokens.size() > 0) {

// (\* "a not\_hf" b c) => (\* "a not\_hf" [hf\_term\_pair a b 0.05] [hf\_term\_pair b c 0.05])

// optHFTokens: [a] hfTokens: [b, c] => optHFTokens: [] hfTokens: [a, b, c]

hfTokens.addAll(optHFTokens);

optHFTokens.clear();

} else if (hfTokens.size() == 1 && optHFTokens.size() > 0) {

// (\* "a b" not\_hf c) => (\* "a b" not\_hf [hf\_term\_pair a b 0.0] [hf\_term\_pair a c 0.005])

// optHFTokens: [a, b] hfTokens: [c] => optHFTokens: [a, b] hfTokens: [a, c]

String term = optHFTokens.iterator().next();

hfTokens.add(term);

}

List<Query> terms = createTermPairs(hfTokens, true, HighFrequencyTermPairs.HF\_DEFAULT\_WEIGHT);

terms.addAll(createTermPairs(optHFTokens, false, 0));

terms.addAll(createPhrasePairs(hfPhrases, HighFrequencyTermPairs.HF\_DEFAULT\_WEIGHT));

terms.addAll(createPhrasePairs(optHFPhrases, 0));

return terms;

}

/\*\*

\* Turns a set of hf terms into a list of hf\_term\_pair operators. Each term will be used at least

\* once in as few pairs as possible.

\* @param tokens

\* @param createSingle If the set contains only one query, the returned list will contain a single

\* Term for that query if createSingle is true, and an empty list otherwise.

\* @param weight Each term pair will be given a score boost of serializedWeight.

\* @return

\*/

private static List<Query> createTermPairs(Set<String> tokens, boolean createSingle,

double weight) {

List<Query> terms = Lists.newArrayList();

if (tokens.size() >= 2) {

int tokensLeft = tokens.size();

String token1 = null;

for (String token2 : tokens) {

if (token1 == null) {

token1 = token2;

} else {

terms.add(createHFTermPair(token1, token2, weight));

if (tokensLeft > 2) { // Only reset if there is more than one token remaining.

token1 = null;

}

}

tokensLeft--;

}

} else if (createSingle && !tokens.isEmpty()) { // Only one high frequency token

// Need to add token as a term because it was removed from the query earlier in rewriting.

Term newTerm = new Term(tokens.iterator().next());

terms.add(newTerm);

}

return terms;

}

private static List<Query> createPhrasePairs(Set<String> phrases, double weight) {

List<Query> ops = Lists.newArrayList();

for (String phrase : phrases) {

String[] terms = phrase.split(" ");

assert terms.length == 2;

SearchOperator op = new SearchOperator(SearchOperator.Type.HF\_PHRASE\_PAIR,

terms[0], terms[1], Double.toString(weight));

ops.add(op);

}

return ops;

}

private static SearchOperator createHFTermPair(String token1, String token2, double weight) {

SearchOperator op = new SearchOperator(SearchOperator.Type.HF\_TERM\_PAIR,

token1, token2, Double.toString(weight));

return op;

}

private static boolean hasAnnotations(com.twitter.search.queryparser.query.Query node) {

return node.hasAnnotations();

}

}