package com.twitter.search.common.query;

import java.util.Collections;

import java.util.List;

import java.util.Map;

import java.util.Set;

import java.util.logging.Level;

import java.util.logging.Logger;

import com.google.common.collect.Sets;

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

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

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.SpecialTerm;

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

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

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

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

/\*\*

\* Visitor to track the fields hits of each node

\* Returns the common fields among conjunctions and the union of the fields amongst disjunctions

\*/

public final class QueryCommonFieldHitsVisitor extends SearchQueryVisitor<Set<String>> {

private static final Logger LOG = Logger.getLogger(QueryCommonFieldHitsVisitor.class.getName());

private Map<Query, Integer> nodeToRankMap;

private Map<Integer, List<String>> hitFieldsByRank;

/\*\*

\* Find query term hit intersections based on hitmap given by HitAttributeHelper

\*

\* @param hitAttributeHelper the HitAttributeHelper

\* @param docID documentID

\* @param query the query searched

\* @return a set of hit fields in String representation

\*/

public static Set<String> findIntersection(

HitAttributeHelper hitAttributeHelper,

int docID,

Query query) {

return findIntersection(hitAttributeHelper.getNodeToRankMap(),

hitAttributeHelper.getHitAttribution(docID),

query);

}

/\*\*

\* Find query term hit intersections based on hitmap given by HitAttributeHelper

\*

\* @param nodeToRankMap the map of query node to its integer rank value

\* @param hitFieldsByRank map of rank to list of hit fields in String representation

\* @param query the query searched

\* @return a set of hit fields in String representation

\*/

public static Set<String> findIntersection(

Map<Query, Integer> nodeToRankMap,

Map<Integer, List<String>> hitFieldsByRank,

Query query) {

QueryCommonFieldHitsVisitor visitor =

new QueryCommonFieldHitsVisitor(nodeToRankMap, hitFieldsByRank);

try {

Set<String> returnSet = query.accept(visitor);

return returnSet;

} catch (QueryParserException e) {

LOG.log(Level.SEVERE, "Could not find intersection for query [" + query + "]: ", e);

return Collections.emptySet();

}

}

private QueryCommonFieldHitsVisitor(Map<Query, Integer> nodeToRankMap,

Map<Integer, List<String>> hitFieldsByRank) {

this.nodeToRankMap = nodeToRankMap;

this.hitFieldsByRank = hitFieldsByRank;

}

@Override

public Set<String> visit(Disjunction disjunction) throws QueryParserException {

Set<String> fieldHitIntersections = Sets.newHashSet();

for (Query child : disjunction.getChildren()) {

fieldHitIntersections.addAll(child.accept(this));

}

return fieldHitIntersections;

}

@Override

public Set<String> visit(Conjunction conjunction) throws QueryParserException {

List<Query> children = conjunction.getChildren();

if (!children.isEmpty()) {

boolean initializedIntersections = false;

Set<String> fieldHitIntersections = Sets.newHashSet();

for (Query child : children) {

Set<String> hits = child.accept(this);

if (hits.isEmpty()) {

// if it is empty, it means this query node is not of term type

// and we do not include these in the field intersection

// eg. cache filters, proximity groups

continue;

}

if (!initializedIntersections) {

fieldHitIntersections.addAll(hits);

initializedIntersections = true;

} else {

fieldHitIntersections.retainAll(hits);

}

}

return fieldHitIntersections;

}

return Collections.emptySet();

}

@Override

public Set<String> visit(Term term) throws QueryParserException {

Set<String> fieldHitIntersections = Sets.newHashSet();

Integer rank = nodeToRankMap.get(term);

if (rank != null) {

List<String> fields = hitFieldsByRank.get(rank);

// for disjunction cases where a term may not have any hits

if (fields != null) {

fieldHitIntersections.addAll(fields);

}

}

return fieldHitIntersections;

}

@Override

public Set<String> visit(SpecialTerm specialTerm) throws QueryParserException {

// This is way of splitting @mentions ensures consistency with way the lucene query is built in

// expertsearch

if (specialTerm.getType() == SpecialTerm.Type.MENTION && specialTerm.getValue().contains("\_")) {

Phrase phrase = new Phrase(specialTerm.getValue().split("\_"));

return phrase.accept(this);

}

return specialTerm.toTermOrPhrase().accept(this);

}

@Override

public Set<String> visit(SearchOperator operator) throws QueryParserException {

return Collections.emptySet();

}

@Override

public Set<String> visit(Link link) throws QueryParserException {

return link.toPhrase().accept(this);

}

@Override

public Set<String> visit(Phrase phrase) throws QueryParserException {

// All terms in the phrase should return the same hits fields, just check the first one

List<String> terms = phrase.getTerms();

if (!terms.isEmpty()) {

Term term = new Term(phrase.getTerms().get(0));

return term.accept(this);

}

return Collections.emptySet();

}

}