package com.twitter.search.earlybird.queryparser;

import java.util.List;

import com.google.common.base.Preconditions;

import com.google.common.collect.ImmutableList;

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

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

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

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

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

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

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

public class ProtectedOperatorQueryRewriter {

private static final String ERROR\_MESSAGE = "Positive 'protected' operator must be in the root"

+ " query node and the root query node must be a Conjunction.";

private static final Query EXCLUDE\_PROTECTED\_OPERATOR =

new SearchOperator(SearchOperator.Type.EXCLUDE, SearchOperatorConstants.PROTECTED);

/\*\*

\* Rewrite a query with positive 'protected' operator into an equivalent query without the positive

\* 'protected' operator. This method assumes the following preconditions hold:

\* 1. 'followedUserIds' is not empty

\* 2. the query's root node is of type Conjunction

\* 3. the query's root node is not negated

\* 4. there is one positive 'protected' operator in the root node

\* 5. there is only one 'protected' operator in the whole query

\*

\* Query with '[include protected]' operator is rewritten into a Disjunction of a query with

\* protected Tweets only and a query with public Tweets only.

\* For example,

\* Original query:

\* (\* "cat" [include protected])

\* with followedUserIds=[1, 7, 12] where 1 and 7 are protected users

\* Rewritten query:

\* (+

\* (\* "cat" [multi\_term\_disjunction from\_user\_id 1 7])

\* (\* "cat" [exclude protected])

\* )

\*

\* Query with '[filter protected]' operator is rewritten with multi\_term\_disjunction from\_user\_id

\* operator.

\* For example,

\* Original query:

\* (\* "cat" [filter protected])

\* with followedUserIds=[1, 7, 12] where 1 and 7 are protected users

\* Rewritten query:

\* (\* "cat" [multi\_term\_disjunction from\_user\_id 1 7])

\*/

public Query rewrite(Query parsedQuery, List<Long> followedUserIds, UserTable userTable) {

Preconditions.checkState(followedUserIds != null && !followedUserIds.isEmpty(),

"'followedUserIds' should not be empty when positive 'protected' operator exists.");

Preconditions.checkState(

parsedQuery.isTypeOf(com.twitter.search.queryparser.query.Query.QueryType.CONJUNCTION),

ERROR\_MESSAGE);

Conjunction parsedConjQuery = (Conjunction) parsedQuery;

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

int opIndex = findPositiveProtectedOperatorIndex(children);

Preconditions.checkState(opIndex >= 0, ERROR\_MESSAGE);

SearchOperator protectedOp = (SearchOperator) children.get(opIndex);

ImmutableList.Builder<Query> otherChildrenBuilder = ImmutableList.builder();

otherChildrenBuilder.addAll(children.subList(0, opIndex));

if (opIndex + 1 < children.size()) {

otherChildrenBuilder.addAll(children.subList(opIndex + 1, children.size()));

}

List<Query> otherChildren = otherChildrenBuilder.build();

List<Long> protectedUserIds = getProtectedUserIds(followedUserIds, userTable);

if (protectedOp.getOperatorType() == SearchOperator.Type.FILTER) {

if (protectedUserIds.isEmpty()) {

// match none query

return Disjunction.EMPTY\_DISJUNCTION;

} else {

return parsedConjQuery.newBuilder()

.setChildren(otherChildren)

.addChild(createFromUserIdMultiTermDisjunctionQuery(protectedUserIds))

.build();

}

} else {

// 'include' or negated 'exclude' operator

// negated 'exclude' is considered the same as 'include' to be consistent with the logic in

// EarlybirdLuceneQueryVisitor

if (protectedUserIds.isEmpty()) {

// return public only query

return parsedConjQuery.newBuilder()

.setChildren(otherChildren)

.addChild(EXCLUDE\_PROTECTED\_OPERATOR)

.build();

} else {

// build a disjunction of protected only query and public only query

Query protectedOnlyQuery = parsedConjQuery.newBuilder()

.setChildren(otherChildren)

.addChild(createFromUserIdMultiTermDisjunctionQuery(protectedUserIds))

.build();

Query publicOnlyQuery = parsedConjQuery.newBuilder()

.setChildren(otherChildren)

.addChild(EXCLUDE\_PROTECTED\_OPERATOR)

.build();

return new Disjunction(protectedOnlyQuery, publicOnlyQuery);

}

}

}

private Query createFromUserIdMultiTermDisjunctionQuery(List<Long> userIds) {

ImmutableList.Builder<String> operandsBuilder = ImmutableList.builder();

operandsBuilder

.add(EarlybirdFieldConstants.EarlybirdFieldConstant.FROM\_USER\_ID\_FIELD.getFieldName());

for (Long userId : userIds) {

operandsBuilder.add(userId.toString());

}

List<String> operands = operandsBuilder.build();

return new SearchOperator(SearchOperator.Type.MULTI\_TERM\_DISJUNCTION, operands);

}

private List<Long> getProtectedUserIds(List<Long> followedUserIds, UserTable userTable) {

ImmutableList.Builder<Long> protectedUserIds = ImmutableList.builder();

for (Long userId : followedUserIds) {

if (userTable.isSet(userId, UserTable.IS\_PROTECTED\_BIT)) {

protectedUserIds.add(userId);

}

}

return protectedUserIds.build();

}

private int findPositiveProtectedOperatorIndex(List<Query> children) {

for (int i = 0; i < children.size(); i++) {

Query child = children.get(i);

if (child instanceof SearchOperator) {

SearchOperator searchOp = (SearchOperator) child;

if (SearchOperatorConstants.PROTECTED.equals(searchOp.getOperand())

&& (isNegateExclude(searchOp) || isPositive(searchOp))) {

return i;

}

}

}

return -1;

}

private boolean isNegateExclude(SearchOperator searchOp) {

return searchOp.mustNotOccur()

&& searchOp.getOperatorType() == SearchOperator.Type.EXCLUDE;

}

private boolean isPositive(SearchOperator searchOp) {

return !searchOp.mustNotOccur()

&& (searchOp.getOperatorType() == SearchOperator.Type.INCLUDE

|| searchOp.getOperatorType() == SearchOperator.Type.FILTER);

}

}