package com.twitter.search.earlybird\_root.filters;

import java.util.Arrays;

import java.util.EnumSet;

import java.util.HashSet;

import java.util.Set;

import java.util.concurrent.ConcurrentHashMap;

import java.util.concurrent.ConcurrentMap;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.common.util.Clock;

import com.twitter.finagle.Service;

import com.twitter.finagle.SimpleFilter;

import com.twitter.search.common.clientstats.RequestCounters;

import com.twitter.search.common.clientstats.RequestCountersEventListener;

import com.twitter.search.earlybird.thrift.EarlybirdRequest;

import com.twitter.search.earlybird.thrift.EarlybirdResponse;

import com.twitter.search.earlybird\_root.common.EarlybirdRequestContext;

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

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

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

import com.twitter.search.queryparser.visitors.DetectVisitor;

import com.twitter.util.Future;

/\*\*

\* This filter exports RequestCounters stats for each unique combination of client\_id and

\* query\_operator. RequestCounters produce 19 stats for each prefix, and we have numerous

\* clients and operators, so this filter can produce a large number of stats. To keep the

\* number of exported stats reasonable we use an allow list of operators. The list currently

\* includes the geo operators while we monitor the impacts of realtime geo filtering. See

\* SEARCH-33699 for project details.

\*

\* To find the stats look for query\_client\_operator\_\* exported by archive roots.

\*

\*\*/

public class ClientIdQueryOperatorStatsFilter

extends SimpleFilter<EarlybirdRequestContext, EarlybirdResponse> {

private static final Logger LOG = LoggerFactory.getLogger(ClientIdQueryOperatorStatsFilter.class);

public static final String COUNTER\_PREFIX\_PATTERN = "query\_client\_operator\_%s\_%s";

private final Clock clock;

private final ConcurrentMap<String, RequestCounters> requestCountersByClientIdAndOperator =

new ConcurrentHashMap<>();

private final Set<SearchOperator.Type> operatorsToRecordStatsFor = new HashSet<>(Arrays.asList(

SearchOperator.Type.GEO\_BOUNDING\_BOX,

SearchOperator.Type.GEOCODE,

SearchOperator.Type.GEOLOCATION\_TYPE,

SearchOperator.Type.NEAR,

SearchOperator.Type.PLACE,

SearchOperator.Type.WITHIN));

public ClientIdQueryOperatorStatsFilter() {

this.clock = Clock.SYSTEM\_CLOCK;

}

@Override

public Future<EarlybirdResponse> apply(

EarlybirdRequestContext requestContext,

Service<EarlybirdRequestContext, EarlybirdResponse> service) {

EarlybirdRequest req = requestContext.getRequest();

Query parsedQuery = requestContext.getParsedQuery();

if (parsedQuery == null) {

return service.apply(requestContext);

}

Set<SearchOperator.Type> operators = getOperators(parsedQuery);

Future<EarlybirdResponse> response = service.apply(requestContext);

for (SearchOperator.Type operator : operators) {

RequestCounters clientOperatorCounters = getClientOperatorCounters(req.clientId, operator);

RequestCountersEventListener<EarlybirdResponse> clientOperatorCountersEventListener =

new RequestCountersEventListener<>(

clientOperatorCounters, clock, EarlybirdSuccessfulResponseHandler.INSTANCE);

response = response.addEventListener(clientOperatorCountersEventListener);

}

return response;

}

/\*\*

\* Gets or creates RequestCounters for the given clientId and operatorType

\*/

private RequestCounters getClientOperatorCounters(String clientId,

SearchOperator.Type operatorType) {

String counterPrefix = String.format(COUNTER\_PREFIX\_PATTERN, clientId, operatorType.toString());

RequestCounters clientCounters = requestCountersByClientIdAndOperator.get(counterPrefix);

if (clientCounters == null) {

clientCounters = new RequestCounters(counterPrefix);

RequestCounters existingCounters =

requestCountersByClientIdAndOperator.putIfAbsent(counterPrefix, clientCounters);

if (existingCounters != null) {

clientCounters = existingCounters;

}

}

return clientCounters;

}

/\*\*

\* Returns a set of the SearchOperator types that are:

\* 1) used by the query

\* 2) included in the allow list: operatorsToRecordStatsFor

\*/

private Set<SearchOperator.Type> getOperators(Query parsedQuery) {

final DetectVisitor detectVisitor = new DetectVisitor(false, SearchOperator.Type.values());

Set<SearchOperator.Type> detectedOperatorTypes = EnumSet.noneOf(SearchOperator.Type.class);

try {

parsedQuery.accept(detectVisitor);

} catch (QueryParserException e) {

LOG.error("Failed to detect SearchOperators in query: " + parsedQuery.toString());

return detectedOperatorTypes;

}

for (Query query : detectVisitor.getDetectedQueries()) {

// This detectVisitor only matches on SearchOperators.

SearchOperator operator = (SearchOperator) query;

SearchOperator.Type operatorType = operator.getOperatorType();

if (operatorsToRecordStatsFor.contains(operatorType)) {

detectedOperatorTypes.add(operatorType);

}

}

return detectedOperatorTypes;

}

}