package com.twitter.search.ingester.pipeline.util;

import java.util.Collection;

import java.util.Collections;

import java.util.HashSet;

import java.util.List;

import java.util.Map;

import java.util.Optional;

import java.util.Set;

import javax.annotation.Nullable;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Preconditions;

import com.google.common.collect.ImmutableList;

import com.google.common.collect.Lists;

import com.google.common.collect.Maps;

import org.apache.thrift.TBase;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.common\_internal.analytics.test\_user\_filter.TestUserFilter;

import com.twitter.common\_internal.text.version.PenguinVersion;

import com.twitter.metastore.client\_v2.MetastoreClient;

import com.twitter.metastore.data.MetastoreColumn;

import com.twitter.metastore.data.MetastoreException;

import com.twitter.metastore.data.MetastoreRow;

import com.twitter.metastore.data.MetastoreValue;

import com.twitter.search.common.metrics.RelevanceStats;

import com.twitter.search.common.metrics.SearchCounter;

import com.twitter.search.common.metrics.SearchRateCounter;

import com.twitter.search.common.metrics.SearchRequestStats;

import com.twitter.search.common.relevance.entities.TwitterMessage;

import com.twitter.search.common.relevance.features.RelevanceSignalConstants;

import com.twitter.search.ingester.model.IngesterTwitterMessage;

import com.twitter.service.metastore.gen.ResponseCode;

import com.twitter.service.metastore.gen.TweepCred;

import com.twitter.util.Function;

import com.twitter.util.Future;

public class UserPropertiesManager {

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

@VisibleForTesting

protected static final List<MetastoreColumn<? extends TBase<?, ?>>> COLUMNS =

ImmutableList.of(MetastoreColumn.TWEEPCRED); // contains tweepcred value

// same spam threshold that is use in tweeypie to spread user level spam to tweets, all tweets

// from user with spam score above such are marked so and removed from search results

@VisibleForTesting

public static final double SPAM\_SCORE\_THRESHOLD = 4.5;

@VisibleForTesting

static final SearchRequestStats MANHATTAN\_METASTORE\_STATS =

SearchRequestStats.export("manhattan\_metastore\_get", true);

private static final MetastoreGetColumnStats GET\_TWEEP\_CRED

= new MetastoreGetColumnStats("tweep\_cred");

@VisibleForTesting

static final SearchRateCounter MISSING\_REPUTATION\_COUNTER = RelevanceStats.exportRate(

"num\_missing\_reputation");

@VisibleForTesting

static final SearchRateCounter INVALID\_REPUTATION\_COUNTER = RelevanceStats.exportRate(

"num\_invalid\_reputation");

@VisibleForTesting

static final SearchRateCounter ACCEPTED\_REPUTATION\_COUNTER = RelevanceStats.exportRate(

"num\_accepted\_reputation");

@VisibleForTesting

static final SearchRateCounter SKIPPED\_REPUTATION\_CHECK\_COUNTER = RelevanceStats.exportRate(

"num\_skipped\_reputation\_check\_for\_test\_user");

@VisibleForTesting

static final SearchCounter DEFAULT\_REPUTATION\_COUNTER = SearchCounter.export(

"messages\_default\_reputation\_count");

@VisibleForTesting

static final SearchCounter MESSAGE\_FROM\_TEST\_USER =

SearchCounter.export("messages\_from\_test\_user");

// User level bits that are spread onto tweets

private static final SearchRateCounter IS\_USER\_NSFW\_COUNTER = RelevanceStats.exportRate(

"num\_is\_nsfw");

private static final SearchRateCounter IS\_USER\_SPAM\_COUNTER = RelevanceStats.exportRate(

"num\_is\_spam");

// count how many tweets has "possibly\_sensitive" set to true in the original json message

private static final SearchRateCounter IS\_SENSITIVE\_FROM\_JSON\_COUNTER = RelevanceStats.exportRate(

"num\_is\_sensitive\_in\_json");

private static final SearchCounter SENSITIVE\_BITS\_COUNTER =

SearchCounter.export("messages\_sensitive\_bits\_set\_count");

private final MetastoreClient metastoreClient;

private final UserPropertiesManager.MetastoreGetColumnStats tweepCredStats;

/\*\*

\* Stats for keeping track of multiGet requests to metastore for a specific data column.

\*/

@VisibleForTesting static class MetastoreGetColumnStats {

/\*\*

\* No data was returned from metastore for a specific user.

\*/

private final SearchCounter notReturned;

/\*\*

\* Metastore returned a successful OK response.

\*/

private final SearchCounter metastoreSuccess;

/\*\*

\* Metastore returned a NOT\_FOUND response for a user.

\*/

private final SearchCounter metastoreNotFound;

/\*\*

\* Metastore returned a BAD\_INPUT response for a user.

\*/

private final SearchCounter metastoreBadInput;

/\*\*

\* Metastore returned a TRANSIENT\_ERROR response for a user.

\*/

private final SearchCounter metastoreTransientError;

/\*\*

\* Metastore returned a PERMANENT\_ERROR response for a user.

\*/

private final SearchCounter metastorePermanentError;

/\*\*

\* Metastore returned an unknown response code for a user.

\*/

private final SearchCounter metastoreUnknownResponseCode;

/\*\*

\* Total number of users that we asked data for in metastore.

\*/

private final SearchCounter totalRequests;

@VisibleForTesting MetastoreGetColumnStats(String columnName) {

String prefix = "manhattan\_metastore\_get\_" + columnName;

notReturned = SearchCounter.export(prefix + "\_response\_not\_returned");

metastoreSuccess = SearchCounter.export(prefix + "\_response\_success");

metastoreNotFound = SearchCounter.export(prefix + "\_response\_not\_found");

metastoreBadInput = SearchCounter.export(prefix + "\_response\_bad\_input");

metastoreTransientError = SearchCounter.export(prefix + "\_response\_transient\_error");

metastorePermanentError = SearchCounter.export(prefix + "\_response\_permanent\_error");

metastoreUnknownResponseCode =

SearchCounter.export(prefix + "\_response\_unknown\_response\_code");

// Have a distinguishable prefix for the total requests stat so that we can use it to get

// a viz rate against wild-carded "prefix\_response\_\*" stats.

totalRequests = SearchCounter.export(prefix + "\_requests");

}

/\*\*

\* Tracks metastore get column stats for an individual user's response.

\* @param responseCode the response code received from metastore. Expected to be null if no

\* response came back at all.

\*/

private void trackMetastoreResponseCode(@Nullable ResponseCode responseCode) {

totalRequests.increment();

if (responseCode == null) {

notReturned.increment();

} else if (responseCode == ResponseCode.OK) {

metastoreSuccess.increment();

} else if (responseCode == ResponseCode.NOT\_FOUND) {

metastoreNotFound.increment();

} else if (responseCode == ResponseCode.BAD\_INPUT) {

metastoreBadInput.increment();

} else if (responseCode == ResponseCode.TRANSIENT\_ERROR) {

metastoreTransientError.increment();

} else if (responseCode == ResponseCode.PERMANENT\_ERROR) {

metastorePermanentError.increment();

} else {

metastoreUnknownResponseCode.increment();

}

}

@VisibleForTesting long getTotalRequests() {

return totalRequests.get();

}

@VisibleForTesting long getNotReturnedCount() {

return notReturned.get();

}

@VisibleForTesting long getMetastoreSuccessCount() {

return metastoreSuccess.get();

}

@VisibleForTesting long getMetastoreNotFoundCount() {

return metastoreNotFound.get();

}

@VisibleForTesting long getMetastoreBadInputCount() {

return metastoreBadInput.get();

}

@VisibleForTesting long getMetastoreTransientErrorCount() {

return metastoreTransientError.get();

}

@VisibleForTesting long getMetastorePermanentErrorCount() {

return metastorePermanentError.get();

}

@VisibleForTesting long getMetastoreUnknownResponseCodeCount() {

return metastoreUnknownResponseCode.get();

}

}

/\*\* Class that holds all user properties from Manhattan. \*/

@VisibleForTesting

protected static class ManhattanUserProperties {

private double spamScore = 0;

private float tweepcred = RelevanceSignalConstants.UNSET\_REPUTATION\_SENTINEL; // default

public ManhattanUserProperties setSpamScore(double newSpamScore) {

this.spamScore = newSpamScore;

return this;

}

public float getTweepcred() {

return tweepcred;

}

public ManhattanUserProperties setTweepcred(float newTweepcred) {

this.tweepcred = newTweepcred;

return this;

}

}

public UserPropertiesManager(MetastoreClient metastoreClient) {

this(metastoreClient, GET\_TWEEP\_CRED);

}

@VisibleForTesting

UserPropertiesManager(

MetastoreClient metastoreClient,

MetastoreGetColumnStats tweepCredStats) {

this.metastoreClient = metastoreClient;

this.tweepCredStats = tweepCredStats;

}

/\*\*

\* Gets user properties including TWEEPCRED, SpamScore values/flags from metastore for the

\* given userids.

\*

\* @param userIds the list of users for which to get the properties.

\* @return mapping from userId to UserProperties. If a user's twepcred score is not present in the

\* metastore, of if there was a problem retrieving it, that user's score will not be set in the

\* returned map.

\*/

@VisibleForTesting

Future<Map<Long, ManhattanUserProperties>> getManhattanUserProperties(final List<Long> userIds) {

Preconditions.checkArgument(userIds != null);

if (metastoreClient == null || userIds.isEmpty()) {

return Future.value(Collections.emptyMap());

}

final long start = System.currentTimeMillis();

return metastoreClient.multiGet(userIds, COLUMNS)

.map(new Function<Map<Long, MetastoreRow>, Map<Long, ManhattanUserProperties>>() {

@Override

public Map<Long, ManhattanUserProperties> apply(Map<Long, MetastoreRow> response) {

long latencyMs = System.currentTimeMillis() - start;

Map<Long, ManhattanUserProperties> resultMap =

Maps.newHashMapWithExpectedSize(userIds.size());

for (Long userId : userIds) {

MetastoreRow row = response.get(userId);

processTweepCredColumn(userId, row, resultMap);

}

MANHATTAN\_METASTORE\_STATS.requestComplete(latencyMs, resultMap.size(), true);

return resultMap;

}

})

.handle(new Function<Throwable, Map<Long, ManhattanUserProperties>>() {

@Override

public Map<Long, ManhattanUserProperties> apply(Throwable t) {

long latencyMs = System.currentTimeMillis() - start;

LOG.error("Exception talking to metastore after " + latencyMs + " ms.", t);

MANHATTAN\_METASTORE\_STATS.requestComplete(latencyMs, 0, false);

return Collections.emptyMap();

}

});

}

/\*\*

\* Process the TweepCred column data returned from metastore, takes TweepCred, fills in the

\* the resultMap as appropriate.

\*/

private void processTweepCredColumn(

Long userId,

MetastoreRow metastoreRow,

Map<Long, ManhattanUserProperties> resultMap) {

MetastoreValue<TweepCred> tweepCredValue =

metastoreRow == null ? null : metastoreRow.getValue(MetastoreColumn.TWEEPCRED);

ResponseCode responseCode = tweepCredValue == null ? null : tweepCredValue.getResponseCode();

tweepCredStats.trackMetastoreResponseCode(responseCode);

if (responseCode == ResponseCode.OK) {

try {

TweepCred tweepCred = tweepCredValue.getValue();

if (tweepCred != null && tweepCred.isSetScore()) {

ManhattanUserProperties manhattanUserProperties =

getOrCreateManhattanUserProperties(userId, resultMap);

manhattanUserProperties.setTweepcred(tweepCred.getScore());

}

} catch (MetastoreException e) {

// guaranteed not to be thrown if ResponseCode.OK

LOG.warn("Unexpected MetastoreException parsing userinfo column!", e);

}

}

}

private static ManhattanUserProperties getOrCreateManhattanUserProperties(

Long userId, Map<Long, ManhattanUserProperties> resultMap) {

ManhattanUserProperties manhattanUserProperties = resultMap.get(userId);

if (manhattanUserProperties == null) {

manhattanUserProperties = new ManhattanUserProperties();

resultMap.put(userId, manhattanUserProperties);

}

return manhattanUserProperties;

}

/\*\*

\* Populates the user properties from the given batch.

\*/

public Future<Collection<IngesterTwitterMessage>> populateUserProperties(

Collection<IngesterTwitterMessage> batch) {

Set<Long> userIds = new HashSet<>();

for (IngesterTwitterMessage message : batch) {

if ((message.getUserReputation() == IngesterTwitterMessage.DOUBLE\_FIELD\_NOT\_PRESENT)

&& !message.isDeleted()) {

Optional<Long> userId = message.getFromUserTwitterId();

if (userId.isPresent()) {

userIds.add(userId.get());

} else {

LOG.error("No user id present for tweet {}", message.getId());

}

}

}

List<Long> uniqIds = Lists.newArrayList(userIds);

Collections.sort(uniqIds); // for testing predictability

Future<Map<Long, ManhattanUserProperties>> manhattanUserPropertiesMap =

getManhattanUserProperties(uniqIds);

return manhattanUserPropertiesMap.map(Function.func(map -> {

for (IngesterTwitterMessage message : batch) {

if (((message.getUserReputation() != IngesterTwitterMessage.DOUBLE\_FIELD\_NOT\_PRESENT)

&& RelevanceSignalConstants.isValidUserReputation(

(int) Math.floor(message.getUserReputation())))

|| message.isDeleted()) {

continue;

}

Optional<Long> optionalUserId = message.getFromUserTwitterId();

if (optionalUserId.isPresent()) {

long userId = optionalUserId.get();

ManhattanUserProperties manhattanUserProperties = map.get(userId);

final boolean isTestUser = TestUserFilter.isTestUserId(userId);

if (isTestUser) {

MESSAGE\_FROM\_TEST\_USER.increment();

}

// legacy setting of tweepcred

setTweepCred(isTestUser, manhattanUserProperties, message);

// set additional fields

if (setSensitiveBits(manhattanUserProperties, message)) {

SENSITIVE\_BITS\_COUNTER.increment();

}

}

}

return batch;

}));

}

// good old tweepcred

private void setTweepCred(

boolean isTestUser,

ManhattanUserProperties manhattanUserProperties,

TwitterMessage message) {

float score = RelevanceSignalConstants.UNSET\_REPUTATION\_SENTINEL;

if (manhattanUserProperties == null) {

if (isTestUser) {

SKIPPED\_REPUTATION\_CHECK\_COUNTER.increment();

} else {

MISSING\_REPUTATION\_COUNTER.increment();

DEFAULT\_REPUTATION\_COUNTER.increment();

}

} else if (!RelevanceSignalConstants.isValidUserReputation(

(int) Math.floor(manhattanUserProperties.tweepcred))) {

if (!isTestUser) {

INVALID\_REPUTATION\_COUNTER.increment();

DEFAULT\_REPUTATION\_COUNTER.increment();

}

} else {

score = manhattanUserProperties.tweepcred;

ACCEPTED\_REPUTATION\_COUNTER.increment();

}

message.setUserReputation(score);

}

// Sets sensitive content, nsfw, and spam flags in TwitterMessage, further

// sets the following bits in encoded features:

// EarlybirdFeatureConfiguration.IS\_SENSITIVE\_FLAG

// EarlybirdFeatureConfiguration.IS\_USER\_NSFW\_FLAG

// EarlybirdFeatureConfiguration.IS\_USER\_SPAM\_FLAG

private boolean setSensitiveBits(

ManhattanUserProperties manhattanUserProperties,

TwitterMessage message) {

if (manhattanUserProperties == null) {

return false;

}

final boolean isUserSpam = manhattanUserProperties.spamScore > SPAM\_SCORE\_THRESHOLD;

// SEARCH-17413: Compute the field with gizmoduck data.

final boolean isUserNSFW = false;

final boolean anySensitiveBitSet = isUserSpam || isUserNSFW;

if (message.isSensitiveContent()) {

// original json has possibly\_sensitive = true, count it

IS\_SENSITIVE\_FROM\_JSON\_COUNTER.increment();

}

if (isUserNSFW) {

// set EarlybirdFeatureConfiguration.IS\_USER\_NSFW\_FLAG

for (PenguinVersion penguinVersion : message.getSupportedPenguinVersions()) {

message.getTweetUserFeatures(penguinVersion).setNsfw(isUserNSFW);

}

IS\_USER\_NSFW\_COUNTER.increment();

}

if (isUserSpam) {

// set EarlybirdFeatureConfiguration.IS\_USER\_SPAM\_FLAG

for (PenguinVersion penguinVersion : message.getSupportedPenguinVersions()) {

message.getTweetUserFeatures(penguinVersion).setSpam(isUserSpam);

}

IS\_USER\_SPAM\_COUNTER.increment();

}

// if any of the sensitive bits are set, we return true

return anySensitiveBitSet;

}

}