package com.twitter.follow\_recommendations.common.candidate\_sources.recent\_engagement

import com.google.inject.Inject

import com.google.inject.Singleton

import com.twitter.dds.jobs.repeated\_profile\_visits.thriftscala.ProfileVisitorInfo

import com.twitter.experiments.general\_metrics.thriftscala.IdType

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.follow\_recommendations.common.clients.real\_time\_real\_graph.Engagement

import com.twitter.follow\_recommendations.common.clients.real\_time\_real\_graph.RealTimeRealGraphClient

import com.twitter.follow\_recommendations.common.models.CandidateUser

import com.twitter.timelines.configapi.HasParams

import com.twitter.timelines.configapi.Params

import com.twitter.hermit.model.Algorithm

import com.twitter.inject.Logging

import com.twitter.product\_mixer.core.functional\_component.candidate\_source.CandidateSource

import com.twitter.product\_mixer.core.model.common.identifier.CandidateSourceIdentifier

import com.twitter.product\_mixer.core.model.marshalling.request.HasClientContext

import com.twitter.stitch.Stitch

import com.twitter.strato.generated.client.rux.RepeatedProfileVisitsAggregateClientColumn

@Singleton

class RepeatedProfileVisitsSource @Inject() (

repeatedProfileVisitsAggregateClientColumn: RepeatedProfileVisitsAggregateClientColumn,

realTimeRealGraphClient: RealTimeRealGraphClient,

statsReceiver: StatsReceiver)

extends CandidateSource[HasParams with HasClientContext, CandidateUser]

with Logging {

val identifier: CandidateSourceIdentifier =

RepeatedProfileVisitsSource.Identifier

val sourceStatsReceiver = statsReceiver.scope("repeated\_profile\_visits\_source")

val offlineFetchErrorCounter = sourceStatsReceiver.counter("offline\_fetch\_error")

val offlineFetchSuccessCounter = sourceStatsReceiver.counter("offline\_fetch\_success")

val onlineFetchErrorCounter = sourceStatsReceiver.counter("online\_fetch\_error")

val onlineFetchSuccessCounter = sourceStatsReceiver.counter("online\_fetch\_success")

val noRepeatedProfileVisitsAboveBucketingThresholdCounter =

sourceStatsReceiver.counter("no\_repeated\_profile\_visits\_above\_bucketing\_threshold")

val hasRepeatedProfileVisitsAboveBucketingThresholdCounter =

sourceStatsReceiver.counter("has\_repeated\_profile\_visits\_above\_bucketing\_threshold")

val noRepeatedProfileVisitsAboveRecommendationsThresholdCounter =

sourceStatsReceiver.counter("no\_repeated\_profile\_visits\_above\_recommendations\_threshold")

val hasRepeatedProfileVisitsAboveRecommendationsThresholdCounter =

sourceStatsReceiver.counter("has\_repeated\_profile\_visits\_above\_recommendations\_threshold")

val includeCandidatesCounter = sourceStatsReceiver.counter("include\_candidates")

val noIncludeCandidatesCounter = sourceStatsReceiver.counter("no\_include\_candidates")

// Returns visited user -> visit count, via off dataset.

def applyWithOfflineDataset(targetUserId: Long): Stitch[Map[Long, Int]] = {

repeatedProfileVisitsAggregateClientColumn.fetcher

.fetch(ProfileVisitorInfo(id = targetUserId, idType = IdType.User)).map(\_.v)

.handle {

case e: Throwable =>

logger.error("Strato fetch for RepeatedProfileVisitsAggregateClientColumn failed: " + e)

offlineFetchErrorCounter.incr()

None

}.onSuccess { result =>

offlineFetchSuccessCounter.incr()

}.map { resultOption =>

resultOption

.flatMap { result =>

result.profileVisitSet.map { profileVisitSet =>

profileVisitSet

.filter(profileVisit => profileVisit.totalTargetVisitsInLast14Days.getOrElse(0) > 0)

.filter(profileVisit => !profileVisit.doesSourceIdFollowTargetId.getOrElse(false))

.flatMap { profileVisit =>

(profileVisit.targetId, profileVisit.totalTargetVisitsInLast14Days) match {

case (Some(targetId), Some(totalVisitsInLast14Days)) =>

Some(targetId -> totalVisitsInLast14Days)

case \_ => None

}

}.toMap[Long, Int]

}

}.getOrElse(Map.empty)

}

}

// Returns visited user -> visit count, via online dataset.

def applyWithOnlineData(targetUserId: Long): Stitch[Map[Long, Int]] = {

val visitedUserToEngagementsStitch: Stitch[Map[Long, Seq[Engagement]]] =

realTimeRealGraphClient.getRecentProfileViewEngagements(targetUserId)

visitedUserToEngagementsStitch

.onFailure { f =>

onlineFetchErrorCounter.incr()

}.onSuccess { result =>

onlineFetchSuccessCounter.incr()

}.map { visitedUserToEngagements =>

visitedUserToEngagements

.mapValues(engagements => engagements.size)

}

}

def getRepeatedVisitedAccounts(params: Params, targetUserId: Long): Stitch[Map[Long, Int]] = {

var results: Stitch[Map[Long, Int]] = Stitch.value(Map.empty)

if (params.getBoolean(RepeatedProfileVisitsParams.UseOnlineDataset)) {

results = applyWithOnlineData(targetUserId)

} else {

results = applyWithOfflineDataset(targetUserId)

}

// Only keep users that had non-zero engagement counts.

results.map(\_.filter(input => input.\_2 > 0))

}

def getRecommendations(params: Params, userId: Long): Stitch[Seq[CandidateUser]] = {

val recommendationThreshold = params.getInt(RepeatedProfileVisitsParams.RecommendationThreshold)

val bucketingThreshold = params.getInt(RepeatedProfileVisitsParams.BucketingThreshold)

// Get the list of repeatedly visited profilts. Only keep accounts with >= bucketingThreshold visits.

val repeatedVisitedAccountsStitch: Stitch[Map[Long, Int]] =

getRepeatedVisitedAccounts(params, userId).map(\_.filter(kv => kv.\_2 >= bucketingThreshold))

repeatedVisitedAccountsStitch.map { candidates =>

// Now check if we should includeCandidates (e.g. whether user is in control bucket or treatment buckets).

if (candidates.isEmpty) {

// User has not visited any accounts above bucketing threshold. We will not bucket user into experiment. Just

// don't return no candidates.

noRepeatedProfileVisitsAboveBucketingThresholdCounter.incr()

Seq.empty

} else {

hasRepeatedProfileVisitsAboveBucketingThresholdCounter.incr()

if (!params.getBoolean(RepeatedProfileVisitsParams.IncludeCandidates)) {

// User has reached bucketing criteria. We check whether to include candidates (e.g. checking which bucket

// the user is in for the experiment). In this case the user is in a bucket to not include any candidates.

noIncludeCandidatesCounter.incr()

Seq.empty

} else {

includeCandidatesCounter.incr()

// We should include candidates. Include any candidates above recommendation thresholds.

val outputCandidatesSeq = candidates

.filter(kv => kv.\_2 >= recommendationThreshold).map { kv =>

val user = kv.\_1

val visitCount = kv.\_2

CandidateUser(user, Some(visitCount.toDouble))

.withCandidateSource(RepeatedProfileVisitsSource.Identifier)

}.toSeq

if (outputCandidatesSeq.isEmpty) {

noRepeatedProfileVisitsAboveRecommendationsThresholdCounter.incr()

} else {

hasRepeatedProfileVisitsAboveRecommendationsThresholdCounter.incr()

}

outputCandidatesSeq

}

}

}

}

override def apply(request: HasParams with HasClientContext): Stitch[Seq[CandidateUser]] = {

request.getOptionalUserId

.map { userId =>

getRecommendations(request.params, userId)

}.getOrElse(Stitch.Nil)

}

}

object RepeatedProfileVisitsSource {

val Identifier = CandidateSourceIdentifier(Algorithm.RepeatedProfileVisits.toString)

}