package com.twitter.follow\_recommendations.common.transforms.modify\_social\_proof

import com.twitter.conversions.DurationOps.\_

import com.twitter.decider.Decider

import com.twitter.decider.RandomRecipient

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.finagle.util.DefaultTimer

import com.twitter.follow\_recommendations.common.base.GatedTransform

import com.twitter.follow\_recommendations.common.clients.graph\_feature\_service.GraphFeatureServiceClient

import com.twitter.follow\_recommendations.common.clients.socialgraph.SocialGraphClient

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

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

import com.twitter.follow\_recommendations.configapi.deciders.DeciderKey

import com.twitter.graph\_feature\_service.thriftscala.EdgeType

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

import com.twitter.snowflake.id.SnowflakeId

import com.twitter.stitch.Stitch

import com.twitter.timelines.configapi.HasParams

import com.twitter.util.logging.Logging

import com.twitter.util.Duration

import com.twitter.util.Time

import javax.inject.Inject

import javax.inject.Singleton

object ModifySocialProof {

val GfsLagDuration: Duration = 14.days

val GfsIntersectionIds: Int = 3

val SgsIntersectionIds: Int = 10

val LeftEdgeTypes: Set[EdgeType] = Set(EdgeType.Following)

val RightEdgeTypes: Set[EdgeType] = Set(EdgeType.FollowedBy)

/\*\*

\* Given the intersection ID's for a particular candidate, update the candidate's social proof

\* @param candidate candidate object

\* @param followProof follow proof to be added (includes id's and count)

\* @param stats stats for tracking

\* @return updated candidate object

\*/

def addIntersectionIdsToCandidate(

candidate: CandidateUser,

followProof: FollowProof,

stats: StatsReceiver

): CandidateUser = {

// create updated set of social proof

val updatedFollowedByOpt = candidate.followedBy match {

case Some(existingFollowedBy) => Some((followProof.followedBy ++ existingFollowedBy).distinct)

case None if followProof.followedBy.nonEmpty => Some(followProof.followedBy.distinct)

case \_ => None

}

val updatedFollowProof = updatedFollowedByOpt.map { updatedFollowedBy =>

val updatedCount = followProof.numIds.max(updatedFollowedBy.size)

// track stats

val numSocialProofAdded = updatedFollowedBy.size - candidate.followedBy.size

addCandidatesWithSocialContextCountStat(stats, numSocialProofAdded)

FollowProof(updatedFollowedBy, updatedCount)

}

candidate.setFollowProof(updatedFollowProof)

}

private def addCandidatesWithSocialContextCountStat(

statsReceiver: StatsReceiver,

count: Int

): Unit = {

if (count > 3) {

statsReceiver.counter("4\_and\_more").incr()

} else {

statsReceiver.counter(count.toString).incr()

}

}

}

/\*\*

\* This class makes a request to gfs/sgs for hydrating additional social proof on each of the

\* provided candidates.

\*/

@Singleton

class ModifySocialProof @Inject() (

gfsClient: GraphFeatureServiceClient,

socialGraphClient: SocialGraphClient,

statsReceiver: StatsReceiver,

decider: Decider = Decider.True)

extends Logging {

import ModifySocialProof.\_

private val stats = statsReceiver.scope(this.getClass.getSimpleName)

private val addedStats = stats.scope("num\_social\_proof\_added\_per\_candidate")

private val gfsStats = stats.scope("graph\_feature\_service")

private val sgsStats = stats.scope("social\_graph\_service")

private val previousProofEmptyCounter = stats.counter("previous\_proof\_empty")

private val emptyFollowProofCounter = stats.counter("empty\_followed\_proof")

/\*\*

\* For each candidate provided, we get the intersectionIds between the user and the candidate,

\* appending the unique results to the social proof (followedBy field) if not already previously

\* seen we query GFS for all users, except for cases specified via the mustCallSgs field or for

\* very new users, who would not have any data in GFS, due to the lag duration of the service's

\* processing. this is determined by GfsLagDuration

\* @param userId id of the target user whom we provide recommendations for

\* @param candidates list of candidates

\* @param intersectionIdsNum if provided, determines the maximum number of accounts we want to be hydrated for social proof

\* @param mustCallSgs Determines if we should query SGS regardless of user age or not.

\* @return list of candidates updated with additional social proof

\*/

def hydrateSocialProof(

userId: Long,

candidates: Seq[CandidateUser],

intersectionIdsNum: Option[Int] = None,

mustCallSgs: Boolean = false,

callSgsCachedColumn: Boolean = false,

gfsLagDuration: Duration = GfsLagDuration,

gfsIntersectionIds: Int = GfsIntersectionIds,

sgsIntersectionIds: Int = SgsIntersectionIds,

): Stitch[Seq[CandidateUser]] = {

addCandidatesWithSocialContextCountStat(

stats.scope("social\_context\_count\_before\_hydration"),

candidates.count(\_.followedBy.isDefined)

)

val candidateIds = candidates.map(\_.id)

val userAgeOpt = SnowflakeId.timeFromIdOpt(userId).map(Time.now - \_)

// this decider gate is used to determine what % of requests is allowed to call

// Graph Feature Service. this is useful for ramping down requests to Graph Feature Service

// when necessary

val deciderKey: String = DeciderKey.EnableGraphFeatureServiceRequests.toString

val enableGfsRequests: Boolean = decider.isAvailable(deciderKey, Some(RandomRecipient))

// if new query sgs

val (candidateToIntersectionIdsMapFut, isGfs) =

if (!enableGfsRequests || mustCallSgs || userAgeOpt.exists(\_ < gfsLagDuration)) {

(

if (callSgsCachedColumn)

socialGraphClient.getIntersectionsFromCachedColumn(

userId,

candidateIds,

intersectionIdsNum.getOrElse(sgsIntersectionIds)

)

else

socialGraphClient.getIntersections(

userId,

candidateIds,

intersectionIdsNum.getOrElse(sgsIntersectionIds)),

false)

} else {

(

gfsClient.getIntersections(

userId,

candidateIds,

intersectionIdsNum.getOrElse(gfsIntersectionIds)),

true)

}

val finalCandidates = candidateToIntersectionIdsMapFut

.map { candidateToIntersectionIdsMap =>

{

previousProofEmptyCounter.incr(candidates.count(\_.followedBy.exists(\_.isEmpty)))

candidates.map { candidate =>

addIntersectionIdsToCandidate(

candidate,

candidateToIntersectionIdsMap.getOrElse(candidate.id, FollowProof(Seq.empty, 0)),

addedStats)

}

}

}

.within(250.milliseconds)(DefaultTimer)

.rescue {

case e: Exception =>

error(e.getMessage)

if (isGfs) {

gfsStats.scope("rescued").counter(e.getClass.getSimpleName).incr()

} else {

sgsStats.scope("rescued").counter(e.getClass.getSimpleName).incr()

}

Stitch.value(candidates)

}

finalCandidates.onSuccess { candidatesSeq =>

emptyFollowProofCounter.incr(candidatesSeq.count(\_.followedBy.exists(\_.isEmpty)))

addCandidatesWithSocialContextCountStat(

stats.scope("social\_context\_count\_after\_hydration"),

candidatesSeq.count(\_.followedBy.isDefined)

)

}

}

}

/\*\*

\* This transform uses ModifySocialProof (which makes a request to gfs/sgs) for hydrating additional

\* social proof on each of the provided candidates.

\*/

@Singleton

class ModifySocialProofTransform @Inject() (modifySocialProof: ModifySocialProof)

extends GatedTransform[HasClientContext with HasParams, CandidateUser]

with Logging {

override def transform(

target: HasClientContext with HasParams,

candidates: Seq[CandidateUser]

): Stitch[Seq[CandidateUser]] =

target.getOptionalUserId

.map(modifySocialProof.hydrateSocialProof(\_, candidates)).getOrElse(Stitch.value(candidates))

}