package com.twitter.frigate.pushservice.refresh\_handler

import com.twitter.channels.common.thriftscala.ApiList

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.base.\_

import com.twitter.frigate.common.rec\_types.RecTypes.isInNetworkTweetType

import com.twitter.frigate.pushservice.model.PushTypes.PushCandidate

import com.twitter.frigate.pushservice.model.PushTypes.RawCandidate

import com.twitter.frigate.pushservice.model.TrendTweetPushCandidate

import com.twitter.frigate.pushservice.ml.PushMLModelScorer

import com.twitter.frigate.pushservice.model.candidate.CopyIds

import com.twitter.frigate.pushservice.refresh\_handler.cross.CandidateCopyExpansion

import com.twitter.frigate.pushservice.util.CandidateHydrationUtil.\_

import com.twitter.frigate.pushservice.util.MrUserStateUtil

import com.twitter.frigate.pushservice.util.RelationshipUtil

import com.twitter.gizmoduck.thriftscala.User

import com.twitter.hermit.predicate.socialgraph.RelationEdge

import com.twitter.storehaus.ReadableStore

import com.twitter.util.Future

case class PushCandidateHydrator(

socialGraphServiceProcessStore: ReadableStore[RelationEdge, Boolean],

safeUserStore: ReadableStore[Long, User],

apiListStore: ReadableStore[Long, ApiList],

candidateCopyCross: CandidateCopyExpansion

)(

implicit statsReceiver: StatsReceiver,

implicit val weightedOpenOrNtabClickModelScorer: PushMLModelScorer) {

lazy val candidateWithCopyNumStat = statsReceiver.stat("candidate\_with\_copy\_num")

lazy val hydratedCandidateStat = statsReceiver.scope("hydrated\_candidates")

lazy val mrUserStateStat = statsReceiver.scope("mr\_user\_state")

lazy val queryStep = statsReceiver.scope("query\_step")

lazy val relationEdgeWithoutDuplicateInQueryStep =

queryStep.counter("number\_of\_relationEdge\_without\_duplicate\_in\_query\_step")

lazy val relationEdgeWithoutDuplicateInQueryStepDistribution =

queryStep.stat("number\_of\_relationEdge\_without\_duplicate\_in\_query\_step\_distribution")

case class Entities(

users: Set[Long] = Set.empty[Long],

relationshipEdges: Set[RelationEdge] = Set.empty[RelationEdge]) {

def merge(otherEntities: Entities): Entities = {

this.copy(

users = this.users ++ otherEntities.users,

relationshipEdges =

this.relationshipEdges ++ otherEntities.relationshipEdges

)

}

}

case class EntitiesMap(

userMap: Map[Long, User] = Map.empty[Long, User],

relationshipMap: Map[RelationEdge, Boolean] = Map.empty[RelationEdge, Boolean])

private def updateCandidateAndCrtStats(

candidate: RawCandidate,

candidateType: String,

numEntities: Int = 1

): Unit = {

statsReceiver

.scope(candidateType).scope(candidate.commonRecType.name).stat(

"totalEntitiesPerCandidateTypePerCrt").add(numEntities)

statsReceiver.scope(candidateType).stat("totalEntitiesPerCandidateType").add(numEntities)

}

private def collectEntities(

candidateDetailsSeq: Seq[CandidateDetails[RawCandidate]]

): Entities = {

candidateDetailsSeq

.map { candidateDetails =>

val pushCandidate = candidateDetails.candidate

val userEntities = pushCandidate match {

case tweetWithSocialContext: RawCandidate with TweetWithSocialContextTraits =>

val authorIdOpt = getAuthorIdFromTweetCandidate(tweetWithSocialContext)

val scUserIds = tweetWithSocialContext.socialContextUserIds.toSet

updateCandidateAndCrtStats(pushCandidate, "tweetWithSocialContext", scUserIds.size + 1)

Entities(users = scUserIds ++ authorIdOpt.toSet)

case \_ => Entities()

}

val relationEntities = {

if (isInNetworkTweetType(pushCandidate.commonRecType)) {

Entities(

relationshipEdges =

RelationshipUtil.getPreCandidateRelationshipsForInNetworkTweets(pushCandidate).toSet

)

} else Entities()

}

userEntities.merge(relationEntities)

}

.foldLeft(Entities()) { (e1, e2) => e1.merge(e2) }

}

/\*\*

\* This method calls Gizmoduck and Social Graph Service, keep the results in EntitiesMap

\* and passed onto the update candidate phase in the hydration step

\*

\* @param entities contains all userIds and relationEdges for all candidates

\* @return EntitiesMap contains userMap and relationshipMap

\*/

private def queryEntities(entities: Entities): Future[EntitiesMap] = {

relationEdgeWithoutDuplicateInQueryStep.incr(entities.relationshipEdges.size)

relationEdgeWithoutDuplicateInQueryStepDistribution.add(entities.relationshipEdges.size)

val relationshipMapFuture = Future

.collect(socialGraphServiceProcessStore.multiGet(entities.relationshipEdges))

.map { resultMap =>

resultMap.collect {

case (relationshipEdge, Some(res)) => relationshipEdge -> res

case (relationshipEdge, None) => relationshipEdge -> false

}

}

val userMapFuture = Future

.collect(safeUserStore.multiGet(entities.users))

.map { userMap =>

userMap.collect {

case (userId, Some(user)) =>

userId -> user

}

}

Future.join(userMapFuture, relationshipMapFuture).map {

case (uMap, rMap) => EntitiesMap(userMap = uMap, relationshipMap = rMap)

}

}

/\*\*

\* @param candidateDetails: recommendation candidates for a user

\* @return sequence of candidates tagged with push and ntab copy id

\*/

private def expandCandidatesWithCopy(

candidateDetails: Seq[CandidateDetails[RawCandidate]]

): Future[Seq[(CandidateDetails[RawCandidate], CopyIds)]] = {

candidateCopyCross.expandCandidatesWithCopyId(candidateDetails)

}

def updateCandidates(

candidateDetailsWithCopies: Seq[(CandidateDetails[RawCandidate], CopyIds)],

entitiesMaps: EntitiesMap

): Seq[CandidateDetails[PushCandidate]] = {

candidateDetailsWithCopies.map {

case (candidateDetail, copyIds) =>

val pushCandidate = candidateDetail.candidate

val userMap = entitiesMaps.userMap

val relationshipMap = entitiesMaps.relationshipMap

val hydratedCandidate = pushCandidate match {

case f1TweetCandidate: F1FirstDegree =>

getHydratedCandidateForF1FirstDegreeTweet(

f1TweetCandidate,

userMap,

relationshipMap,

copyIds)

case tweetRetweet: TweetRetweetCandidate =>

getHydratedCandidateForTweetRetweet(tweetRetweet, userMap, copyIds)

case tweetFavorite: TweetFavoriteCandidate =>

getHydratedCandidateForTweetFavorite(tweetFavorite, userMap, copyIds)

case tripTweetCandidate: OutOfNetworkTweetCandidate with TripCandidate =>

getHydratedCandidateForTripTweetCandidate(tripTweetCandidate, userMap, copyIds)

case outOfNetworkTweetCandidate: OutOfNetworkTweetCandidate with TopicCandidate =>

getHydratedCandidateForOutOfNetworkTweetCandidate(

outOfNetworkTweetCandidate,

userMap,

copyIds)

case topicProofTweetCandidate: TopicProofTweetCandidate =>

getHydratedTopicProofTweetCandidate(topicProofTweetCandidate, userMap, copyIds)

case subscribedSearchTweetCandidate: SubscribedSearchTweetCandidate =>

getHydratedSubscribedSearchTweetCandidate(

subscribedSearchTweetCandidate,

userMap,

copyIds)

case listRecommendation: ListPushCandidate =>

getHydratedListCandidate(apiListStore, listRecommendation, copyIds)

case discoverTwitterCandidate: DiscoverTwitterCandidate =>

getHydratedCandidateForDiscoverTwitterCandidate(discoverTwitterCandidate, copyIds)

case topTweetImpressionsCandidate: TopTweetImpressionsCandidate =>

getHydratedCandidateForTopTweetImpressionsCandidate(

topTweetImpressionsCandidate,

copyIds)

case trendTweetCandidate: TrendTweetCandidate =>

new TrendTweetPushCandidate(

trendTweetCandidate,

trendTweetCandidate.authorId.flatMap(userMap.get),

copyIds)

case unknownCandidate =>

throw new IllegalArgumentException(

s"Incorrect candidate for hydration: ${unknownCandidate.commonRecType}")

}

CandidateDetails(

hydratedCandidate,

source = candidateDetail.source

)

}

}

def apply(

candidateDetails: Seq[CandidateDetails[RawCandidate]]

): Future[Seq[CandidateDetails[PushCandidate]]] = {

val isLoggedOutRequest =

candidateDetails.headOption.exists(\_.candidate.target.isLoggedOutUser)

if (!isLoggedOutRequest) {

candidateDetails.headOption.map { cd =>

MrUserStateUtil.updateMrUserStateStats(cd.candidate.target)(mrUserStateStat)

}

}

expandCandidatesWithCopy(candidateDetails).flatMap { candidateDetailsWithCopy =>

candidateWithCopyNumStat.add(candidateDetailsWithCopy.size)

val entities = collectEntities(candidateDetailsWithCopy.map(\_.\_1))

queryEntities(entities).flatMap { entitiesMap =>

val updatedCandidates = updateCandidates(candidateDetailsWithCopy, entitiesMap)

updatedCandidates.foreach { cand =>

hydratedCandidateStat.counter(cand.candidate.commonRecType.name).incr()

}

Future.value(updatedCandidates)

}

}

}

}