package com.twitter.frigate.pushservice.adaptor

import com.twitter.conversions.DurationOps.\_

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.base.CandidateSource

import com.twitter.frigate.common.base.CandidateSourceEligible

import com.twitter.frigate.common.base.TopTweetImpressionsCandidate

import com.twitter.frigate.common.store.RecentTweetsQuery

import com.twitter.frigate.common.util.SnowflakeUtils

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

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

import com.twitter.frigate.pushservice.params.{PushFeatureSwitchParams => FS}

import com.twitter.frigate.pushservice.store.TweetImpressionsStore

import com.twitter.frigate.pushservice.util.PushDeviceUtil

import com.twitter.stitch.tweetypie.TweetyPie.TweetyPieResult

import com.twitter.storehaus.FutureOps

import com.twitter.storehaus.ReadableStore

import com.twitter.util.Future

case class TweetImpressionsCandidate(

tweetId: Long,

tweetyPieResultOpt: Option[TweetyPieResult],

impressionsCountOpt: Option[Long])

case class TopTweetImpressionsCandidateAdaptor(

recentTweetsFromTflockStore: ReadableStore[RecentTweetsQuery, Seq[Seq[Long]]],

tweetyPieStore: ReadableStore[Long, TweetyPieResult],

tweetyPieStoreNoVF: ReadableStore[Long, TweetyPieResult],

tweetImpressionsStore: TweetImpressionsStore,

globalStats: StatsReceiver)

extends CandidateSource[Target, RawCandidate]

with CandidateSourceEligible[Target, RawCandidate] {

private val stats = globalStats.scope("TopTweetImpressionsAdaptor")

private val tweetImpressionsCandsStat = stats.stat("top\_tweet\_impressions\_cands\_dist")

private val eligibleUsersCounter = stats.counter("eligible\_users")

private val noneligibleUsersCounter = stats.counter("noneligible\_users")

private val meetsMinTweetsRequiredCounter = stats.counter("meets\_min\_tweets\_required")

private val belowMinTweetsRequiredCounter = stats.counter("below\_min\_tweets\_required")

private val aboveMaxInboundFavoritesCounter = stats.counter("above\_max\_inbound\_favorites")

private val meetsImpressionsRequiredCounter = stats.counter("meets\_impressions\_required")

private val belowImpressionsRequiredCounter = stats.counter("below\_impressions\_required")

private val meetsFavoritesThresholdCounter = stats.counter("meets\_favorites\_threshold")

private val aboveFavoritesThresholdCounter = stats.counter("above\_favorites\_threshold")

private val emptyImpressionsMapCounter = stats.counter("empty\_impressions\_map")

private val tflockResultsStat = stats.stat("tflock", "results")

private val emptyTflockResult = stats.counter("tflock", "empty\_result")

private val nonEmptyTflockResult = stats.counter("tflock", "non\_empty\_result")

private val originalTweetsStat = stats.stat("tweets", "original\_tweets")

private val retweetsStat = stats.stat("tweets", "retweets")

private val allRetweetsOnlyCounter = stats.counter("tweets", "all\_retweets\_only")

private val allOriginalTweetsOnlyCounter = stats.counter("tweets", "all\_original\_tweets\_only")

private val emptyTweetypieMap = stats.counter("", "empty\_tweetypie\_map")

private val emptyTweetyPieResult = stats.stat("", "empty\_tweetypie\_result")

private val allEmptyTweetypieResults = stats.counter("", "all\_empty\_tweetypie\_results")

private val eligibleUsersAfterImpressionsFilter =

stats.counter("eligible\_users\_after\_impressions\_filter")

private val eligibleUsersAfterFavoritesFilter =

stats.counter("eligible\_users\_after\_favorites\_filter")

private val eligibleUsersWithEligibleTweets =

stats.counter("eligible\_users\_with\_eligible\_tweets")

private val eligibleTweetCands = stats.stat("eligible\_tweet\_cands")

private val getCandsRequestCounter =

stats.counter("top\_tweet\_impressions\_get\_request")

override val name: String = this.getClass.getSimpleName

override def get(inputTarget: Target): Future[Option[Seq[RawCandidate]]] = {

getCandsRequestCounter.incr()

val eligibleCandidatesFut = getTweetImpressionsCandidates(inputTarget)

eligibleCandidatesFut.map { eligibleCandidates =>

if (eligibleCandidates.nonEmpty) {

eligibleUsersWithEligibleTweets.incr()

eligibleTweetCands.add(eligibleCandidates.size)

val candidate = getMostImpressionsTweet(eligibleCandidates)

Some(

Seq(

generateTopTweetImpressionsCandidate(

inputTarget,

candidate.tweetId,

candidate.tweetyPieResultOpt,

candidate.impressionsCountOpt.getOrElse(0L))))

} else None

}

}

private def getTweetImpressionsCandidates(

inputTarget: Target

): Future[Seq[TweetImpressionsCandidate]] = {

val originalTweets = getRecentOriginalTweetsForUser(inputTarget)

originalTweets.flatMap { tweetyPieResultsMap =>

val numDaysSearchForOriginalTweets =

inputTarget.params(FS.TopTweetImpressionsOriginalTweetsNumDaysSearch)

val moreRecentTweetIds =

getMoreRecentTweetIds(tweetyPieResultsMap.keySet.toSeq, numDaysSearchForOriginalTweets)

val isEligible = isEligibleUser(inputTarget, tweetyPieResultsMap, moreRecentTweetIds)

if (isEligible) filterByEligibility(inputTarget, tweetyPieResultsMap, moreRecentTweetIds)

else Future.Nil

}

}

private def getRecentOriginalTweetsForUser(

targetUser: Target

): Future[Map[Long, TweetyPieResult]] = {

val tweetyPieResultsMapFut = getTflockStoreResults(targetUser).flatMap { recentTweetIds =>

FutureOps.mapCollect((targetUser.params(FS.EnableVFInTweetypie) match {

case true => tweetyPieStore

case false => tweetyPieStoreNoVF

}).multiGet(recentTweetIds.toSet))

}

tweetyPieResultsMapFut.map { tweetyPieResultsMap =>

if (tweetyPieResultsMap.isEmpty) {

emptyTweetypieMap.incr()

Map.empty

} else removeRetweets(tweetyPieResultsMap)

}

}

private def getTflockStoreResults(targetUser: Target): Future[Seq[Long]] = {

val maxResults = targetUser.params(FS.TopTweetImpressionsRecentTweetsByAuthorStoreMaxResults)

val maxAge = targetUser.params(FS.TopTweetImpressionsTotalFavoritesLimitNumDaysSearch)

val recentTweetsQuery =

RecentTweetsQuery(

userIds = Seq(targetUser.targetId),

maxResults = maxResults,

maxAge = maxAge.days

)

recentTweetsFromTflockStore

.get(recentTweetsQuery).map {

case Some(tweetIdsAll) =>

val tweetIds = tweetIdsAll.headOption.getOrElse(Seq.empty)

val numTweets = tweetIds.size

if (numTweets > 0) {

tflockResultsStat.add(numTweets)

nonEmptyTflockResult.incr()

} else emptyTflockResult.incr()

tweetIds

case \_ => Nil

}

}

private def removeRetweets(

tweetyPieResultsMap: Map[Long, Option[TweetyPieResult]]

): Map[Long, TweetyPieResult] = {

val nonEmptyTweetyPieResults: Map[Long, TweetyPieResult] = tweetyPieResultsMap.collect {

case (key, Some(value)) => (key, value)

}

emptyTweetyPieResult.add(tweetyPieResultsMap.size - nonEmptyTweetyPieResults.size)

if (nonEmptyTweetyPieResults.nonEmpty) {

val originalTweets = nonEmptyTweetyPieResults.filter {

case (\_, tweetyPieResult) =>

tweetyPieResult.sourceTweet.isEmpty

}

val numOriginalTweets = originalTweets.size

val numRetweets = nonEmptyTweetyPieResults.size - originalTweets.size

originalTweetsStat.add(numOriginalTweets)

retweetsStat.add(numRetweets)

if (numRetweets == 0) allOriginalTweetsOnlyCounter.incr()

if (numOriginalTweets == 0) allRetweetsOnlyCounter.incr()

originalTweets

} else {

allEmptyTweetypieResults.incr()

Map.empty

}

}

private def getMoreRecentTweetIds(

tweetIds: Seq[Long],

numDays: Int

): Seq[Long] = {

tweetIds.filter { tweetId =>

SnowflakeUtils.isRecent(tweetId, numDays.days)

}

}

private def isEligibleUser(

inputTarget: Target,

tweetyPieResults: Map[Long, TweetyPieResult],

recentTweetIds: Seq[Long]

): Boolean = {

val minNumTweets = inputTarget.params(FS.TopTweetImpressionsMinNumOriginalTweets)

lazy val totalFavoritesLimit =

inputTarget.params(FS.TopTweetImpressionsTotalInboundFavoritesLimit)

if (recentTweetIds.size >= minNumTweets) {

meetsMinTweetsRequiredCounter.incr()

val isUnderLimit = isUnderTotalInboundFavoritesLimit(tweetyPieResults, totalFavoritesLimit)

if (isUnderLimit) eligibleUsersCounter.incr()

else {

aboveMaxInboundFavoritesCounter.incr()

noneligibleUsersCounter.incr()

}

isUnderLimit

} else {

belowMinTweetsRequiredCounter.incr()

noneligibleUsersCounter.incr()

false

}

}

private def getFavoriteCounts(

tweetyPieResult: TweetyPieResult

): Long = tweetyPieResult.tweet.counts.flatMap(\_.favoriteCount).getOrElse(0L)

private def isUnderTotalInboundFavoritesLimit(

tweetyPieResults: Map[Long, TweetyPieResult],

totalFavoritesLimit: Long

): Boolean = {

val favoritesIterator = tweetyPieResults.valuesIterator.map(getFavoriteCounts)

val totalInboundFavorites = favoritesIterator.sum

totalInboundFavorites <= totalFavoritesLimit

}

def filterByEligibility(

inputTarget: Target,

tweetyPieResults: Map[Long, TweetyPieResult],

tweetIds: Seq[Long]

): Future[Seq[TweetImpressionsCandidate]] = {

lazy val minNumImpressions: Long = inputTarget.params(FS.TopTweetImpressionsMinRequired)

lazy val maxNumLikes: Long = inputTarget.params(FS.TopTweetImpressionsMaxFavoritesPerTweet)

for {

filteredImpressionsMap <- getFilteredImpressionsMap(tweetIds, minNumImpressions)

tweetIdsFilteredByFavorites <-

getTweetIdsFilteredByFavorites(filteredImpressionsMap.keySet, tweetyPieResults, maxNumLikes)

} yield {

if (filteredImpressionsMap.nonEmpty) eligibleUsersAfterImpressionsFilter.incr()

if (tweetIdsFilteredByFavorites.nonEmpty) eligibleUsersAfterFavoritesFilter.incr()

val candidates = tweetIdsFilteredByFavorites.map { tweetId =>

TweetImpressionsCandidate(

tweetId,

tweetyPieResults.get(tweetId),

filteredImpressionsMap.get(tweetId))

}

tweetImpressionsCandsStat.add(candidates.length)

candidates

}

}

private def getFilteredImpressionsMap(

tweetIds: Seq[Long],

minNumImpressions: Long

): Future[Map[Long, Long]] = {

getImpressionsCounts(tweetIds).map { impressionsMap =>

if (impressionsMap.isEmpty) emptyImpressionsMapCounter.incr()

impressionsMap.filter {

case (\_, numImpressions) =>

val isValid = numImpressions >= minNumImpressions

if (isValid) {

meetsImpressionsRequiredCounter.incr()

} else {

belowImpressionsRequiredCounter.incr()

}

isValid

}

}

}

private def getTweetIdsFilteredByFavorites(

filteredTweetIds: Set[Long],

tweetyPieResults: Map[Long, TweetyPieResult],

maxNumLikes: Long

): Future[Seq[Long]] = {

val filteredByFavoritesTweetIds = filteredTweetIds.filter { tweetId =>

val tweetyPieResultOpt = tweetyPieResults.get(tweetId)

val isValid = tweetyPieResultOpt.exists { tweetyPieResult =>

getFavoriteCounts(tweetyPieResult) <= maxNumLikes

}

if (isValid) meetsFavoritesThresholdCounter.incr()

else aboveFavoritesThresholdCounter.incr()

isValid

}

Future(filteredByFavoritesTweetIds.toSeq)

}

private def getMostImpressionsTweet(

filteredResults: Seq[TweetImpressionsCandidate]

): TweetImpressionsCandidate = {

val maxImpressions: Long = filteredResults.map {

\_.impressionsCountOpt.getOrElse(0L)

}.max

val mostImpressionsCandidates: Seq[TweetImpressionsCandidate] =

filteredResults.filter(\_.impressionsCountOpt.getOrElse(0L) == maxImpressions)

mostImpressionsCandidates.maxBy(\_.tweetId)

}

private def getImpressionsCounts(

tweetIds: Seq[Long]

): Future[Map[Long, Long]] = {

val impressionCountMap = tweetIds.map { tweetId =>

tweetId -> tweetImpressionsStore

.getCounts(tweetId).map(\_.getOrElse(0L))

}.toMap

Future.collect(impressionCountMap)

}

private def generateTopTweetImpressionsCandidate(

inputTarget: Target,

\_tweetId: Long,

result: Option[TweetyPieResult],

\_impressionsCount: Long

): RawCandidate = {

new RawCandidate with TopTweetImpressionsCandidate {

override val target: Target = inputTarget

override val tweetId: Long = \_tweetId

override val tweetyPieResult: Option[TweetyPieResult] = result

override val impressionsCount: Long = \_impressionsCount

}

}

override def isCandidateSourceAvailable(target: Target): Future[Boolean] = {

val enabledTopTweetImpressionsNotification =

target.params(FS.EnableTopTweetImpressionsNotification)

PushDeviceUtil

.isRecommendationsEligible(target).map(\_ && enabledTopTweetImpressionsNotification)

}

}