package com.twitter.recos.user\_tweet\_entity\_graph

import java.util.Random

import com.twitter.concurrent.AsyncQueue

import com.twitter.conversions.DurationOps.\_

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.graphjet.algorithms.\_

import com.twitter.graphjet.algorithms.filters.\_

import com.twitter.graphjet.algorithms.counting.TopSecondDegreeByCountResponse

import com.twitter.graphjet.algorithms.counting.tweet.TopSecondDegreeByCountForTweet

import com.twitter.graphjet.algorithms.counting.tweet.TopSecondDegreeByCountRequestForTweet

import com.twitter.graphjet.bipartite.NodeMetadataLeftIndexedMultiSegmentBipartiteGraph

import com.twitter.logging.Logger

import com.twitter.recos.graph\_common.FinagleStatsReceiverWrapper

import com.twitter.recos.model.SalsaQueryRunner.SalsaRunnerConfig

import com.twitter.recos.recos\_common.thriftscala.SocialProofType

import com.twitter.recos.user\_tweet\_entity\_graph.thriftscala.RecommendTweetEntityRequest

import com.twitter.recos.user\_tweet\_entity\_graph.thriftscala.TweetEntityDisplayLocation

import com.twitter.recos.user\_tweet\_entity\_graph.thriftscala.TweetType

import com.twitter.recos.util.Stats.trackBlockStats

import com.twitter.util.Future

import com.twitter.util.JavaTimer

import com.twitter.util.Try

import it.unimi.dsi.fastutil.longs.Long2DoubleOpenHashMap

import it.unimi.dsi.fastutil.longs.LongOpenHashSet

import scala.collection.JavaConverters.\_

import com.twitter.graphjet.algorithms.RecommendationType

import com.twitter.recos.user\_tweet\_entity\_graph.thriftscala.{

RecommendationType => ThriftRecommendationType

}

import scala.collection.Map

import scala.collection.Set

object TweetRecommendationsRunner {

private val DefaultTweetTypes: Seq[TweetType] =

Seq(TweetType.Regular, TweetType.Summary, TweetType.Photo, TweetType.Player)

private val DefaultF1ExactSocialProofSize = 1

private val DefaultRareTweetRecencyMillis: Long = 7.days.inMillis

/\*\*

\* Map valid social proof types specified by clients to an array of bytes. If clients do not

\* specify any social proof type unions in thrift, it will return an empty set by default.

\*/

private def getSocialProofTypeUnions(

socialProofTypeUnions: Option[Set[Seq[SocialProofType]]]

): Set[Array[Byte]] = {

socialProofTypeUnions

.map {

\_.map {

\_.map {

\_.getValue.toByte

}.toArray

}

}

.getOrElse(Set.empty)

}

private def getRecommendationTypes(

recommendationTypes: Seq[ThriftRecommendationType]

): Set[RecommendationType] = {

recommendationTypes.flatMap {

\_ match {

case ThriftRecommendationType.Tweet => Some(RecommendationType.TWEET)

case ThriftRecommendationType.Hashtag => Some(RecommendationType.HASHTAG)

case ThriftRecommendationType.Url => Some(RecommendationType.URL)

case \_ =>

throw new Exception("Unmatched Recommendation Type in getRecommendationTypes")

}

}.toSet

}

private def convertThriftEnumsToJavaEnums(

maxResults: Option[Map[ThriftRecommendationType, Int]]

): Map[RecommendationType, Integer] = {

maxResults

.map {

\_.flatMap {

\_ match {

case (ThriftRecommendationType.Tweet, v) => Some((RecommendationType.TWEET, v: Integer))

case (ThriftRecommendationType.Hashtag, v) =>

Some((RecommendationType.HASHTAG, v: Integer))

case (ThriftRecommendationType.Url, v) => Some((RecommendationType.URL, v: Integer))

case \_ =>

throw new Exception("Unmatched Recommendation Type in convertThriftEnumsToJavaEnums")

}

}

}

.getOrElse(Map.empty)

}

}

/\*\*

\* The MagicRecsRunner creates a queue of reader threads, MagicRecs, and each one reads from the

\* graph and computes recommendations.

\*/

class TweetRecommendationsRunner(

bipartiteGraph: NodeMetadataLeftIndexedMultiSegmentBipartiteGraph,

salsaRunnerConfig: SalsaRunnerConfig,

statsReceiverWrapper: FinagleStatsReceiverWrapper) {

import TweetRecommendationsRunner.\_

private val log: Logger = Logger()

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

private val magicRecsFailureCounter = stats.counter("failure")

private val pollCounter = stats.counter("poll")

private val pollTimeoutCounter = stats.counter("pollTimeout")

private val offerCounter = stats.counter("offer")

private val pollLatencyStat = stats.stat("pollLatency")

private val magicRecsQueue = new AsyncQueue[TopSecondDegreeByCountForTweet]

(0 until salsaRunnerConfig.numSalsaRunners).foreach { \_ =>

magicRecsQueue.offer(

new TopSecondDegreeByCountForTweet(

bipartiteGraph,

salsaRunnerConfig.expectedNodesToHitInSalsa,

statsReceiverWrapper.scope(this.getClass.getSimpleName)

)

)

}

private implicit val timer: JavaTimer = new JavaTimer(true)

private def getBaseFilters(

staleTweetDuration: Long,

tweetTypes: Seq[TweetType]

) = {

List(

// Keep RecentTweetFilter first since it's the cheapest

new RecentTweetFilter(staleTweetDuration, statsReceiverWrapper),

new TweetCardFilter(

tweetTypes.contains(TweetType.Regular),

tweetTypes.contains(TweetType.Summary),

tweetTypes.contains(TweetType.Photo),

tweetTypes.contains(TweetType.Player),

false, // no promoted tweets

statsReceiverWrapper

),

new DirectInteractionsFilter(bipartiteGraph, statsReceiverWrapper),

new RequestedSetFilter(statsReceiverWrapper),

new SocialProofTypesFilter(statsReceiverWrapper)

)

}

/\*\*

\* Helper method to interpret the output of MagicRecs graph

\*

\* @param magicRecsResponse is the response from running MagicRecs

\* @return a sequence of candidate ids, with score and list of social proofs

\*/

private def transformMagicRecsResponse(

magicRecsResponse: Option[TopSecondDegreeByCountResponse]

): Seq[RecommendationInfo] = {

val responses = magicRecsResponse match {

case Some(response) => response.getRankedRecommendations.asScala.toSeq

case \_ => Nil

}

responses

}

/\*\*

\* Helper function to determine different post-process filtering logic in GraphJet,

\* based on display locations

\*/

private def getFiltersByDisplayLocations(

displayLocation: TweetEntityDisplayLocation,

whitelistAuthors: LongOpenHashSet,

blacklistAuthors: LongOpenHashSet,

validSocialProofs: Array[Byte]

) = {

displayLocation match {

case TweetEntityDisplayLocation.MagicRecsF1 =>

Seq(

new ANDFilters(

List[ResultFilter](

new TweetAuthorFilter(

bipartiteGraph,

whitelistAuthors,

new LongOpenHashSet(),

statsReceiverWrapper),

new ExactUserSocialProofSizeFilter(

DefaultF1ExactSocialProofSize,

validSocialProofs,

statsReceiverWrapper

)

).asJava,

statsReceiverWrapper

),

// Blacklist filter must be applied separately from F1's AND filter chain

new TweetAuthorFilter(

bipartiteGraph,

new LongOpenHashSet(),

blacklistAuthors,

statsReceiverWrapper)

)

case TweetEntityDisplayLocation.MagicRecsRareTweet =>

Seq(

new TweetAuthorFilter(

bipartiteGraph,

whitelistAuthors,

blacklistAuthors,

statsReceiverWrapper),

new RecentEdgeMetadataFilter(

DefaultRareTweetRecencyMillis,

UserTweetEdgeTypeMask.Tweet.id.toByte,

statsReceiverWrapper

)

)

case \_ =>

Seq(

new TweetAuthorFilter(

bipartiteGraph,

whitelistAuthors,

blacklistAuthors,

statsReceiverWrapper))

}

}

/\*\*

\* Helper method to run salsa computation and convert the results to Option

\*

\* @param magicRecs is magicRecs reader on bipartite graph

\* @param magicRecsRequest is the magicRecs request

\* @return is an option of MagicRecsResponse

\*/

private def getMagicRecsResponse(

magicRecs: TopSecondDegreeByCountForTweet,

magicRecsRequest: TopSecondDegreeByCountRequestForTweet

)(

implicit statsReceiver: StatsReceiver

): Option[TopSecondDegreeByCountResponse] = {

trackBlockStats(stats) {

val random = new Random()

// compute recs -- need to catch and print exceptions here otherwise they are swallowed

val magicRecsAttempt =

Try(magicRecs.computeRecommendations(magicRecsRequest, random)).onFailure { e =>

magicRecsFailureCounter.incr()

log.error(e, "MagicRecs computation failed")

}

magicRecsAttempt.toOption

}

}

private def getMagicRecsRequest(

request: RecommendTweetEntityRequest

): TopSecondDegreeByCountRequestForTweet = {

val requesterId = request.requesterId

val leftSeedNodes = new Long2DoubleOpenHashMap(

request.seedsWithWeights.keys.toArray,

request.seedsWithWeights.values.toArray

)

val tweetsToExcludeArray = new LongOpenHashSet(request.excludedTweetIds.getOrElse(Nil).toArray)

val staleTweetDuration = request.maxTweetAgeInMillis.getOrElse(RecosConfig.maxTweetAgeInMillis)

val staleEngagementDuration =

request.maxEngagementAgeInMillis.getOrElse(RecosConfig.maxEngagementAgeInMillis)

val tweetTypes = request.tweetTypes.getOrElse(DefaultTweetTypes)

val tweetAuthors = new LongOpenHashSet(request.tweetAuthors.getOrElse(Nil).toArray)

val excludedTweetAuthors = new LongOpenHashSet(

request.excludedTweetAuthors.getOrElse(Nil).toArray)

val validSocialProofs =

UserTweetEdgeTypeMask.getUserTweetGraphSocialProofTypes(request.socialProofTypes)

val resultFilterChain = new ResultFilterChain(

(

getBaseFilters(staleTweetDuration, tweetTypes) ++

getFiltersByDisplayLocations(

displayLocation = request.displayLocation,

whitelistAuthors = tweetAuthors,

blacklistAuthors = excludedTweetAuthors,

validSocialProofs = validSocialProofs

)

).asJava

)

new TopSecondDegreeByCountRequestForTweet(

requesterId,

leftSeedNodes,

tweetsToExcludeArray,

getRecommendationTypes(request.recommendationTypes).asJava,

convertThriftEnumsToJavaEnums(request.maxResultsByType).asJava,

UserTweetEdgeTypeMask.SIZE,

request.maxUserSocialProofSize.getOrElse(RecosConfig.maxUserSocialProofSize),

request.maxTweetSocialProofSize.getOrElse(RecosConfig.maxTweetSocialProofSize),

convertThriftEnumsToJavaEnums(request.minUserSocialProofSizes).asJava,

validSocialProofs,

staleTweetDuration,

staleEngagementDuration,

resultFilterChain,

getSocialProofTypeUnions(request.socialProofTypeUnions).asJava

)

}

def apply(request: RecommendTweetEntityRequest): Future[Seq[RecommendationInfo]] = {

pollCounter.incr()

val t0 = System.currentTimeMillis

magicRecsQueue.poll().map { magicRecs =>

val pollTime = System.currentTimeMillis - t0

pollLatencyStat.add(pollTime)

val magicRecsResponse = Try {

if (pollTime < salsaRunnerConfig.timeoutSalsaRunner) {

val magicRecsRequest = getMagicRecsRequest(request)

transformMagicRecsResponse(

getMagicRecsResponse(magicRecs, magicRecsRequest)(statsReceiverWrapper.statsReceiver)

)

} else {

// if we did not get a magicRecs in time, then fail fast here and immediately put it back

log.warning("magicRecsQueue polling timeout")

pollTimeoutCounter.incr()

throw new RuntimeException("magicRecs poll timeout")

Nil

}

} ensure {

magicRecsQueue.offer(magicRecs)

offerCounter.incr()

}

magicRecsResponse.toOption getOrElse Nil

}

}

}