package com.twitter.graph\_feature\_service.worker.handlers

import com.twitter.finagle.stats.{Stat, StatsReceiver}

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

WorkerIntersectionRequest,

WorkerIntersectionResponse,

WorkerIntersectionValue

}

import com.twitter.graph\_feature\_service.util.{FeatureTypesCalculator, IntersectionValueCalculator}

import com.twitter.graph\_feature\_service.util.IntersectionValueCalculator.\_

import com.twitter.graph\_feature\_service.worker.util.GraphContainer

import com.twitter.servo.request.RequestHandler

import com.twitter.util.Future

import java.nio.ByteBuffer

import javax.inject.{Inject, Singleton}

@Singleton

class WorkerGetIntersectionHandler @Inject() (

graphContainer: GraphContainer,

statsReceiver: StatsReceiver)

extends RequestHandler[WorkerIntersectionRequest, WorkerIntersectionResponse] {

import WorkerGetIntersectionHandler.\_

private val stats: StatsReceiver = statsReceiver.scope("srv/get\_intersection")

private val numCandidatesCount = stats.counter("total\_num\_candidates")

private val toPartialGraphQueryStat = stats.stat("to\_partial\_graph\_query\_latency")

private val fromPartialGraphQueryStat = stats.stat("from\_partial\_graph\_query\_latency")

private val intersectionCalculationStat = stats.stat("computation\_latency")

override def apply(request: WorkerIntersectionRequest): Future[WorkerIntersectionResponse] = {

numCandidatesCount.incr(request.candidateUserIds.length)

val userId = request.userId

// NOTE: do not change the order of candidates

val candidateIds = request.candidateUserIds

// NOTE: do not change the order of features

val featureTypes =

FeatureTypesCalculator.getFeatureTypes(request.presetFeatureTypes, request.featureTypes)

val leftEdges = featureTypes.map(\_.leftEdgeType).distinct

val rightEdges = featureTypes.map(\_.rightEdgeType).distinct

val rightEdgeMap = Stat.time(toPartialGraphQueryStat) {

rightEdges.map { rightEdge =>

val map = graphContainer.toPartialMap.get(rightEdge) match {

case Some(graph) =>

candidateIds.flatMap { candidateId =>

graph.apply(candidateId).map(candidateId -> \_)

}.toMap

case None =>

Map.empty[Long, ByteBuffer]

}

rightEdge -> map

}.toMap

}

val leftEdgeMap = Stat.time(fromPartialGraphQueryStat) {

leftEdges.flatMap { leftEdge =>

graphContainer.toPartialMap.get(leftEdge).flatMap(\_.apply(userId)).map(leftEdge -> \_)

}.toMap

}

val res = Stat.time(intersectionCalculationStat) {

WorkerIntersectionResponse(

// NOTE that candidate ordering is important

candidateIds.map { candidateId =>

// NOTE that the featureTypes ordering is important

featureTypes.map {

featureType =>

val leftNeighborsOpt = leftEdgeMap.get(featureType.leftEdgeType)

val rightNeighborsOpt =

rightEdgeMap.get(featureType.rightEdgeType).flatMap(\_.get(candidateId))

if (leftNeighborsOpt.isEmpty && rightNeighborsOpt.isEmpty) {

EmptyWorkerIntersectionValue

} else if (rightNeighborsOpt.isEmpty) {

EmptyWorkerIntersectionValue.copy(

leftNodeDegree = computeArraySize(leftNeighborsOpt.get)

)

} else if (leftNeighborsOpt.isEmpty) {

EmptyWorkerIntersectionValue.copy(

rightNodeDegree = computeArraySize(rightNeighborsOpt.get)

)

} else {

IntersectionValueCalculator(

leftNeighborsOpt.get,

rightNeighborsOpt.get,

request.intersectionIdLimit)

}

}

}

)

}

Future.value(res)

}

}

object WorkerGetIntersectionHandler {

val EmptyWorkerIntersectionValue: WorkerIntersectionValue = WorkerIntersectionValue(0, 0, 0, Nil)

}