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

import com.twitter.finagle.stats.Stat

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.graph\_feature\_service.server.handlers.ServerGetIntersectionHandler.GetIntersectionRequest

import com.twitter.graph\_feature\_service.server.stores.FeatureTypesEncoder

import com.twitter.graph\_feature\_service.server.stores.GetIntersectionStore.GetIntersectionQuery

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

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

import com.twitter.graph\_feature\_service.util.FeatureTypesCalculator

import com.twitter.servo.request.RequestHandler

import com.twitter.storehaus.ReadableStore

import com.twitter.util.Future

import com.twitter.util.Memoize

import javax.inject.Inject

import javax.inject.Named

import javax.inject.Singleton

@Singleton

class ServerGetIntersectionHandler @Inject() (

@Named("ReadThroughGetIntersectionStore")

readThroughStore: ReadableStore[GetIntersectionQuery, CachedIntersectionResult],

@Named("BypassCacheGetIntersectionStore")

readOnlyStore: ReadableStore[GetIntersectionQuery, CachedIntersectionResult]

)(

implicit statsReceiver: StatsReceiver)

extends RequestHandler[GetIntersectionRequest, GfsIntersectionResponse] {

import ServerGetIntersectionHandler.\_

// TODO: Track all the stats based on PresetFeatureType and update the dashboard

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

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

private val numCandidatesStat = stats.stat("num\_candidates")

private val numFeaturesStat = stats.stat("num\_features")

private val userEmptyCount = stats.counter("user\_empty\_count")

private val candidateEmptyRateStat = stats.stat("candidate\_empty\_rate")

private val candidateNumEmptyStat = stats.stat("candidate\_num\_empty")

private val missedRateStat = stats.stat("miss\_rate")

private val numMissedStat = stats.stat("num\_missed")

// Assume the order from HTL doesn't change. Only log the HTL query now.

private val featureStatMap = FeatureTypesCalculator.presetFeatureTypes.map { feature =>

val featureString = s"${feature.leftEdgeType.name}\_${feature.rightEdgeType.name}"

feature -> Array(

stats.counter(s"feature\_type\_${featureString}\_total"),

stats.counter(s"feature\_type\_${featureString}\_count\_zero"),

stats.counter(s"feature\_type\_${featureString}\_left\_zero"),

stats.counter(s"feature\_type\_${featureString}\_right\_zero")

)

}.toMap

private val sourceCandidateNumStats = Memoize[PresetFeatureTypes, Stat] { presetFeature =>

stats.stat(s"source\_candidate\_num\_${presetFeature.name}")

}

override def apply(request: GetIntersectionRequest): Future[GfsIntersectionResponse] = {

val featureTypes = request.calculatedFeatureTypes

val numCandidates = request.candidateUserIds.length

val numFeatures = featureTypes.length

numCandidatesCount.incr(numCandidates)

numCandidatesStat.add(numCandidates)

numFeaturesStat.add(numFeatures)

sourceCandidateNumStats(request.presetFeatureTypes).add(numCandidates)

// Note: do not change the orders of features and candidates.

val candidateIds = request.candidateUserIds

if (featureTypes.isEmpty || candidateIds.isEmpty) {

Future.value(DefaultGfsIntersectionResponse)

} else {

Future

.collect {

val getIntersectionStore = if (request.cacheable) readThroughStore else readOnlyStore

getIntersectionStore.multiGet(GetIntersectionQuery.buildQueries(request))

}.map { responses =>

val results = responses.collect {

case (query, Some(result)) =>

query.candidateId -> GfsIntersectionResult(

query.candidateId,

query.calculatedFeatureTypes.zip(result.values).map {

case (featureType, value) =>

IntersectionValue(

featureType,

Some(value.count),

if (value.intersectionIds.isEmpty) None else Some(value.intersectionIds),

Some(value.leftNodeDegree),

Some(value.rightNodeDegree)

)

}

)

}

// Keep the response order same as input

val processedResults = candidateIds.map { candidateId =>

results.getOrElse(candidateId, GfsIntersectionResult(candidateId, List.empty))

}

val candidateEmptyNum =

processedResults.count(

\_.intersectionValues.exists(value => isZero(value.rightNodeDegree)))

val numMissed = processedResults.count(\_.intersectionValues.size != numFeatures)

if (processedResults.exists(

\_.intersectionValues.forall(value => isZero(value.leftNodeDegree)))) {

userEmptyCount.incr()

}

candidateNumEmptyStat.add(candidateEmptyNum)

candidateEmptyRateStat.add(candidateEmptyNum.toFloat / numCandidates)

numMissedStat.add(numMissed)

missedRateStat.add(numMissed.toFloat / numCandidates)

processedResults.foreach { result =>

result.intersectionValues.zip(featureTypes).foreach {

case (value, featureType) =>

featureStatMap.get(featureType).foreach { statsArray =>

statsArray(TotalIndex).incr()

if (isZero(value.count)) {

statsArray(CountIndex).incr()

}

if (isZero(value.leftNodeDegree)) {

statsArray(LeftIndex).incr()

}

if (isZero(value.rightNodeDegree)) {

statsArray(RightIndex).incr()

}

}

}

}

GfsIntersectionResponse(processedResults)

}

}

}

}

private[graph\_feature\_service] object ServerGetIntersectionHandler {

case class GetIntersectionRequest(

userId: Long,

candidateUserIds: Seq[Long],

featureTypes: Seq[FeatureType],

presetFeatureTypes: PresetFeatureTypes,

intersectionIdLimit: Option[Int],

cacheable: Boolean) {

lazy val calculatedFeatureTypes: Seq[FeatureType] =

FeatureTypesCalculator.getFeatureTypes(presetFeatureTypes, featureTypes)

lazy val calculatedFeatureTypesString: String =

FeatureTypesEncoder(calculatedFeatureTypes)

}

object GetIntersectionRequest {

def fromGfsIntersectionRequest(

request: GfsIntersectionRequest,

cacheable: Boolean

): GetIntersectionRequest = {

GetIntersectionRequest(

request.userId,

request.candidateUserIds,

request.featureTypes,

PresetFeatureTypes.Empty,

request.intersectionIdLimit,

cacheable)

}

def fromGfsPresetIntersectionRequest(

request: GfsPresetIntersectionRequest,

cacheable: Boolean

): GetIntersectionRequest = {

GetIntersectionRequest(

request.userId,

request.candidateUserIds,

List.empty,

request.presetFeatureTypes,

request.intersectionIdLimit,

cacheable)

}

}

private val DefaultGfsIntersectionResponse = GfsIntersectionResponse()

private val TotalIndex = 0

private val CountIndex = 1

private val LeftIndex = 2

private val RightIndex = 3

def isZero(opt: Option[Int]): Boolean = {

!opt.exists(\_ != 0)

}

}