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

import com.twitter.finagle.RequestTimeoutException

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

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

import com.twitter.graph\_feature\_service.server.modules.GraphFeatureServiceWorkerClients

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

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

import com.twitter.inject.Logging

import com.twitter.storehaus.ReadableStore

import com.twitter.util.Future

import javax.inject.Singleton

import scala.collection.mutable.ArrayBuffer

@Singleton

case class GetIntersectionStore(

graphFeatureServiceWorkerClients: GraphFeatureServiceWorkerClients,

statsReceiver: StatsReceiver)

extends ReadableStore[GetIntersectionQuery, CachedIntersectionResult]

with Logging {

import GetIntersectionStore.\_

private val stats = statsReceiver.scope("get\_intersection\_store")

private val requestCount = stats.counter(name = "request\_count")

private val aggregatorLatency = stats.stat("aggregator\_latency")

private val timeOutCounter = stats.counter("worker\_timeouts")

private val unknownErrorCounter = stats.counter("unknown\_errors")

override def multiGet[K1 <: GetIntersectionQuery](

ks: Set[K1]

): Map[K1, Future[Option[CachedIntersectionResult]]] = {

if (ks.isEmpty) {

Map.empty

} else {

requestCount.incr()

val head = ks.head

// We assume all the GetIntersectionQuery use the same userId and featureTypes

val userId = head.userId

val featureTypes = head.featureTypes

val presetFeatureTypes = head.presetFeatureTypes

val calculatedFeatureTypes = head.calculatedFeatureTypes

val intersectionIdLimit = head.intersectionIdLimit

val request = WorkerIntersectionRequest(

userId,

ks.map(\_.candidateId).toArray,

featureTypes,

presetFeatureTypes,

intersectionIdLimit

)

val resultFuture = Future

.collect(

graphFeatureServiceWorkerClients.workers.map { worker =>

worker

.getIntersection(request)

.rescue {

case \_: RequestTimeoutException =>

timeOutCounter.incr()

Future.value(DefaultWorkerIntersectionResponse)

case e =>

unknownErrorCounter.incr()

logger.error("Failure to load result.", e)

Future.value(DefaultWorkerIntersectionResponse)

}

}

).map { responses =>

Stat.time(aggregatorLatency) {

gfsIntersectionResponseAggregator(

responses,

calculatedFeatureTypes,

request.candidateUserIds,

intersectionIdLimit

)

}

}

ks.map { query =>

query -> resultFuture.map(\_.get(query.candidateId))

}.toMap

}

}

/\*\*

\* Function to merge GfsIntersectionResponse from workers into one result.

\*/

private def gfsIntersectionResponseAggregator(

responseList: Seq[WorkerIntersectionResponse],

features: Seq[FeatureType],

candidates: Seq[Long],

intersectionIdLimit: Int

): Map[Long, CachedIntersectionResult] = {

// Map of (candidate -> features -> type -> value)

val cube = Array.fill[Int](candidates.length, features.length, 3)(0)

// Map of (candidate -> features -> intersectionIds)

val ids = Array.fill[Option[ArrayBuffer[Long]]](candidates.length, features.length)(None)

val notZero = intersectionIdLimit != 0

for {

response <- responseList

(features, candidateIndex) <- response.results.zipWithIndex

(workerValue, featureIndex) <- features.zipWithIndex

} {

cube(candidateIndex)(featureIndex)(CountIndex) += workerValue.count

cube(candidateIndex)(featureIndex)(LeftDegreeIndex) += workerValue.leftNodeDegree

cube(candidateIndex)(featureIndex)(RightDegreeIndex) += workerValue.rightNodeDegree

if (notZero && workerValue.intersectionIds.nonEmpty) {

val arrayBuffer = ids(candidateIndex)(featureIndex) match {

case Some(buffer) => buffer

case None =>

val buffer = ArrayBuffer[Long]()

ids(candidateIndex)(featureIndex) = Some(buffer)

buffer

}

val intersectionIds = workerValue.intersectionIds

// Scan the intersectionId based on the Shard. The response order is consistent.

if (arrayBuffer.size < intersectionIdLimit) {

if (intersectionIds.size > intersectionIdLimit - arrayBuffer.size) {

arrayBuffer ++= intersectionIds.slice(0, intersectionIdLimit - arrayBuffer.size)

} else {

arrayBuffer ++= intersectionIds

}

}

}

}

candidates.zipWithIndex.map {

case (candidate, candidateIndex) =>

candidate -> CachedIntersectionResult(features.indices.map { featureIndex =>

WorkerIntersectionValue(

cube(candidateIndex)(featureIndex)(CountIndex),

cube(candidateIndex)(featureIndex)(LeftDegreeIndex),

cube(candidateIndex)(featureIndex)(RightDegreeIndex),

ids(candidateIndex)(featureIndex).getOrElse(Nil)

)

})

}.toMap

}

}

object GetIntersectionStore {

private[graph\_feature\_service] case class GetIntersectionQuery(

userId: Long,

candidateId: Long,

featureTypes: Seq[FeatureType],

presetFeatureTypes: PresetFeatureTypes,

featureTypesString: String,

calculatedFeatureTypes: Seq[FeatureType],

intersectionIdLimit: Int)

private[graph\_feature\_service] object GetIntersectionQuery {

def buildQueries(request: GetIntersectionRequest): Set[GetIntersectionQuery] = {

request.candidateUserIds.toSet.map { candidateId: Long =>

GetIntersectionQuery(

request.userId,

candidateId,

request.featureTypes,

request.presetFeatureTypes,

request.calculatedFeatureTypesString,

request.calculatedFeatureTypes,

request.intersectionIdLimit.getOrElse(DefaultIntersectionIdLimit)

)

}

}

}

// Don't return the intersectionId for better performance

private val DefaultIntersectionIdLimit = 0

private val DefaultWorkerIntersectionResponse = WorkerIntersectionResponse()

private val CountIndex = 0

private val LeftDegreeIndex = 1

private val RightDegreeIndex = 2

}