package com.twitter.follow\_recommendations.common.clients.socialgraph

import com.twitter.escherbird.util.stitchcache.StitchCache

import com.twitter.finagle.stats.NullStatsReceiver

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.follow\_recommendations.common.base.StatsUtil

import com.twitter.follow\_recommendations.common.models.FollowProof

import com.twitter.follow\_recommendations.common.models.UserIdWithTimestamp

import com.twitter.inject.Logging

import com.twitter.socialgraph.thriftscala.EdgesRequest

import com.twitter.socialgraph.thriftscala.IdsRequest

import com.twitter.socialgraph.thriftscala.IdsResult

import com.twitter.socialgraph.thriftscala.LookupContext

import com.twitter.socialgraph.thriftscala.OverCapacity

import com.twitter.socialgraph.thriftscala.PageRequest

import com.twitter.socialgraph.thriftscala.RelationshipType

import com.twitter.socialgraph.thriftscala.SrcRelationship

import com.twitter.socialgraph.util.ByteBufferUtil

import com.twitter.stitch.Stitch

import com.twitter.stitch.socialgraph.SocialGraph

import com.twitter.strato.client.Fetcher

import com.twitter.strato.generated.client.onboarding.socialGraphService.IdsClientColumn

import com.twitter.util.Duration

import com.twitter.util.Time

import java.nio.ByteBuffer

import javax.inject.Inject

import javax.inject.Singleton

case class RecentEdgesQuery(

userId: Long,

relations: Seq[RelationshipType],

// prefer to default value to better utilize the caching function of stitch

count: Option[Int] = Some(SocialGraphClient.MaxQuerySize),

performUnion: Boolean = true,

recentEdgesWindowOpt: Option[Duration] = None,

targets: Option[Seq[Long]] = None)

case class EdgeRequestQuery(

userId: Long,

relation: RelationshipType,

count: Option[Int] = Some(SocialGraphClient.MaxQuerySize),

performUnion: Boolean = true,

recentEdgesWindowOpt: Option[Duration] = None,

targets: Option[Seq[Long]] = None)

@Singleton

class SocialGraphClient @Inject() (

socialGraph: SocialGraph,

idsClientColumn: IdsClientColumn,

statsReceiver: StatsReceiver = NullStatsReceiver)

extends Logging {

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

private val cacheStats = stats.scope("cache")

private val getIntersectionsStats = stats.scope("getIntersections")

private val getIntersectionsFromCachedColumnStats =

stats.scope("getIntersectionsFromCachedColumn")

private val getRecentEdgesStats = stats.scope("getRecentEdges")

private val getRecentEdgesCachedStats = stats.scope("getRecentEdgesCached")

private val getRecentEdgesFromCachedColumnStats = stats.scope("getRecentEdgesFromCachedColumn")

private val getRecentEdgesCachedInternalStats = stats.scope("getRecentEdgesCachedInternal")

private val getRecentEdgesWithTimeStats = stats.scope("getRecentEdgesWithTime")

val sgsIdsFetcher: Fetcher[IdsRequest, Unit, IdsResult] = idsClientColumn.fetcher

private val recentEdgesCache = StitchCache[RecentEdgesQuery, Seq[Long]](

maxCacheSize = SocialGraphClient.MaxCacheSize,

ttl = SocialGraphClient.CacheTTL,

statsReceiver = cacheStats,

underlyingCall = getRecentEdges

)

def getRecentEdgesCached(

rq: RecentEdgesQuery,

useCachedStratoColumn: Boolean = true

): Stitch[Seq[Long]] = {

getRecentEdgesCachedStats.counter("requests").incr()

if (useCachedStratoColumn) {

getRecentEdgesFromCachedColumn(rq)

} else {

StatsUtil.profileStitch(

getRecentEdgesCachedInternal(rq),

getRecentEdgesCachedInternalStats

)

}

}

def getRecentEdgesCachedInternal(rq: RecentEdgesQuery): Stitch[Seq[Long]] = {

recentEdgesCache.readThrough(rq)

}

def getRecentEdgesFromCachedColumn(rq: RecentEdgesQuery): Stitch[Seq[Long]] = {

val pageRequest = rq.recentEdgesWindowOpt match {

case Some(recentEdgesWindow) =>

PageRequest(

count = rq.count,

cursor = Some(getEdgeCursor(recentEdgesWindow)),

selectAll = Some(true)

)

case \_ => PageRequest(count = rq.count)

}

val idsRequest = IdsRequest(

rq.relations.map { relationshipType =>

SrcRelationship(

source = rq.userId,

relationshipType = relationshipType,

targets = rq.targets

)

},

pageRequest = Some(pageRequest),

context = Some(LookupContext(performUnion = Some(rq.performUnion)))

)

val socialGraphStitch = sgsIdsFetcher

.fetch(idsRequest, Unit)

.map(\_.v)

.map { result =>

result

.map { idResult =>

val userIds: Seq[Long] = idResult.ids

getRecentEdgesFromCachedColumnStats.stat("num\_edges").add(userIds.size)

userIds

}.getOrElse(Seq.empty)

}

.rescue {

case e: Exception =>

stats.counter(e.getClass.getSimpleName).incr()

Stitch.Nil

}

StatsUtil.profileStitch(

socialGraphStitch,

getRecentEdgesFromCachedColumnStats

)

}

def getRecentEdges(rq: RecentEdgesQuery): Stitch[Seq[Long]] = {

val pageRequest = rq.recentEdgesWindowOpt match {

case Some(recentEdgesWindow) =>

PageRequest(

count = rq.count,

cursor = Some(getEdgeCursor(recentEdgesWindow)),

selectAll = Some(true)

)

case \_ => PageRequest(count = rq.count)

}

val socialGraphStitch = socialGraph

.ids(

IdsRequest(

rq.relations.map { relationshipType =>

SrcRelationship(

source = rq.userId,

relationshipType = relationshipType,

targets = rq.targets

)

},

pageRequest = Some(pageRequest),

context = Some(LookupContext(performUnion = Some(rq.performUnion)))

)

)

.map { idsResult =>

val userIds: Seq[Long] = idsResult.ids

getRecentEdgesStats.stat("num\_edges").add(userIds.size)

userIds

}

.rescue {

case e: OverCapacity =>

stats.counter(e.getClass.getSimpleName).incr()

logger.warn("SGS Over Capacity", e)

Stitch.Nil

}

StatsUtil.profileStitch(

socialGraphStitch,

getRecentEdgesStats

)

}

// This method return recent edges of (userId, timeInMs)

def getRecentEdgesWithTime(rq: EdgeRequestQuery): Stitch[Seq[UserIdWithTimestamp]] = {

val pageRequest = rq.recentEdgesWindowOpt match {

case Some(recentEdgesWindow) =>

PageRequest(

count = rq.count,

cursor = Some(getEdgeCursor(recentEdgesWindow)),

selectAll = Some(true)

)

case \_ => PageRequest(count = rq.count)

}

val socialGraphStitch = socialGraph

.edges(

EdgesRequest(

SrcRelationship(

source = rq.userId,

relationshipType = rq.relation,

targets = rq.targets

),

pageRequest = Some(pageRequest),

context = Some(LookupContext(performUnion = Some(rq.performUnion)))

)

)

.map { edgesResult =>

val userIds = edgesResult.edges.map { socialEdge =>

UserIdWithTimestamp(socialEdge.target, socialEdge.updatedAt)

}

getRecentEdgesWithTimeStats.stat("num\_edges").add(userIds.size)

userIds

}

.rescue {

case e: OverCapacity =>

stats.counter(e.getClass.getSimpleName).incr()

logger.warn("SGS Over Capacity", e)

Stitch.Nil

}

StatsUtil.profileStitch(

socialGraphStitch,

getRecentEdgesWithTimeStats

)

}

// This method returns the cursor for a time duration, such that all the edges returned by SGS will be created

// in the range (now-window, now)

def getEdgeCursor(window: Duration): ByteBuffer = {

val cursorInLong = (-(Time.now - window).inMilliseconds) << 20

ByteBufferUtil.fromLong(cursorInLong)

}

// notice that this is more expensive but more realtime than the GFS one

def getIntersections(

userId: Long,

candidateIds: Seq[Long],

numIntersectionIds: Int

): Stitch[Map[Long, FollowProof]] = {

val socialGraphStitch: Stitch[Map[Long, FollowProof]] = Stitch

.collect(candidateIds.map { candidateId =>

socialGraph

.ids(

IdsRequest(

Seq(

SrcRelationship(userId, RelationshipType.Following),

SrcRelationship(candidateId, RelationshipType.FollowedBy)

),

pageRequest = Some(PageRequest(count = Some(numIntersectionIds)))

)

).map { idsResult =>

getIntersectionsStats.stat("num\_edges").add(idsResult.ids.size)

(candidateId -> FollowProof(idsResult.ids, idsResult.ids.size))

}

}).map(\_.toMap)

.rescue {

case e: OverCapacity =>

stats.counter(e.getClass.getSimpleName).incr()

logger.warn("social graph over capacity in hydrating social proof", e)

Stitch.value(Map.empty)

}

StatsUtil.profileStitch(

socialGraphStitch,

getIntersectionsStats

)

}

def getIntersectionsFromCachedColumn(

userId: Long,

candidateIds: Seq[Long],

numIntersectionIds: Int

): Stitch[Map[Long, FollowProof]] = {

val socialGraphStitch: Stitch[Map[Long, FollowProof]] = Stitch

.collect(candidateIds.map { candidateId =>

val idsRequest = IdsRequest(

Seq(

SrcRelationship(userId, RelationshipType.Following),

SrcRelationship(candidateId, RelationshipType.FollowedBy)

),

pageRequest = Some(PageRequest(count = Some(numIntersectionIds)))

)

sgsIdsFetcher

.fetch(idsRequest, Unit)

.map(\_.v)

.map { resultOpt =>

resultOpt.map { idsResult =>

getIntersectionsFromCachedColumnStats.stat("num\_edges").add(idsResult.ids.size)

candidateId -> FollowProof(idsResult.ids, idsResult.ids.size)

}

}

}).map(\_.flatten.toMap)

.rescue {

case e: Exception =>

stats.counter(e.getClass.getSimpleName).incr()

Stitch.value(Map.empty)

}

StatsUtil.profileStitch(

socialGraphStitch,

getIntersectionsFromCachedColumnStats

)

}

def getInvalidRelationshipUserIds(

userId: Long,

maxNumRelationship: Int = SocialGraphClient.MaxNumInvalidRelationship

): Stitch[Seq[Long]] = {

getRecentEdges(

RecentEdgesQuery(

userId,

SocialGraphClient.InvalidRelationshipTypes,

Some(maxNumRelationship)

)

)

}

def getInvalidRelationshipUserIdsFromCachedColumn(

userId: Long,

maxNumRelationship: Int = SocialGraphClient.MaxNumInvalidRelationship

): Stitch[Seq[Long]] = {

getRecentEdgesFromCachedColumn(

RecentEdgesQuery(

userId,

SocialGraphClient.InvalidRelationshipTypes,

Some(maxNumRelationship)

)

)

}

def getRecentFollowedUserIds(userId: Long): Stitch[Seq[Long]] = {

getRecentEdges(

RecentEdgesQuery(

userId,

Seq(RelationshipType.Following)

)

)

}

def getRecentFollowedUserIdsFromCachedColumn(userId: Long): Stitch[Seq[Long]] = {

getRecentEdgesFromCachedColumn(

RecentEdgesQuery(

userId,

Seq(RelationshipType.Following)

)

)

}

def getRecentFollowedUserIdsWithTime(userId: Long): Stitch[Seq[UserIdWithTimestamp]] = {

getRecentEdgesWithTime(

EdgeRequestQuery(

userId,

RelationshipType.Following

)

)

}

def getRecentFollowedByUserIds(userId: Long): Stitch[Seq[Long]] = {

getRecentEdges(

RecentEdgesQuery(

userId,

Seq(RelationshipType.FollowedBy)

)

)

}

def getRecentFollowedByUserIdsFromCachedColumn(userId: Long): Stitch[Seq[Long]] = {

getRecentEdgesFromCachedColumn(

RecentEdgesQuery(

userId,

Seq(RelationshipType.FollowedBy)

)

)

}

def getRecentFollowedUserIdsWithTimeWindow(

userId: Long,

timeWindow: Duration

): Stitch[Seq[Long]] = {

getRecentEdges(

RecentEdgesQuery(

userId,

Seq(RelationshipType.Following),

recentEdgesWindowOpt = Some(timeWindow)

)

)

}

}

object SocialGraphClient {

val MaxQuerySize: Int = 500

val MaxCacheSize: Int = 5000000

// Ref: src/thrift/com/twitter/socialgraph/social\_graph\_service.thrift

val MaxNumInvalidRelationship: Int = 5000

val CacheTTL: Duration = Duration.fromHours(24)

val InvalidRelationshipTypes: Seq[RelationshipType] = Seq(

RelationshipType.HideRecommendations,

RelationshipType.Blocking,

RelationshipType.BlockedBy,

RelationshipType.Muting,

RelationshipType.MutedBy,

RelationshipType.ReportedAsSpam,

RelationshipType.ReportedAsSpamBy,

RelationshipType.ReportedAsAbuse,

RelationshipType.ReportedAsAbuseBy,

RelationshipType.FollowRequestOutgoing,

RelationshipType.Following,

RelationshipType.UsedToFollow,

)

/\*\*

\*

\* Whether to call SGS to validate each candidate based on the number of invalid relationship users

\* prefetched during request building step. This aims to not omit any invalid candidates that are

\* not filtered out in previous steps.

\* If the number is 0, this might be a fail-opened SGS call.

\* If the number is larger or equal to 5000, this could hit SGS page size limit.

\* Both cases account for a small percentage of the total traffic (<5%).

\*

\* @param numInvalidRelationshipUsers number of invalid relationship users fetched from getInvalidRelationshipUserIds

\* @return whether to enable post-ranker SGS predicate

\*/

def enablePostRankerSgsPredicate(numInvalidRelationshipUsers: Int): Boolean = {

numInvalidRelationshipUsers == 0 || numInvalidRelationshipUsers >= MaxNumInvalidRelationship

}

}