package com.twitter.recosinjector.clients

import com.twitter.conversions.DurationOps.\_

import com.twitter.finagle.memcached.Client

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.io.Buf

import com.twitter.recos.internal.thriftscala.{RecosHoseEntities, RecosHoseEntity}

import com.twitter.servo.cache.ThriftSerializer

import com.twitter.util.{Duration, Future, Time}

import org.apache.thrift.protocol.TBinaryProtocol

case class CacheEntityEntry(

cachePrefix: String,

hashedEntityId: Int,

entity: String) {

val fullKey: String = cachePrefix + hashedEntityId

}

object RecosHoseEntitiesCache {

val EntityTTL: Duration = 30.hours

val EntitiesSerializer =

new ThriftSerializer[RecosHoseEntities](RecosHoseEntities, new TBinaryProtocol.Factory())

val HashtagPrefix: String = "h"

val UrlPrefix: String = "u"

}

/\*\*

\* A cache layer to store entities.

\* Graph services like user\_tweet\_entity\_graph and user\_url\_graph store user interactions with

\* entities in a tweet, such as HashTags and URLs. These entities are string values that can be

\* potentially very big. Therefore, we instead store a hashed id in the graph edge, and keep a

\* (hashedId -> entity) mapping in this cache. The actual entity values can be recovered

\* by the graph service at serving time using this cache.

\*/

class RecosHoseEntitiesCache(client: Client) {

import RecosHoseEntitiesCache.\_

private def isEntityWithinTTL(entity: RecosHoseEntity, ttlInMillis: Long): Boolean = {

entity.timestamp.exists(timestamp => Time.now.inMilliseconds - timestamp <= ttlInMillis)

}

/\*\*

\* Add a new RecosHoseEntity into RecosHoseEntities

\*/

private def updateRecosHoseEntities(

existingEntitiesOpt: Option[RecosHoseEntities],

newEntityString: String,

stats: StatsReceiver

): RecosHoseEntities = {

val existingEntities = existingEntitiesOpt.map(\_.entities).getOrElse(Nil)

// Discard expired and duplicate existing entities

val validExistingEntities = existingEntities

.filter(entity => isEntityWithinTTL(entity, EntityTTL.inMillis))

.filter(\_.entity != newEntityString)

val newRecosHoseEntity = RecosHoseEntity(newEntityString, Some(Time.now.inMilliseconds))

RecosHoseEntities(validExistingEntities :+ newRecosHoseEntity)

}

private def getRecosHoseEntitiesCache(

cacheEntries: Seq[CacheEntityEntry],

stats: StatsReceiver

): Future[Map[String, Option[RecosHoseEntities]]] = {

client

.get(cacheEntries.map(\_.fullKey))

.map(\_.map {

case (cacheKey, buf) =>

val recosHoseEntitiesTry = EntitiesSerializer.from(Buf.ByteArray.Owned.extract(buf))

if (recosHoseEntitiesTry.isThrow) {

stats.counter("cache\_get\_deserialization\_failure").incr()

}

cacheKey -> recosHoseEntitiesTry.toOption

})

.onSuccess { \_ => stats.counter("get\_cache\_success").incr() }

.onFailure { ex =>

stats.scope("get\_cache\_failure").counter(ex.getClass.getSimpleName).incr()

}

}

private def putRecosHoseEntitiesCache(

cacheKey: String,

recosHoseEntities: RecosHoseEntities,

stats: StatsReceiver

): Unit = {

val serialized = EntitiesSerializer.to(recosHoseEntities)

if (serialized.isThrow) {

stats.counter("cache\_put\_serialization\_failure").incr()

}

serialized.toOption.map { bytes =>

client

.set(cacheKey, 0, EntityTTL.fromNow, Buf.ByteArray.Owned(bytes))

.onSuccess { \_ => stats.counter("put\_cache\_success").incr() }

.onFailure { ex =>

stats.scope("put\_cache\_failure").counter(ex.getClass.getSimpleName).incr()

}

}

}

/\*\*

\* Store a list of new entities into the cache by their cacheKeys, and remove expired/invalid

\* values in the existing cache entries at the same time

\*/

def updateEntitiesCache(

newCacheEntries: Seq[CacheEntityEntry],

stats: StatsReceiver

): Future[Unit] = {

stats.counter("update\_cache\_request").incr()

getRecosHoseEntitiesCache(newCacheEntries, stats)

.map { existingCacheEntries =>

newCacheEntries.foreach { newCacheEntry =>

val fullKey = newCacheEntry.fullKey

val existingRecosHoseEntities = existingCacheEntries.get(fullKey).flatten

stats.stat("num\_existing\_entities").add(existingRecosHoseEntities.size)

if (existingRecosHoseEntities.isEmpty) {

stats.counter("existing\_entities\_empty").incr()

}

val updatedRecosHoseEntities = updateRecosHoseEntities(

existingRecosHoseEntities,

newCacheEntry.entity,

stats

)

stats.stat("num\_updated\_entities").add(updatedRecosHoseEntities.entities.size)

if (updatedRecosHoseEntities.entities.nonEmpty) {

putRecosHoseEntitiesCache(fullKey, updatedRecosHoseEntities, stats)

}

}

}

.onSuccess { \_ => stats.counter("update\_cache\_success").incr() }

.onFailure { ex =>

stats.scope("update\_cache\_failure").counter(ex.getClass.getSimpleName).incr()

}

}

}