package com.twitter.timelineranker.util

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.storehaus.Store

import com.twitter.timelineranker.contentfeatures.ContentFeaturesProvider

import com.twitter.timelineranker.model.RecapQuery

import com.twitter.timelineranker.recap.model.ContentFeatures

import com.twitter.timelines.model.TweetId

import com.twitter.timelines.util.FailOpenHandler

import com.twitter.timelines.util.FutureUtils

import com.twitter.timelines.util.stats.FutureObserver

import com.twitter.util.Future

object CachingContentFeaturesProvider {

private sealed trait CacheResult

private object CacheFailure extends CacheResult

private object CacheMiss extends CacheResult

private case class CacheHit(t: ContentFeatures) extends CacheResult

def isHit(result: CacheResult): Boolean = result != CacheMiss && result != CacheFailure

def isMiss(result: CacheResult): Boolean = result == CacheMiss

}

class CachingContentFeaturesProvider(

underlying: ContentFeaturesProvider,

contentFeaturesCache: Store[TweetId, ContentFeatures],

statsReceiver: StatsReceiver)

extends ContentFeaturesProvider {

import CachingContentFeaturesProvider.\_

private val scopedStatsReceiver = statsReceiver.scope("CachingContentFeaturesProvider")

private val cacheScope = scopedStatsReceiver.scope("cache")

private val cacheReadsCounter = cacheScope.counter("reads")

private val cacheReadFailOpenHandler = new FailOpenHandler(cacheScope.scope("reads"))

private val cacheHitsCounter = cacheScope.counter("hits")

private val cacheMissesCounter = cacheScope.counter("misses")

private val cacheFailuresCounter = cacheScope.counter("failures")

private val cacheWritesCounter = cacheScope.counter("writes")

private val cacheWriteObserver = FutureObserver(cacheScope.scope("writes"))

private val underlyingScope = scopedStatsReceiver.scope("underlying")

private val underlyingReadsCounter = underlyingScope.counter("reads")

override def apply(

query: RecapQuery,

tweetIds: Seq[TweetId]

): Future[Map[TweetId, ContentFeatures]] = {

if (tweetIds.nonEmpty) {

val distinctTweetIds = tweetIds.toSet

readFromCache(distinctTweetIds).flatMap { cacheResultsFuture =>

val (resultsFromCache, missedTweetIds) = partitionHitsMisses(cacheResultsFuture)

if (missedTweetIds.nonEmpty) {

underlyingReadsCounter.incr(missedTweetIds.size)

val resultsFromUnderlyingFu = underlying(query, missedTweetIds)

resultsFromUnderlyingFu.onSuccess(writeToCache)

resultsFromUnderlyingFu

.map(resultsFromUnderlying => resultsFromCache ++ resultsFromUnderlying)

} else {

Future.value(resultsFromCache)

}

}

} else {

FutureUtils.EmptyMap

}

}

private def readFromCache(tweetIds: Set[TweetId]): Future[Seq[(TweetId, CacheResult)]] = {

cacheReadsCounter.incr(tweetIds.size)

Future.collect(

contentFeaturesCache

.multiGet(tweetIds)

.toSeq

.map {

case (tweetId, cacheResultOptionFuture) =>

cacheReadFailOpenHandler(

cacheResultOptionFuture.map {

case Some(t: ContentFeatures) => tweetId -> CacheHit(t)

case None => tweetId -> CacheMiss

}

) { \_: Throwable => Future.value(tweetId -> CacheFailure) }

}

)

}

private def partitionHitsMisses(

cacheResults: Seq[(TweetId, CacheResult)]

): (Map[TweetId, ContentFeatures], Seq[TweetId]) = {

val (hits, missesAndFailures) = cacheResults.partition {

case (\_, cacheResult) => isHit(cacheResult)

}

val (misses, cacheFailures) = missesAndFailures.partition {

case (\_, cacheResult) => isMiss(cacheResult)

}

val cacheHits = hits.collect { case (tweetId, CacheHit(t)) => (tweetId, t) }.toMap

val cacheMisses = misses.collect { case (tweetId, \_) => tweetId }

cacheHitsCounter.incr(cacheHits.size)

cacheMissesCounter.incr(cacheMisses.size)

cacheFailuresCounter.incr(cacheFailures.size)

(cacheHits, cacheMisses)

}

private def writeToCache(results: Map[TweetId, ContentFeatures]): Unit = {

if (results.nonEmpty) {

cacheWritesCounter.incr(results.size)

val indexedResults = results.map {

case (tweetId, contentFeatures) =>

(tweetId, Some(contentFeatures))

}

contentFeaturesCache

.multiPut(indexedResults)

.map {

case (\_, statusFu) =>

cacheWriteObserver(statusFu)

}

}

}

}