package com.twitter.tweetypie

package store

import com.fasterxml.jackson.databind.ObjectMapper

import com.fasterxml.jackson.module.scala.DefaultScalaModule

import com.twitter.scrooge.TFieldBlob

import com.twitter.servo.cache.LockingCache.\_

import com.twitter.servo.cache.\_

import com.twitter.tweetypie.additionalfields.AdditionalFields

import com.twitter.tweetypie.repository.CachedBounceDeleted.isBounceDeleted

import com.twitter.tweetypie.repository.CachedBounceDeleted.toBounceDeletedCachedTweet

import com.twitter.tweetypie.repository.\_

import com.twitter.tweetypie.store.TweetUpdate.\_

import com.twitter.tweetypie.thriftscala.\_

import com.twitter.util.Time

import diffshow.DiffShow

trait CachingTweetStore

extends TweetStoreBase[CachingTweetStore]

with InsertTweet.Store

with ReplicatedInsertTweet.Store

with DeleteTweet.Store

with AsyncDeleteTweet.Store

with ReplicatedDeleteTweet.Store

with UndeleteTweet.Store

with AsyncUndeleteTweet.Store

with ReplicatedUndeleteTweet.Store

with SetAdditionalFields.Store

with ReplicatedSetAdditionalFields.Store

with DeleteAdditionalFields.Store

with AsyncDeleteAdditionalFields.Store

with ReplicatedDeleteAdditionalFields.Store

with ScrubGeo.Store

with ReplicatedScrubGeo.Store

with Takedown.Store

with ReplicatedTakedown.Store

with Flush.Store

with UpdatePossiblySensitiveTweet.Store

with AsyncUpdatePossiblySensitiveTweet.Store

with ReplicatedUpdatePossiblySensitiveTweet.Store {

def wrap(w: TweetStore.Wrap): CachingTweetStore =

new TweetStoreWrapper(w, this)

with CachingTweetStore

with InsertTweet.StoreWrapper

with ReplicatedInsertTweet.StoreWrapper

with DeleteTweet.StoreWrapper

with AsyncDeleteTweet.StoreWrapper

with ReplicatedDeleteTweet.StoreWrapper

with UndeleteTweet.StoreWrapper

with AsyncUndeleteTweet.StoreWrapper

with ReplicatedUndeleteTweet.StoreWrapper

with SetAdditionalFields.StoreWrapper

with ReplicatedSetAdditionalFields.StoreWrapper

with DeleteAdditionalFields.StoreWrapper

with AsyncDeleteAdditionalFields.StoreWrapper

with ReplicatedDeleteAdditionalFields.StoreWrapper

with ScrubGeo.StoreWrapper

with ReplicatedScrubGeo.StoreWrapper

with Takedown.StoreWrapper

with ReplicatedTakedown.StoreWrapper

with Flush.StoreWrapper

with UpdatePossiblySensitiveTweet.StoreWrapper

with AsyncUpdatePossiblySensitiveTweet.StoreWrapper

with ReplicatedUpdatePossiblySensitiveTweet.StoreWrapper

}

object CachingTweetStore {

val Action: AsyncWriteAction.CacheUpdate.type = AsyncWriteAction.CacheUpdate

def apply(

tweetCache: LockingCache[TweetKey, Cached[CachedTweet]],

tweetKeyFactory: TweetKeyFactory,

stats: StatsReceiver

): CachingTweetStore = {

val ops =

new CachingTweetStoreOps(

tweetCache,

tweetKeyFactory,

stats

)

new CachingTweetStore {

override val insertTweet: FutureEffect[InsertTweet.Event] = {

FutureEffect[InsertTweet.Event](e =>

ops.insertTweet(e.internalTweet, e.initialTweetUpdateRequest))

}

override val replicatedInsertTweet: FutureEffect[ReplicatedInsertTweet.Event] =

FutureEffect[ReplicatedInsertTweet.Event](e =>

ops.insertTweet(e.cachedTweet, e.initialTweetUpdateRequest))

override val deleteTweet: FutureEffect[DeleteTweet.Event] =

FutureEffect[DeleteTweet.Event](e =>

ops.deleteTweet(e.tweet.id, updateOnly = true, isBounceDelete = e.isBounceDelete))

override val asyncDeleteTweet: FutureEffect[AsyncDeleteTweet.Event] =

FutureEffect[AsyncDeleteTweet.Event](e =>

ops.deleteTweet(e.tweet.id, updateOnly = true, isBounceDelete = e.isBounceDelete))

override val retryAsyncDeleteTweet: FutureEffect[

TweetStoreRetryEvent[AsyncDeleteTweet.Event]

] =

TweetStore.retry(Action, asyncDeleteTweet)

override val replicatedDeleteTweet: FutureEffect[ReplicatedDeleteTweet.Event] =

FutureEffect[ReplicatedDeleteTweet.Event](e =>

ops.deleteTweet(

tweetId = e.tweet.id,

updateOnly = e.isErasure,

isBounceDelete = e.isBounceDelete

))

override val undeleteTweet: FutureEffect[UndeleteTweet.Event] =

FutureEffect[UndeleteTweet.Event](e => ops.undeleteTweet(e.internalTweet))

override val asyncUndeleteTweet: FutureEffect[AsyncUndeleteTweet.Event] =

FutureEffect[AsyncUndeleteTweet.Event](e => ops.undeleteTweet(e.cachedTweet))

override val retryAsyncUndeleteTweet: FutureEffect[

TweetStoreRetryEvent[AsyncUndeleteTweet.Event]

] =

TweetStore.retry(Action, asyncUndeleteTweet)

override val replicatedUndeleteTweet: FutureEffect[ReplicatedUndeleteTweet.Event] =

FutureEffect[ReplicatedUndeleteTweet.Event](e => ops.undeleteTweet(e.cachedTweet))

override val setAdditionalFields: FutureEffect[SetAdditionalFields.Event] =

FutureEffect[SetAdditionalFields.Event](e => ops.setAdditionalFields(e.additionalFields))

override val replicatedSetAdditionalFields: FutureEffect[

ReplicatedSetAdditionalFields.Event

] =

FutureEffect[ReplicatedSetAdditionalFields.Event](e =>

ops.setAdditionalFields(e.additionalFields))

override val deleteAdditionalFields: FutureEffect[DeleteAdditionalFields.Event] =

FutureEffect[DeleteAdditionalFields.Event](e =>

ops.deleteAdditionalFields(e.tweetId, e.fieldIds))

override val asyncDeleteAdditionalFields: FutureEffect[AsyncDeleteAdditionalFields.Event] =

FutureEffect[AsyncDeleteAdditionalFields.Event](e =>

ops.deleteAdditionalFields(e.tweetId, e.fieldIds))

override val retryAsyncDeleteAdditionalFields: FutureEffect[

TweetStoreRetryEvent[AsyncDeleteAdditionalFields.Event]

] =

TweetStore.retry(Action, asyncDeleteAdditionalFields)

override val replicatedDeleteAdditionalFields: FutureEffect[

ReplicatedDeleteAdditionalFields.Event

] =

FutureEffect[ReplicatedDeleteAdditionalFields.Event](e =>

ops.deleteAdditionalFields(e.tweetId, e.fieldIds))

override val scrubGeo: FutureEffect[ScrubGeo.Event] =

FutureEffect[ScrubGeo.Event](e => ops.scrubGeo(e.tweetIds))

override val replicatedScrubGeo: FutureEffect[ReplicatedScrubGeo.Event] =

FutureEffect[ReplicatedScrubGeo.Event](e => ops.scrubGeo(e.tweetIds))

override val takedown: FutureEffect[Takedown.Event] =

FutureEffect[Takedown.Event](e => ops.takedown(e.tweet))

override val replicatedTakedown: FutureEffect[ReplicatedTakedown.Event] =

FutureEffect[ReplicatedTakedown.Event](e => ops.takedown(e.tweet))

override val flush: FutureEffect[Flush.Event] =

FutureEffect[Flush.Event](e => ops.flushTweets(e.tweetIds, logExisting = e.logExisting))

.onlyIf(\_.flushTweets)

override val updatePossiblySensitiveTweet: FutureEffect[UpdatePossiblySensitiveTweet.Event] =

FutureEffect[UpdatePossiblySensitiveTweet.Event](e => ops.updatePossiblySensitive(e.tweet))

override val replicatedUpdatePossiblySensitiveTweet: FutureEffect[

ReplicatedUpdatePossiblySensitiveTweet.Event

] =

FutureEffect[ReplicatedUpdatePossiblySensitiveTweet.Event](e =>

ops.updatePossiblySensitive(e.tweet))

override val asyncUpdatePossiblySensitiveTweet: FutureEffect[

AsyncUpdatePossiblySensitiveTweet.Event

] =

FutureEffect[AsyncUpdatePossiblySensitiveTweet.Event](e =>

ops.updatePossiblySensitive(e.tweet))

override val retryAsyncUpdatePossiblySensitiveTweet: FutureEffect[

TweetStoreRetryEvent[AsyncUpdatePossiblySensitiveTweet.Event]

] =

TweetStore.retry(Action, asyncUpdatePossiblySensitiveTweet)

}

}

}

private class CachingTweetStoreOps(

tweetCache: LockingCache[TweetKey, Cached[CachedTweet]],

tweetKeyFactory: TweetKeyFactory,

stats: StatsReceiver,

evictionRetries: Int = 3) {

type CachedTweetHandler = Handler[Cached[CachedTweet]]

private val preferNewestPicker = new PreferNewestCached[CachedTweet]

private val evictionFailedCounter = stats.counter("eviction\_failures")

private val cacheFlushesLog = Logger("com.twitter.tweetypie.store.CacheFlushesLog")

private[this] val mapper = new ObjectMapper().registerModule(DefaultScalaModule)

/\*\*

\* Inserts a tweet into cache, recording all compiled additional fields and all

\* included passthrough fields. Additionally if the insertion event contains

\* a 'InitialTweetUpdateRequest` we will update the cache entry for this tweet's

\* initialTweet.

\*/

def insertTweet(

ct: CachedTweet,

initialTweetUpdateRequest: Option[InitialTweetUpdateRequest]

): Future[Unit] =

lockAndSet(

ct.tweet.id,

insertTweetHandler(ct)

).flatMap { \_ =>

initialTweetUpdateRequest match {

case Some(request) =>

lockAndSet(

request.initialTweetId,

updateTweetHandler(tweet => InitialTweetUpdate.updateTweet(tweet, request))

)

case None =>

Future.Unit

}

}

/\*\*

\* Writes a `deleted` tombstone to cache. If `updateOnly` is true, then we only

\* write the tombstone if the tweet is already in cache. If `isBounceDelete` we

\* write a special bounce-deleted CachedTweet record to cache.

\*/

def deleteTweet(tweetId: TweetId, updateOnly: Boolean, isBounceDelete: Boolean): Future[Unit] = {

// We only need to store a CachedTweet value the tweet is bounce-deleted to support rendering

// timeline tombstones for tweets that violated the Twitter Rules. see go/bounced-tweet

val cachedValue = if (isBounceDelete) {

found(toBounceDeletedCachedTweet(tweetId))

} else {

writeThroughCached[CachedTweet](None, CachedValueStatus.Deleted)

}

val pickerHandler =

if (updateOnly) {

deleteTweetUpdateOnlyHandler(cachedValue)

} else {

deleteTweetHandler(cachedValue)

}

lockAndSet(tweetId, pickerHandler)

}

def undeleteTweet(ct: CachedTweet): Future[Unit] =

lockAndSet(

ct.tweet.id,

insertTweetHandler(ct)

)

def setAdditionalFields(tweet: Tweet): Future[Unit] =

lockAndSet(tweet.id, setFieldsHandler(AdditionalFields.additionalFields(tweet)))

def deleteAdditionalFields(tweetId: TweetId, fieldIds: Seq[FieldId]): Future[Unit] =

lockAndSet(tweetId, deleteFieldsHandler(fieldIds))

def scrubGeo(tweetIds: Seq[TweetId]): Future[Unit] =

Future.join {

tweetIds.map { id =>

// First, attempt to modify any tweets that are in cache to

// avoid having to reload the cached tweet from storage.

lockAndSet(id, scrubGeoHandler).unit.rescue {

case \_: OptimisticLockingCache.LockAndSetFailure =>

// If the modification fails, then remove whatever is in

// cache. This is much more likely to succeed because it

// does not require multiple successful requests to cache.

// This will force the tweet to be loaded from storage the

// next time it is requested, and the stored tweet will have

// the geo information removed.

//

// This eviction path was added due to frequent failures of

// the in-place modification code path, causing geoscrub

// daemon tasks to fail.

evictOne(tweetKeyFactory.fromId(id), evictionRetries)

}

}

}

def takedown(tweet: Tweet): Future[Unit] =

lockAndSet(tweet.id, updateCachedTweetHandler(copyTakedownFieldsForUpdate(tweet)))

def updatePossiblySensitive(tweet: Tweet): Future[Unit] =

lockAndSet(tweet.id, updateTweetHandler(copyNsfwFieldsForUpdate(tweet)))

def flushTweets(tweetIds: Seq[TweetId], logExisting: Boolean = false): Future[Unit] = {

val tweetKeys = tweetIds.map(tweetKeyFactory.fromId)

Future.when(logExisting) { logExistingValues(tweetKeys) }.ensure {

evictAll(tweetKeys)

}

}

/\*\*

\* A LockingCache.Handler that inserts a tweet into cache.

\*/

private def insertTweetHandler(newValue: CachedTweet): Handler[Cached[CachedTweet]] =

AlwaysSetHandler(Some(writeThroughCached(Some(newValue), CachedValueStatus.Found)))

private def foundAndNotBounced(c: Cached[CachedTweet]) =

c.status == CachedValueStatus.Found && !isBounceDeleted(c)

/\*\*

\* A LockingCache.Handler that updates an existing CachedTweet in cache.

\*/

private def updateTweetHandler(update: Tweet => Tweet): CachedTweetHandler =

inCache =>

for {

cached <- inCache.filter(foundAndNotBounced)

cachedTweet <- cached.value

updatedTweet = update(cachedTweet.tweet)

} yield found(cachedTweet.copy(tweet = updatedTweet))

/\*\*

\* A LockingCache.Handler that updates an existing CachedTweet in cache.

\*/

private def updateCachedTweetHandler(update: CachedTweet => CachedTweet): CachedTweetHandler =

inCache =>

for {

cached <- inCache.filter(foundAndNotBounced)

cachedTweet <- cached.value

updatedCachedTweet = update(cachedTweet)

} yield found(updatedCachedTweet)

private def deleteTweetHandler(value: Cached[CachedTweet]): CachedTweetHandler =

PickingHandler(value, preferNewestPicker)

private def deleteTweetUpdateOnlyHandler(value: Cached[CachedTweet]): CachedTweetHandler =

UpdateOnlyPickingHandler(value, preferNewestPicker)

private def setFieldsHandler(additional: Seq[TFieldBlob]): CachedTweetHandler =

inCache =>

for {

cached <- inCache.filter(foundAndNotBounced)

cachedTweet <- cached.value

updatedTweet = AdditionalFields.setAdditionalFields(cachedTweet.tweet, additional)

updatedCachedTweet = CachedTweet(updatedTweet)

} yield found(updatedCachedTweet)

private def deleteFieldsHandler(fieldIds: Seq[FieldId]): CachedTweetHandler =

inCache =>

for {

cached <- inCache.filter(foundAndNotBounced)

cachedTweet <- cached.value

updatedTweet = AdditionalFields.unsetFields(cachedTweet.tweet, fieldIds)

scrubbedCachedTweet = cachedTweet.copy(tweet = updatedTweet)

} yield found(scrubbedCachedTweet)

private val scrubGeoHandler: CachedTweetHandler =

inCache =>

for {

cached <- inCache.filter(foundAndNotBounced)

cachedTweet <- cached.value

tweet = cachedTweet.tweet

coreData <- tweet.coreData if hasGeo(tweet)

scrubbedCoreData = coreData.copy(coordinates = None, placeId = None)

scrubbedTweet = tweet.copy(coreData = Some(scrubbedCoreData), place = None)

scrubbedCachedTweet = cachedTweet.copy(tweet = scrubbedTweet)

} yield found(scrubbedCachedTweet)

private def evictOne(key: TweetKey, tries: Int): Future[Int] =

tweetCache.delete(key).transform {

case Throw(\_) if tries > 1 => evictOne(key, tries - 1)

case Throw(\_) => Future.value(1)

case Return(\_) => Future.value(0)

}

private def evictAll(keys: Seq[TweetKey]): Future[Unit] =

Future

.collect {

keys.map(evictOne(\_, evictionRetries))

}

.onSuccess { (failures: Seq[Int]) => evictionFailedCounter.incr(failures.sum) }

.unit

private def logExistingValues(keys: Seq[TweetKey]): Future[Unit] =

tweetCache

.get(keys)

.map { existing =>

for {

(key, cached) <- existing.found

cachedTweet <- cached.value

tweet = cachedTweet.tweet

} yield {

cacheFlushesLog.info(

mapper.writeValueAsString(

Map(

"key" -> key,

"tweet\_id" -> tweet.id,

"tweet" -> DiffShow.show(tweet)

)

)

)

}

}

.unit

private def found(value: CachedTweet): Cached[CachedTweet] =

writeThroughCached(Some(value), CachedValueStatus.Found)

private def writeThroughCached[V](value: Option[V], status: CachedValueStatus): Cached[V] = {

val now = Time.now

Cached(value, status, now, None, Some(now))

}

private def lockAndSet(tweetId: TweetId, handler: LockingCache.Handler[Cached[CachedTweet]]) =

tweetCache.lockAndSet(tweetKeyFactory.fromId(tweetId), handler).unit

}