package com.twitter.tweetypie

package store

import com.twitter.concurrent.Serialized

import com.twitter.servo.cache.LockingCache.Handler

import com.twitter.servo.cache.\_

import com.twitter.tweetypie.repository.BookmarksKey

import com.twitter.tweetypie.repository.FavsKey

import com.twitter.tweetypie.repository.QuotesKey

import com.twitter.tweetypie.repository.RepliesKey

import com.twitter.tweetypie.repository.RetweetsKey

import com.twitter.tweetypie.repository.TweetCountKey

import com.twitter.util.Duration

import com.twitter.util.Timer

import scala.collection.mutable

trait TweetCountsCacheUpdatingStore

extends TweetStoreBase[TweetCountsCacheUpdatingStore]

with InsertTweet.Store

with AsyncInsertTweet.Store

with ReplicatedInsertTweet.Store

with DeleteTweet.Store

with AsyncDeleteTweet.Store

with ReplicatedDeleteTweet.Store

with UndeleteTweet.Store

with ReplicatedUndeleteTweet.Store

with AsyncIncrFavCount.Store

with ReplicatedIncrFavCount.Store

with AsyncIncrBookmarkCount.Store

with ReplicatedIncrBookmarkCount.Store

with AsyncSetRetweetVisibility.Store

with ReplicatedSetRetweetVisibility.Store

with Flush.Store {

def wrap(w: TweetStore.Wrap): TweetCountsCacheUpdatingStore = {

new TweetStoreWrapper(w, this)

with TweetCountsCacheUpdatingStore

with InsertTweet.StoreWrapper

with AsyncInsertTweet.StoreWrapper

with ReplicatedInsertTweet.StoreWrapper

with DeleteTweet.StoreWrapper

with AsyncDeleteTweet.StoreWrapper

with ReplicatedDeleteTweet.StoreWrapper

with UndeleteTweet.StoreWrapper

with ReplicatedUndeleteTweet.StoreWrapper

with AsyncIncrFavCount.StoreWrapper

with ReplicatedIncrFavCount.StoreWrapper

with AsyncIncrBookmarkCount.StoreWrapper

with ReplicatedIncrBookmarkCount.StoreWrapper

with AsyncSetRetweetVisibility.StoreWrapper

with ReplicatedSetRetweetVisibility.StoreWrapper

with Flush.StoreWrapper

}

}

/\*\*

\* An implementation of TweetStore that updates tweet-specific counts in

\* the CountsCache.

\*/

object TweetCountsCacheUpdatingStore {

private type Action = TweetCountKey => Future[Unit]

def keys(tweetId: TweetId): Seq[TweetCountKey] =

Seq(

RetweetsKey(tweetId),

RepliesKey(tweetId),

FavsKey(tweetId),

QuotesKey(tweetId),

BookmarksKey(tweetId))

def relatedKeys(tweet: Tweet): Seq[TweetCountKey] =

Seq(

getReply(tweet).flatMap(\_.inReplyToStatusId).map(RepliesKey(\_)),

getQuotedTweet(tweet).map(quotedTweet => QuotesKey(quotedTweet.tweetId)),

getShare(tweet).map(share => RetweetsKey(share.sourceStatusId))

).flatten

// pick all keys except quotes key

def relatedKeysWithoutQuotesKey(tweet: Tweet): Seq[TweetCountKey] =

relatedKeys(tweet).filterNot(\_.isInstanceOf[QuotesKey])

def apply(countsStore: CachedCountsStore): TweetCountsCacheUpdatingStore = {

val incr: Action = key => countsStore.incr(key, 1)

val decr: Action = key => countsStore.incr(key, -1)

val init: Action = key => countsStore.add(key, 0)

val delete: Action = key => countsStore.delete(key)

def initCounts(tweetId: TweetId) = Future.join(keys(tweetId).map(init))

def incrRelatedCounts(tweet: Tweet, excludeQuotesKey: Boolean = false) = {

Future.join {

if (excludeQuotesKey) {

relatedKeysWithoutQuotesKey(tweet).map(incr)

} else {

relatedKeys(tweet).map(incr)

}

}

}

def deleteCounts(tweetId: TweetId) = Future.join(keys(tweetId).map(delete))

// Decrement all the counters if is the last quote, otherwise avoid decrementing quote counters

def decrRelatedCounts(tweet: Tweet, isLastQuoteOfQuoter: Boolean = false) = {

Future.join {

if (isLastQuoteOfQuoter) {

relatedKeys(tweet).map(decr)

} else {

relatedKeysWithoutQuotesKey(tweet).map(decr)

}

}

}

def updateFavCount(tweetId: TweetId, delta: Int) =

countsStore.incr(FavsKey(tweetId), delta).unit

def updateBookmarkCount(tweetId: TweetId, delta: Int) =

countsStore.incr(BookmarksKey(tweetId), delta).unit

// these are use specifically for setRetweetVisibility

def incrRetweetCount(tweetId: TweetId) = incr(RetweetsKey(tweetId))

def decrRetweetCount(tweetId: TweetId) = decr(RetweetsKey(tweetId))

new TweetCountsCacheUpdatingStore {

override val insertTweet: FutureEffect[InsertTweet.Event] =

FutureEffect[InsertTweet.Event](e => initCounts(e.tweet.id))

override val asyncInsertTweet: FutureEffect[AsyncInsertTweet.Event] =

FutureEffect[AsyncInsertTweet.Event] { e =>

incrRelatedCounts(e.cachedTweet.tweet, e.quoterHasAlreadyQuotedTweet)

}

override val retryAsyncInsertTweet: FutureEffect[

TweetStoreRetryEvent[AsyncInsertTweet.Event]

] =

FutureEffect.unit[TweetStoreRetryEvent[AsyncInsertTweet.Event]]

override val replicatedInsertTweet: FutureEffect[ReplicatedInsertTweet.Event] =

FutureEffect[ReplicatedInsertTweet.Event] { e =>

Future

.join(

initCounts(e.tweet.id),

incrRelatedCounts(e.tweet, e.quoterHasAlreadyQuotedTweet)).unit

}

override val deleteTweet: FutureEffect[DeleteTweet.Event] =

FutureEffect[DeleteTweet.Event](e => deleteCounts(e.tweet.id))

override val asyncDeleteTweet: FutureEffect[AsyncDeleteTweet.Event] =

FutureEffect[AsyncDeleteTweet.Event](e => decrRelatedCounts(e.tweet, e.isLastQuoteOfQuoter))

override val retryAsyncDeleteTweet: FutureEffect[

TweetStoreRetryEvent[AsyncDeleteTweet.Event]

] =

FutureEffect.unit[TweetStoreRetryEvent[AsyncDeleteTweet.Event]]

override val replicatedDeleteTweet: FutureEffect[ReplicatedDeleteTweet.Event] =

FutureEffect[ReplicatedDeleteTweet.Event] { e =>

Future

.join(deleteCounts(e.tweet.id), decrRelatedCounts(e.tweet, e.isLastQuoteOfQuoter)).unit

}

override val undeleteTweet: FutureEffect[UndeleteTweet.Event] =

FutureEffect[UndeleteTweet.Event] { e =>

incrRelatedCounts(e.tweet, e.quoterHasAlreadyQuotedTweet)

}

override val replicatedUndeleteTweet: FutureEffect[ReplicatedUndeleteTweet.Event] =

FutureEffect[ReplicatedUndeleteTweet.Event] { e =>

incrRelatedCounts(e.tweet, e.quoterHasAlreadyQuotedTweet)

}

override val asyncIncrFavCount: FutureEffect[AsyncIncrFavCount.Event] =

FutureEffect[AsyncIncrFavCount.Event](e => updateFavCount(e.tweetId, e.delta))

override val replicatedIncrFavCount: FutureEffect[ReplicatedIncrFavCount.Event] =

FutureEffect[ReplicatedIncrFavCount.Event](e => updateFavCount(e.tweetId, e.delta))

override val asyncIncrBookmarkCount: FutureEffect[AsyncIncrBookmarkCount.Event] =

FutureEffect[AsyncIncrBookmarkCount.Event](e => updateBookmarkCount(e.tweetId, e.delta))

override val replicatedIncrBookmarkCount: FutureEffect[ReplicatedIncrBookmarkCount.Event] =

FutureEffect[ReplicatedIncrBookmarkCount.Event] { e =>

updateBookmarkCount(e.tweetId, e.delta)

}

override val asyncSetRetweetVisibility: FutureEffect[AsyncSetRetweetVisibility.Event] =

FutureEffect[AsyncSetRetweetVisibility.Event] { e =>

if (e.visible) incrRetweetCount(e.srcId) else decrRetweetCount(e.srcId)

}

override val retryAsyncSetRetweetVisibility: FutureEffect[

TweetStoreRetryEvent[AsyncSetRetweetVisibility.Event]

] =

FutureEffect.unit[TweetStoreRetryEvent[AsyncSetRetweetVisibility.Event]]

override val replicatedSetRetweetVisibility: FutureEffect[

ReplicatedSetRetweetVisibility.Event

] =

FutureEffect[ReplicatedSetRetweetVisibility.Event] { e =>

if (e.visible) incrRetweetCount(e.srcId) else decrRetweetCount(e.srcId)

}

override val flush: FutureEffect[Flush.Event] =

FutureEffect[Flush.Event] { e => Future.collect(e.tweetIds.map(deleteCounts)).unit }

.onlyIf(\_.flushCounts)

}

}

}

/\*\*

\* A simple trait around the cache operations needed by TweetCountsCacheUpdatingStore.

\*/

trait CachedCountsStore {

def add(key: TweetCountKey, count: Count): Future[Unit]

def delete(key: TweetCountKey): Future[Unit]

def incr(key: TweetCountKey, delta: Count): Future[Unit]

}

object CachedCountsStore {

def fromLockingCache(cache: LockingCache[TweetCountKey, Cached[Count]]): CachedCountsStore =

new CachedCountsStore {

def add(key: TweetCountKey, count: Count): Future[Unit] =

cache.add(key, toCached(count)).unit

def delete(key: TweetCountKey): Future[Unit] =

cache.delete(key).unit

def incr(key: TweetCountKey, delta: Count): Future[Unit] =

cache.lockAndSet(key, IncrDecrHandler(delta)).unit

}

def toCached(count: Count): Cached[Count] = {

val now = Time.now

Cached(Some(count), CachedValueStatus.Found, now, Some(now))

}

case class IncrDecrHandler(delta: Long) extends Handler[Cached[Count]] {

override def apply(inCache: Option[Cached[Count]]): Option[Cached[Count]] =

inCache.flatMap(incrCount)

private[this] def incrCount(oldCached: Cached[Count]): Option[Cached[Count]] = {

oldCached.value.map { oldCount => oldCached.copy(value = Some(saferIncr(oldCount))) }

}

private[this] def saferIncr(value: Long) = math.max(0, value + delta)

override lazy val toString: String = "IncrDecrHandler(%s)".format(delta)

}

object QueueIsFullException extends Exception

}

/\*\*

\* An implementation of CachedCountsStore that can queue and aggregate multiple incr

\* updates to the same key together. Currently, updates for a key only start to aggregate

\* after there is a failure to incr on the underlying store, which often indicates contention

\* due to a high level of updates. After a failure, a key is promoted into a "tracked" state,

\* and subsequent updates are aggregated together. Periodically, the aggregated updates will

\* be flushed. If the flush for a key succeeds and no more updates have come in during the flush,

\* then the key is demoted out of the tracked state. Otherwise, updates continue to aggregate

\* until the next flush attempt.

\*/

class AggregatingCachedCountsStore(

underlying: CachedCountsStore,

timer: Timer,

flushInterval: Duration,

maxSize: Int,

stats: StatsReceiver)

extends CachedCountsStore

with Serialized {

private[this] val pendingUpdates: mutable.Map[TweetCountKey, Count] =

new mutable.HashMap[TweetCountKey, Count]

private[this] var trackingCount: Int = 0

private[this] val promotionCounter = stats.counter("promotions")

private[this] val demotionCounter = stats.counter("demotions")

private[this] val updateCounter = stats.counter("aggregated\_updates")

private[this] val overflowCounter = stats.counter("overflows")

private[this] val flushFailureCounter = stats.counter("flush\_failures")

private[this] val trackingCountGauge = stats.addGauge("tracking")(trackingCount.toFloat)

timer.schedule(flushInterval) { flush() }

def add(key: TweetCountKey, count: Count): Future[Unit] =

underlying.add(key, count)

def delete(key: TweetCountKey): Future[Unit] =

underlying.delete(key)

def incr(key: TweetCountKey, delta: Count): Future[Unit] =

aggregateIfTracked(key, delta).flatMap {

case true => Future.Unit

case false =>

underlying

.incr(key, delta)

.rescue { case \_ => aggregate(key, delta) }

}

/\*\*

\* Queues an update to be aggregated and applied to a key at a later time, but only if we are

\* already aggregating updates for the key.

\*

\* @return true the delta was aggregated, false if the key is not being tracked

\* and the incr should be attempted directly.

\*/

private[this] def aggregateIfTracked(key: TweetCountKey, delta: Count): Future[Boolean] =

serialized {

pendingUpdates.get(key) match {

case None => false

case Some(current) =>

updateCounter.incr()

pendingUpdates(key) = current + delta

true

}

}

/\*\*

\* Queues an update to be aggregated and applied to a key at a later time.

\*/

private[this] def aggregate(key: TweetCountKey, delta: Count): Future[Unit] =

serialized {

val alreadyTracked = pendingUpdates.contains(key)

if (!alreadyTracked) {

if (pendingUpdates.size < maxSize)

promotionCounter.incr()

else {

overflowCounter.incr()

throw CachedCountsStore.QueueIsFullException

}

}

(pendingUpdates.get(key).getOrElse(0L) + delta) match {

case 0 =>

pendingUpdates.remove(key)

demotionCounter.incr()

case aggregatedDelta =>

pendingUpdates(key) = aggregatedDelta

}

trackingCount = pendingUpdates.size

}

private[this] def flush(): Future[Unit] = {

for {

// make a copy of the updates to flush, so that updates can continue to be queued

// while the flush is in progress. if an individual flush succeeds, then we

// go back and update pendingUpdates.

updates <- serialized { pendingUpdates.toSeq.toList }

() <- Future.join(for ((key, delta) <- updates) yield flush(key, delta))

} yield ()

}

private[this] def flush(key: TweetCountKey, delta: Count): Future[Unit] =

underlying

.incr(key, delta)

.flatMap(\_ => aggregate(key, -delta))

.handle { case ex => flushFailureCounter.incr() }

}