package com.twitter.tweetypie.storage

import com.twitter.conversions.DurationOps.\_

import com.twitter.finagle.stats.Counter

import com.twitter.finagle.stats.NullStatsReceiver

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.logging.Logger

import com.twitter.snowflake.id.SnowflakeId

import com.twitter.stitch.Stitch

import com.twitter.stitch.StitchSeqGroup

import com.twitter.storage.client.manhattan.kv.DeniedManhattanException

import com.twitter.storage.client.manhattan.kv.ManhattanException

import com.twitter.tweetypie.storage.TweetStateRecord.BounceDeleted

import com.twitter.tweetypie.storage.TweetStateRecord.HardDeleted

import com.twitter.tweetypie.storage.TweetStateRecord.SoftDeleted

import com.twitter.tweetypie.storage.TweetStorageClient.GetTweet

import com.twitter.tweetypie.storage.TweetUtils.\_

import com.twitter.util.Duration

import com.twitter.util.Return

import com.twitter.util.Throw

import com.twitter.util.Time

object GetTweetHandler {

private[this] val logger = Logger(getClass)

//////////////////////////////////////////////////

// Logging racy reads for later validation.

val RacyTweetWindow: Duration = 10.seconds

/\*\*

\* If this read is soon after the tweet was created, then we would usually

\* expect it to be served from cache. This early read indicates that this

\* tweet is prone to consistency issues, so we log what's present in

\* Manhattan at the time of the read for later analysis.

\*/

private[this] def logRacyRead(tweetId: TweetId, records: Seq[TweetManhattanRecord]): Unit =

if (SnowflakeId.isSnowflakeId(tweetId)) {

val tweetAge = Time.now.since(SnowflakeId(tweetId).time)

if (tweetAge <= RacyTweetWindow) {

val sb = new StringBuilder

sb.append("racy\_tweet\_read\t")

.append(tweetId)

.append('\t')

.append(tweetAge.inMilliseconds) // Log the age for analysis purposes

records.foreach { rec =>

sb.append('\t')

.append(rec.lkey)

rec.value.timestamp.foreach { ts =>

// If there is a timestamp for this key, log it so that we can tell

// later on whether a value should have been present. We expect

// keys written in a single write to have the same timestamp, and

// generally, keys written in separate writes will have different

// timestamps. The timestamp value is optional in Manhattan, but

// we expect there to always be a value for the timestamp.

sb.append(':')

.append(ts.inMilliseconds)

}

}

logger.info(sb.toString)

}

}

/\*\*

\* Convert a set of records from Manhattan into a GetTweet.Response.

\*/

def tweetResponseFromRecords(

tweetId: TweetId,

mhRecords: Seq[TweetManhattanRecord],

statsReceiver: StatsReceiver = NullStatsReceiver

): GetTweet.Response =

if (mhRecords.isEmpty) {

GetTweet.Response.NotFound

} else {

// If no internal fields are present or no required fields present, we consider the tweet

// as not returnable (even if some additional fields are present)

def tweetFromRecords(tweetId: TweetId, mhRecords: Seq[TweetManhattanRecord]) = {

val storedTweet = buildStoredTweet(tweetId, mhRecords)

if (storedTweet.getFieldBlobs(expectedFields).nonEmpty) {

if (isValid(storedTweet)) {

statsReceiver.counter("valid").incr()

Some(StorageConversions.fromStoredTweet(storedTweet))

} else {

log.info(s"Invalid Tweet Id: $tweetId")

statsReceiver.counter("invalid").incr()

None

}

} else {

// The Tweet contained none of the fields defined in `expectedFields`

log.info(s"Expected Fields Not Present Tweet Id: $tweetId")

statsReceiver.counter("expected\_fields\_not\_present").incr()

None

}

}

val stateRecord = TweetStateRecord.mostRecent(mhRecords)

stateRecord match {

// some other cases don't require an attempt to construct a Tweet

case Some(\_: SoftDeleted) | Some(\_: HardDeleted) => GetTweet.Response.Deleted

// all other cases require an attempt to construct a Tweet, which may not be successful

case \_ =>

logRacyRead(tweetId, mhRecords)

(stateRecord, tweetFromRecords(tweetId, mhRecords)) match {

// BounceDeleted contains the Tweet data so that callers can access data on the the

// tweet (e.g. hard delete daemon requires conversationId and userId. There are no

// plans for Tweetypie server to make use of the returned tweet at this time.

case (Some(\_: BounceDeleted), Some(tweet)) => GetTweet.Response.BounceDeleted(tweet)

case (Some(\_: BounceDeleted), None) => GetTweet.Response.Deleted

case (\_, Some(tweet)) => GetTweet.Response.Found(tweet)

case \_ => GetTweet.Response.NotFound

}

}

}

def apply(read: ManhattanOperations.Read, statsReceiver: StatsReceiver): GetTweet = {

object stats {

val getTweetScope = statsReceiver.scope("getTweet")

val deniedCounter: Counter = getTweetScope.counter("mh\_denied")

val mhExceptionCounter: Counter = getTweetScope.counter("mh\_exception")

val nonFatalExceptionCounter: Counter = getTweetScope.counter("non\_fatal\_exception")

val notFoundCounter: Counter = getTweetScope.counter("not\_found")

}

object mhGroup extends StitchSeqGroup[TweetId, Seq[TweetManhattanRecord]] {

override def run(tweetIds: Seq[TweetId]): Stitch[Seq[Seq[TweetManhattanRecord]]] = {

Stats.addWidthStat("getTweet", "tweetIds", tweetIds.size, statsReceiver)

Stitch.traverse(tweetIds)(read(\_))

}

}

tweetId =>

if (tweetId <= 0) {

Stitch.NotFound

} else {

Stitch

.call(tweetId, mhGroup)

.map(mhRecords => tweetResponseFromRecords(tweetId, mhRecords, stats.getTweetScope))

.liftToTry

.map {

case Throw(mhException: DeniedManhattanException) =>

stats.deniedCounter.incr()

Throw(RateLimited("", mhException))

// Encountered some other Manhattan error

case t @ Throw(\_: ManhattanException) =>

stats.mhExceptionCounter.incr()

t

// Something else happened

case t @ Throw(ex) =>

stats.nonFatalExceptionCounter.incr()

TweetUtils.log

.warning(ex, s"Unhandled exception in GetTweetHandler for tweetId: $tweetId")

t

case r @ Return(GetTweet.Response.NotFound) =>

stats.notFoundCounter.incr()

r

case r @ Return(\_) => r

}

.lowerFromTry

}

}

}