package com.twitter.tweetypie.storage

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.stitch.Stitch

import com.twitter.tweetypie.storage.TweetKey.LKey.ForceAddedStateKey

import com.twitter.tweetypie.storage.TweetStorageClient.HardDeleteTweet

import com.twitter.tweetypie.storage.TweetStorageClient.HardDeleteTweet.Response.\_

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

import com.twitter.util.Return

import com.twitter.util.Throw

import com.twitter.util.Time

import com.twitter.util.Try

object HardDeleteTweetHandler {

/\*\*

\* When a tweet is removed lkeys with these prefixes will be deleted permanently.

\*/

private[storage] def isKeyToBeDeleted(key: TweetKey): Boolean =

key.lKey match {

case (TweetKey.LKey.CoreFieldsKey | TweetKey.LKey.InternalFieldsKey(\_) |

TweetKey.LKey.AdditionalFieldsKey(\_) | TweetKey.LKey.SoftDeletionStateKey |

TweetKey.LKey.BounceDeletionStateKey | TweetKey.LKey.UnDeletionStateKey |

TweetKey.LKey.ForceAddedStateKey) =>

true

case \_ => false

}

/\*\*

\* When hard deleting, there are two actions, writing the record and

\* removing the tweet data. If we are performing any action, we will

\* always try to remove the tweet data. If the tweet does not yet have a

\* hard deletion record, then we will need to write one. This method

\* returns the HardDeleted record if it needs to be written, and None

\* if it has already been written.

\*

\* If the tweet is not in a deleted state we signal this with a

\* Throw(NotDeleted).

\*/

private[storage] def getHardDeleteStateRecord(

tweetId: TweetId,

records: Seq[TweetManhattanRecord],

mhTimestamp: Time,

stats: StatsReceiver

): Try[Option[TweetStateRecord.HardDeleted]] = {

val mostRecent = TweetStateRecord.mostRecent(records)

val currentStateStr = mostRecent.map(\_.name).getOrElse("no\_tweet\_state\_record")

stats.counter(currentStateStr).incr()

mostRecent match {

case Some(

record @ (TweetStateRecord.SoftDeleted(\_, \_) | TweetStateRecord.BounceDeleted(\_, \_))) =>

Return(

Some(

TweetStateRecord.HardDeleted(

tweetId = tweetId,

// createdAt is the hard deletion timestamp when dealing with hard deletes in Manhattan

createdAt = mhTimestamp.inMillis,

// deletedAt is the soft deletion timestamp when dealing with hard deletes in Manhattan

deletedAt = record.createdAt

)

)

)

case Some(\_: TweetStateRecord.HardDeleted) =>

Return(None)

case Some(\_: TweetStateRecord.ForceAdded) =>

Throw(NotDeleted(tweetId, Some(ForceAddedStateKey)))

case Some(\_: TweetStateRecord.Undeleted) =>

Throw(NotDeleted(tweetId, Some(TweetKey.LKey.UnDeletionStateKey)))

case None =>

Throw(NotDeleted(tweetId, None))

}

}

/\*\*

\* This handler returns HardDeleteTweet.Response.Deleted if data associated with the tweet is deleted,

\* either as a result of this request or a previous one.

\*

\* The most recently added record determines the tweet's state. This method will only delete data

\* for tweets in the soft-delete or hard-delete state. (Calling hardDeleteTweet for tweets that have

\* already been hard-deleted will remove any lkeys that may not have been deleted previously).

\*/

def apply(

read: ManhattanOperations.Read,

insert: ManhattanOperations.Insert,

delete: ManhattanOperations.Delete,

scribe: Scribe,

stats: StatsReceiver

): TweetId => Stitch[HardDeleteTweet.Response] = {

val hardDeleteStats = stats.scope("hardDeleteTweet")

val hardDeleteTweetCancelled = hardDeleteStats.counter("cancelled")

val beforeStateStats = hardDeleteStats.scope("before\_state")

def removeRecords(keys: Seq[TweetKey], mhTimestamp: Time): Stitch[Unit] =

Stitch

.collect(keys.map(key => delete(key, Some(mhTimestamp)).liftToTry))

.map(collectWithRateLimitCheck)

.lowerFromTry

def writeRecord(record: Option[TweetStateRecord.HardDeleted]): Stitch[Unit] =

record match {

case Some(r) =>

insert(r.toTweetMhRecord).onSuccess { \_ =>

scribe.logRemoved(

r.tweetId,

Time.fromMilliseconds(r.createdAt),

isSoftDeleted = false

)

}

case None => Stitch.Unit

}

tweetId =>

read(tweetId)

.flatMap { records =>

val hardDeletionTimestamp = Time.now

val keysToBeDeleted: Seq[TweetKey] = records.map(\_.key).filter(isKeyToBeDeleted)

getHardDeleteStateRecord(

tweetId,

records,

hardDeletionTimestamp,

beforeStateStats) match {

case Return(record) =>

Stitch

.join(

writeRecord(record),

removeRecords(keysToBeDeleted, hardDeletionTimestamp)

).map(\_ =>

// If the tweetId is non-snowflake and has previously been hard deleted

// there will be no coreData record to fall back on to get the tweet

// creation time and createdAtMillis will be None.

Deleted(

// deletedAtMillis: when the tweet was hard deleted

deletedAtMillis = Some(hardDeletionTimestamp.inMillis),

// createdAtMillis: when the tweet itself was created

// (as opposed to when the deletion record was created)

createdAtMillis =

TweetUtils.creationTimeFromTweetIdOrMHRecords(tweetId, records)

))

case Throw(notDeleted: NotDeleted) =>

hardDeleteTweetCancelled.incr()

Stitch.value(notDeleted)

case Throw(e) => Stitch.exception(e) // this should never happen

}

}

}

}