package com.twitter.tweetypie.storage

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.stitch.Stitch

import com.twitter.tweetypie.storage.TweetStorageClient.Undelete

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

import com.twitter.util.Time

object UndeleteHandler {

def apply(

read: ManhattanOperations.Read,

localInsert: ManhattanOperations.Insert,

remoteInsert: ManhattanOperations.Insert,

delete: ManhattanOperations.Delete,

undeleteWindowHours: Int,

stats: StatsReceiver

): Undelete = {

def withinUndeleteWindow(timestampMs: Long) =

(Time.now - Time.fromMilliseconds(timestampMs)).inHours < undeleteWindowHours

def prepareUndelete(

tweetId: TweetId,

records: Seq[TweetManhattanRecord]

): (Undelete.Response, Option[TweetManhattanRecord]) = {

val undeleteRecord =

Some(TweetStateRecord.Undeleted(tweetId, Time.now.inMillis).toTweetMhRecord)

TweetStateRecord.mostRecent(records) match {

// check if we need to undo a soft deletion

case Some(TweetStateRecord.SoftDeleted(\_, createdAt)) =>

if (createdAt > 0) {

if (withinUndeleteWindow(createdAt)) {

(

mkSuccessfulUndeleteResponse(tweetId, records, Some(createdAt)),

undeleteRecord

)

} else {

(Undelete.Response(Undelete.UndeleteResponseCode.BackupNotFound), None)

}

} else {

throw InternalError(s"Timestamp unavailable for $tweetId")

}

// BounceDeleted tweets may not be undeleted. see go/bouncedtweet

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

(Undelete.Response(Undelete.UndeleteResponseCode.BackupNotFound), None)

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

// We still want to write the undelete record, because at this point, we only know that the local DC's

// winning record is not a soft/hard deletion record, while its possible that the remote DC's winning

// record might still be a soft deletion record. Having said that, we don't want to set it to true

// if the winning record is forceAdd, as the forceAdd call should have ensured that both DCs had the

// forceAdd record.

(mkSuccessfulUndeleteResponse(tweetId, records), undeleteRecord)

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

(mkSuccessfulUndeleteResponse(tweetId, records), None)

// lets write the undeletion record just in case there is a softdeletion record in flight

case None => (mkSuccessfulUndeleteResponse(tweetId, records), undeleteRecord)

}

}

// Write the undelete record both locally and remotely to protect

// against races with hard delete replication. We only need this

// protection for the insertion of the undelete record.

def multiInsert(record: TweetManhattanRecord): Stitch[Unit] =

Stitch

.collect(

Seq(

localInsert(record).liftToTry,

remoteInsert(record).liftToTry

)

)

.map(collectWithRateLimitCheck)

.lowerFromTry

def deleteSoftDeleteRecord(tweetId: TweetId): Stitch[Unit] = {

val mhKey = TweetKey.softDeletionStateKey(tweetId)

delete(mhKey, None)

}

tweetId =>

for {

records <- read(tweetId)

(response, undeleteRecord) = prepareUndelete(tweetId, records)

\_ <- Stitch.collect(undeleteRecord.map(multiInsert)).unit

\_ <- deleteSoftDeleteRecord(tweetId)

} yield {

response

}

}

private[storage] def mkSuccessfulUndeleteResponse(

tweetId: TweetId,

records: Seq[TweetManhattanRecord],

timestampOpt: Option[Long] = None

) =

Undelete.Response(

Undelete.UndeleteResponseCode.Success,

Some(

StorageConversions.fromStoredTweet(buildStoredTweet(tweetId, records))

),

archivedAtMillis = timestampOpt

)

}