package com.twitter.tweetypie.storage

import com.twitter.logging.Logger

import com.twitter.scrooge.TFieldBlob

import com.twitter.snowflake.id.SnowflakeId

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

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

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

import com.twitter.tweetypie.storage\_internal.thriftscala.StoredTweet

import com.twitter.util.Return

import com.twitter.util.Throw

import com.twitter.util.Try

object TweetUtils {

val log: Logger = Logger("com.twitter.tweetypie.storage.TweetStorageLibrary")

import FieldResponseCodec.ValueNotFoundException

/\*\*

\* It's rare, but we have seen tweets with userId=0, which is likely the result of a

\* failed/partial delete. Treat these as invalid tweets, which are returned to callers

\* as not found.

\*/

def isValid(tweet: StoredTweet): Boolean =

tweet.userId.exists(\_ != 0) && tweet.text.nonEmpty &&

tweet.createdVia.nonEmpty && tweet.createdAtSec.nonEmpty

/\*\*

\* Helper function to extract Scrubbed field Ids from the result returned by reading entire tweet prefix

\* function.

\*

\* @param records The sequence of MH records for the given tweetId

\*

\* @return The set of scrubbed field ids

\*/

private[tweetypie] def extractScrubbedFields(records: Seq[TweetManhattanRecord]): Set[Short] =

records

.map(r => r.lkey)

.collect { case TweetKey.LKey.ScrubbedFieldKey(fieldId) => fieldId }

.toSet

private[tweetypie] val expectedFields =

TweetFields.requiredFieldIds.toSet - TweetFields.tweetIdField

/\*\*

\* Find the timestamp from a tweetId and a list of MH records. This is used when

\* you need a timestamp and you aren't sure that tweetId is a snowflake id.

\*

\* @param tweetId A tweetId you want the timestamp for.

\* @param records Tbird\_mh records keyed on tweetId, one of which should be the

\* core fields record.

\* @return A milliseconds timestamp if one could be found.

\*/

private[tweetypie] def creationTimeFromTweetIdOrMHRecords(

tweetId: Long,

records: Seq[TweetManhattanRecord]

): Option[Long] =

SnowflakeId

.unixTimeMillisOptFromId(tweetId).orElse({

records

.find(\_.lkey == TweetKey.LKey.CoreFieldsKey)

.flatMap { coreFields =>

CoreFieldsCodec

.fromTFieldBlob(

TFieldBlobCodec.fromByteBuffer(coreFields.value.contents)

).createdAtSec.map(seconds => seconds \* 1000)

}

})

/\*\*

\* Helper function used to parse manhattan results for fields in a tweet (given in the form of

\* Sequence of (FieldKey, Try[Unit]) pairs) and build a TweetResponse object.

\*

\* @param callerName The name of the caller function. Used for error messages

\* @param tweetId Id of the Tweet for which TweetResponse is being built

\* @param fieldResults Sequence of (FieldKey, Try[Unit]).

\*

\* @return TweetResponse object

\*/

private[tweetypie] def buildTweetResponse(

callerName: String,

tweetId: Long,

fieldResults: Map[FieldId, Try[Unit]]

): TweetResponse = {

// Count Found/Not Found

val successCount =

fieldResults.foldLeft(0) {

case (count, (\_, Return(\_))) => count + 1

case (count, (\_, Throw(\_: ValueNotFoundException))) => count + 1

case (count, \_) => count

}

val fieldResponsesMap = getFieldResponses(callerName, tweetId, fieldResults)

val overallCode = if (successCount > 0 && successCount == fieldResults.size) {

TweetResponseCode.Success

} else {

// If any field was rate limited, then we consider the entire tweet to be rate limited. So first we scan

// the field results to check such an occurrence.

val wasRateLimited = fieldResults.exists { fieldResult =>

fieldResult.\_2 match {

case Throw(e: DeniedManhattanException) => true

case \_ => false

}

}

// Were we rate limited for any of the additional fields?

if (wasRateLimited) {

TweetResponseCode.OverCapacity

} else if (successCount == 0) {

// successCount is < fieldResults.size at this point. So if allOrNone is true or

// if successCount == 0 (i.e failed on all Fields), the overall code should be 'Failure'

TweetResponseCode.Failure

} else {

// allOrNone == false AND successCount > 0 at this point. Clearly the overallCode should be Partial

TweetResponseCode.Partial

}

}

TweetResponse(tweetId, overallCode, Some(fieldResponsesMap))

}

/\*\*

\* Helper function to convert manhattan results into a Map[FieldId, FieldResponse]

\*

\* @param fieldResults Sequence of (TweetKey, TFieldBlob).

\*/

private[tweetypie] def getFieldResponses(

callerName: String,

tweetId: TweetId,

fieldResults: Map[FieldId, Try[\_]]

): Map[FieldId, FieldResponse] =

fieldResults.map {

case (fieldId, resp) =>

def keyStr = TweetKey.fieldKey(tweetId, fieldId).toString

resp match {

case Return(\_) =>

fieldId -> FieldResponse(FieldResponseCode.Success, None)

case Throw(mhException: ManhattanException) =>

val errMsg = s"Exception in $callerName. Key: $keyStr. Error: $mhException"

mhException match {

case \_: ValueNotFoundException => // ValueNotFound is not an error

case \_ => log.error(errMsg)

}

fieldId -> FieldResponseCodec.fromThrowable(mhException, Some(errMsg))

case Throw(e) =>

val errMsg = s"Exception in $callerName. Key: $keyStr. Error: $e"

log.error(errMsg)

fieldId -> FieldResponse(FieldResponseCode.Error, Some(errMsg))

}

}

/\*\*

\* Helper function to build a TweetResponse object when being rate limited. Its possible that only some of the fields

\* got rate limited, so we indicate which fields got processed successfully, and which encountered some sort of error.

\*

\* @param tweetId Tweet id

\* @param callerName name of API calling this function

\* @param fieldResponses field responses for the case where

\*

\* @return The TweetResponse object

\*/

private[tweetypie] def buildTweetOverCapacityResponse(

callerName: String,

tweetId: Long,

fieldResponses: Map[FieldId, Try[Unit]]

) = {

val fieldResponsesMap = getFieldResponses(callerName, tweetId, fieldResponses)

TweetResponse(tweetId, TweetResponseCode.OverCapacity, Some(fieldResponsesMap))

}

/\*\*

\* Build a StoredTweet from a Seq of records. Core fields are handled specially.

\*/

private[tweetypie] def buildStoredTweet(

tweetId: TweetId,

records: Seq[TweetManhattanRecord],

includeScrubbed: Boolean = false,

): StoredTweet = {

getStoredTweetBlobs(records, includeScrubbed)

.flatMap { fieldBlob =>

// When fieldId == TweetFields.rootCoreFieldId, we have further work to do since the

// 'value' is really serialized/packed version of all core fields. In this case we'll have

// to unpack it into many TFieldBlobs.

if (fieldBlob.id == TweetFields.rootCoreFieldId) {

// We won't throw any error in this function and instead let the caller function handle this

// condition (i.e If the caller function does not find any values for the core-fields in

// the returned map, it should assume that the tweet is not found)

CoreFieldsCodec.unpackFields(fieldBlob).values.toSeq

} else {

Seq(fieldBlob)

}

}.foldLeft(StoredTweet(tweetId))(\_.setField(\_))

}

private[tweetypie] def buildValidStoredTweet(

tweetId: TweetId,

records: Seq[TweetManhattanRecord]

): Option[StoredTweet] = {

val storedTweet = buildStoredTweet(tweetId, records)

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

Some(storedTweet)

} else {

None

}

}

/\*\*

\* Return a TFieldBlob for each StoredTweet field defined in this set of records.

\* @param includeScrubbed when false, result will not include scrubbed fields even

\* if the data is present in the set of records.

\*/

private[tweetypie] def getStoredTweetBlobs(

records: Seq[TweetManhattanRecord],

includeScrubbed: Boolean = false,

): Seq[TFieldBlob] = {

val scrubbed = extractScrubbedFields(records)

records

.flatMap { r =>

// extract LKey.FieldKey records if they are not scrubbed and get their TFieldBlobs

r.key match {

case fullKey @ TweetKey(\_, key: TweetKey.LKey.FieldKey)

if includeScrubbed || !scrubbed.contains(key.fieldId) =>

try {

val fieldBlob = TFieldBlobCodec.fromByteBuffer(r.value.contents)

if (fieldBlob.field.id != key.fieldId) {

throw new AssertionError(

s"Blob stored for $fullKey has unexpected id ${fieldBlob.field.id}"

)

}

Some(fieldBlob)

} catch {

case e: VersionMismatchError =>

log.error(

s"Failed to decode bytebuffer for $fullKey: ${e.getMessage}"

)

throw e

}

case \_ => None

}

}

}

/\*\*

\* Its important to bubble up rate limiting exceptions as they would likely be the root cause for other issues

\* (timeouts etc.), so we scan for this particular exception, and if found, we bubble that up specifically

\*

\* @param seqOfTries The sequence of tries which may contain within it a rate limit exception

\*

\* @return if a rate limiting exn was detected, this will be a Throw(e: DeniedManhattanException)

\* otherwise it will be a Return(\_) only if all individual tries succeeded

\*/

private[tweetypie] def collectWithRateLimitCheck(seqOfTries: Seq[Try[Unit]]): Try[Unit] = {

val rateLimitThrowOpt = seqOfTries.find {

case Throw(e: DeniedManhattanException) => true

case \_ => false

}

rateLimitThrowOpt.getOrElse(

Try.collect(seqOfTries).map(\_ => ())

) // Operation is considered successful only if all the deletions are successful

}

}