package com.twitter.tweetypie

package repository

import com.twitter.flockdb.client.\_

import com.twitter.stitch.SeqGroup

import com.twitter.stitch.Stitch

import com.twitter.stitch.compat.LegacySeqGroup

case class ConversationIdKey(tweetId: TweetId, parentId: TweetId)

object ConversationIdRepository {

type Type = ConversationIdKey => Stitch[TweetId]

def apply(multiSelectOne: Iterable[Select[StatusGraph]] => Future[Seq[Option[Long]]]): Type =

key => Stitch.call(key, Group(multiSelectOne))

private case class Group(

multiSelectOne: Iterable[Select[StatusGraph]] => Future[Seq[Option[Long]]])

extends SeqGroup[ConversationIdKey, TweetId] {

private[this] def getConversationIds(

keys: Seq[ConversationIdKey],

getLookupId: ConversationIdKey => TweetId

): Future[Map[ConversationIdKey, TweetId]] = {

val distinctIds = keys.map(getLookupId).distinct

val tflockQueries = distinctIds.map(ConversationGraph.to)

if (tflockQueries.isEmpty) {

Future.value(Map[ConversationIdKey, TweetId]())

} else {

multiSelectOne(tflockQueries).map { results =>

// first, we need to match up the distinct ids requested with the corresponding result

val resultMap =

distinctIds

.zip(results)

.collect {

case (id, Some(conversationId)) => id -> conversationId

}

.toMap

// then we need to map keys into the above map

keys.flatMap { key => resultMap.get(getLookupId(key)).map(key -> \_) }.toMap

}

}

}

/\*\*

\* Returns a key-value result that maps keys to the tweet's conversation IDs.

\*

\* Example:

\* Tweet B is a reply to tweet A with conversation ID c.

\* We want to get B's conversation ID. Then, for the request

\*

\* ConversationIdRequest(B.id, A.id)

\*

\* our key-value result's "found" map will contain a pair (B.id -> c).

\*/

protected override def run(keys: Seq[ConversationIdKey]): Future[Seq[Try[TweetId]]] =

LegacySeqGroup.liftToSeqTry(

for {

// Try to get the conversation IDs for the parent tweets

convIdsFromParent <- getConversationIds(keys, \_.parentId)

// Collect the tweet IDs whose parents' conversation IDs couldn't be found.

// We assume that happened in one of two cases:

// \* for a tweet whose parent has been deleted

// \* for a tweet whose parent is the root of a conversation

// Note: In either case, we will try to look up the conversation ID of the tweet whose parent

// couldn't be found. If that can't be found either, we will eventually return the parent ID.

tweetsWhoseParentsDontHaveConvoIds = keys.toSet -- convIdsFromParent.keys

// Collect the conversation IDs for the tweets whose parents have not been found, now using the

// tweets' own IDs.

convIdsFromTweet <-

getConversationIds(tweetsWhoseParentsDontHaveConvoIds.toSeq, \_.tweetId)

// Combine the by-parent-ID and by-tweet-ID results.

convIdMap = convIdsFromParent ++ convIdsFromTweet

// Assign conversation IDs to all not-found tweet IDs.

// A tweet might not have received a conversation ID if

// \* the parent of the tweet is the root of the conversation, and we are in the write path

// for creating the tweet. In that case, the conversation ID should be the tweet's parent

// ID.

// \* it had been created before TFlock started handling conversation IDs. In that case, the

// conversation ID will just point to the parent tweet so that we can have a conversation of

// at least two tweets.

// So in both cases, we want to return the tweet's parent ID.

} yield {

keys.map {

case k @ ConversationIdKey(t, p) => convIdMap.getOrElse(k, p)

}

}

)

}

}