/\*\* Copyright 2010 Twitter, Inc. \*/

package com.twitter.tweetypie

package tflock

import com.twitter.finagle.stats.Counter

import com.twitter.flockdb.client.\_

import com.twitter.flockdb.client.thriftscala.Priority

import com.twitter.snowflake.id.SnowflakeId

import com.twitter.tweetypie.serverutil.StoredCard

import com.twitter.tweetypie.thriftscala.\_

import com.twitter.util.Future

import scala.collection.mutable.ListBuffer

object TFlockIndexer {

/\*\*

\* Printable names for some edge types currently defined in [[com.twitter.flockdb.client]].

\* Used to defined stats counters for adding edges.

\*/

val graphNames: Map[Int, String] =

Map(

CardTweetsGraph.id -> "card\_tweets",

ConversationGraph.id -> "conversation",

DirectedAtUserIdGraph.id -> "directed\_at\_user\_id",

InvitedUsersGraph.id -> "invited\_users",

MediaTimelineGraph.id -> "media\_timeline",

MentionsGraph.id -> "mentions",

NarrowcastSentTweetsGraph.id -> "narrowcast\_sent\_tweets",

NullcastedTweetsGraph.id -> "nullcasted\_tweets",

QuotersGraph.id -> "quoters",

QuotesGraph.id -> "quotes",

QuoteTweetsIndexGraph.id -> "quote\_tweets\_index",

RepliesToTweetsGraph.id -> "replies\_to\_tweets",

RetweetsByMeGraph.id -> "retweets\_by\_me",

RetweetsGraph.id -> "retweets",

RetweetsOfMeGraph.id -> "retweets\_of\_me",

RetweetSourceGraph.id -> "retweet\_source",

TweetsRetweetedGraph.id -> "tweets\_retweeted",

UserTimelineGraph.id -> "user\_timeline",

CreatorSubscriptionTimelineGraph.id -> "creator\_subscription\_timeline",

CreatorSubscriptionMediaTimelineGraph.id -> "creator\_subscription\_image\_timeline",

)

/\*\*

\* On edge deletion, edges are either archived permanently or retained for 3 months, based on

\* the retention policy in the above confluence page.

\*

\* These two retention policies correspond to the two deletion techniques: archive and remove.

\* We call removeEdges for edges with a short retention policy and archiveEdges for edges with

\* a permanent retention policy.

\*/

val graphsWithRemovedEdges: Seq[Int] =

Seq(

CardTweetsGraph.id,

CuratedTimelineGraph.id,

CuratedTweetsGraph.id,

DirectedAtUserIdGraph.id,

MediaTimelineGraph.id,

MutedConversationsGraph.id,

QuotersGraph.id,

QuotesGraph.id,

QuoteTweetsIndexGraph.id,

ReportedTweetsGraph.id,

RetweetsOfMeGraph.id,

RetweetSourceGraph.id,

SoftLikesGraph.id,

TweetsRetweetedGraph.id,

CreatorSubscriptionTimelineGraph.id,

CreatorSubscriptionMediaTimelineGraph.id,

)

/\*\*

\* These edges should be left in place when bounced tweets are deleted.

\* These edges are removed during hard deletion.

\*

\* This is done so external teams (timelines) can execute on these edges for

\* tombstone feature.

\*/

val bounceDeleteGraphIds: Set[Int] =

Set(

UserTimelineGraph.id,

ConversationGraph.id

)

def makeCounters(stats: StatsReceiver, operation: String): Map[Int, Counter] = {

TFlockIndexer.graphNames

.mapValues(stats.scope(\_).counter(operation))

.withDefaultValue(stats.scope("unknown").counter(operation))

}

}

/\*\*

\* @param backgroundIndexingPriority specifies the queue to use for

\* background indexing operations. This is useful for making the

\* effects of background indexing operations (such as deleting edges

\* for deleted Tweets) available sooner in testing scenarios

\* (end-to-end tests or development instances). It is set to

\* Priority.Low in production to reduce the load on high priority

\* queues that we use for prominently user-visible operations.

\*/

class TFlockIndexer(

tflock: TFlockClient,

hasMedia: Tweet => Boolean,

backgroundIndexingPriority: Priority,

stats: StatsReceiver)

extends TweetIndexer {

private[this] val FutureNil = Future.Nil

private[this] val archiveCounters = TFlockIndexer.makeCounters(stats, "archive")

private[this] val removeCounters = TFlockIndexer.makeCounters(stats, "remove")

private[this] val insertCounters = TFlockIndexer.makeCounters(stats, "insert")

private[this] val negateCounters = TFlockIndexer.makeCounters(stats, "negate")

private[this] val foregroundIndexingPriority: Priority = Priority.High

override def createIndex(tweet: Tweet): Future[Unit] =

createEdges(tweet, isUndelete = false)

override def undeleteIndex(tweet: Tweet): Future[Unit] =

createEdges(tweet, isUndelete = true)

private[this] case class PartitionedEdges(

longRetention: Seq[ExecuteEdge[StatusGraph]] = Nil,

shortRetention: Seq[ExecuteEdge[StatusGraph]] = Nil,

negate: Seq[ExecuteEdge[StatusGraph]] = Nil,

ignore: Seq[ExecuteEdge[StatusGraph]] = Nil)

private[this] def partitionEdgesForDelete(

edges: Seq[ExecuteEdge[StatusGraph]],

isBounceDelete: Boolean

) =

edges.foldLeft(PartitionedEdges()) {

// Two dependees of UserTimelineGraph edge states to satisfy: timelines & safety tools.

// Timelines show bounce-deleted tweets as tombstones; regular deletes are not shown.

// - i.e. timelineIds = UserTimelineGraph(Normal || Negative)

// Safety tools show deleted tweets to authorized internal review agents

// - i.e. deletedIds = UserTimelineGraph(Removed || Negative)

case (partitionedEdges, edge) if isBounceDelete && edge.graphId == UserTimelineGraph.id =>

partitionedEdges.copy(negate = edge +: partitionedEdges.negate)

case (partitionedEdges, edge) if isBounceDelete && edge.graphId == ConversationGraph.id =>

// Bounce-deleted tweets remain rendered as tombstones in conversations, so do not modify

// the ConversationGraph edge state

partitionedEdges.copy(ignore = edge +: partitionedEdges.ignore)

case (partitionedEdges, edge)

if TFlockIndexer.graphsWithRemovedEdges.contains(edge.graphId) =>

partitionedEdges.copy(shortRetention = edge +: partitionedEdges.shortRetention)

case (partitionedEdges, edge) =>

partitionedEdges.copy(longRetention = edge +: partitionedEdges.longRetention)

}

override def deleteIndex(tweet: Tweet, isBounceDelete: Boolean): Future[Unit] =

for {

edges <- getEdges(tweet, isCreate = false, isDelete = true, isUndelete = false)

partitionedEdges = partitionEdgesForDelete(edges, isBounceDelete)

() <-

Future

.join(

tflock

.archiveEdges(partitionedEdges.longRetention, backgroundIndexingPriority)

.onSuccess(\_ =>

partitionedEdges.longRetention.foreach(e => archiveCounters(e.graphId).incr())),

tflock

.removeEdges(partitionedEdges.shortRetention, backgroundIndexingPriority)

.onSuccess(\_ =>

partitionedEdges.shortRetention.foreach(e => removeCounters(e.graphId).incr())),

tflock

.negateEdges(partitionedEdges.negate, backgroundIndexingPriority)

.onSuccess(\_ =>

partitionedEdges.negate.foreach(e => negateCounters(e.graphId).incr()))

)

.unit

} yield ()

/\*\*

\* This operation is called when a user is put into or taken out of

\* a state in which their retweets should no longer be visible

\* (e.g. suspended or ROPO).

\*/

override def setRetweetVisibility(retweetId: TweetId, setVisible: Boolean): Future[Unit] = {

val retweetEdge = Seq(ExecuteEdge(retweetId, RetweetsGraph, None, Reverse))

if (setVisible) {

tflock

.insertEdges(retweetEdge, backgroundIndexingPriority)

.onSuccess(\_ => insertCounters(RetweetsGraph.id).incr())

} else {

tflock

.archiveEdges(retweetEdge, backgroundIndexingPriority)

.onSuccess(\_ => archiveCounters(RetweetsGraph.id).incr())

}

}

private[this] def createEdges(tweet: Tweet, isUndelete: Boolean): Future[Unit] =

for {

edges <- getEdges(tweet = tweet, isCreate = true, isDelete = false, isUndelete = isUndelete)

() <- tflock.insertEdges(edges, foregroundIndexingPriority)

} yield {

// Count all the edges we've successfully added:

edges.foreach(e => insertCounters(e.graphId).incr())

}

private[this] def addRTEdges(

tweet: Tweet,

share: Share,

isCreate: Boolean,

edges: ListBuffer[ExecuteEdge[StatusGraph]],

futureEdges: ListBuffer[Future[Seq[ExecuteEdge[StatusGraph]]]]

): Unit = {

edges += RetweetsOfMeGraph.edge(share.sourceUserId, tweet.id)

edges += RetweetsByMeGraph.edge(getUserId(tweet), tweet.id)

edges += RetweetsGraph.edge(share.sourceStatusId, tweet.id)

if (isCreate) {

edges += ExecuteEdge(

sourceId = getUserId(tweet),

graph = RetweetSourceGraph,

destinationIds = Some(Seq(share.sourceStatusId)),

direction = Forward,

position = Some(SnowflakeId(tweet.id).time.inMillis)

)

edges.append(TweetsRetweetedGraph.edge(share.sourceUserId, share.sourceStatusId))

} else {

edges += RetweetSourceGraph.edge(getUserId(tweet), share.sourceStatusId)

// if this is the last retweet we need to remove it from the source user's

// tweets retweeted graph

futureEdges.append(

tflock.count(RetweetsGraph.from(share.sourceStatusId)).flatMap { count =>

if (count <= 1) {

tflock.selectAll(RetweetsGraph.from(share.sourceStatusId)).map { tweets =>

if (tweets.size <= 1)

Seq(TweetsRetweetedGraph.edge(share.sourceUserId, share.sourceStatusId))

else

Nil

}

} else {

FutureNil

}

}

)

}

}

private[this] def addReplyEdges(

tweet: Tweet,

edges: ListBuffer[ExecuteEdge[StatusGraph]]

): Unit = {

getReply(tweet).foreach { reply =>

reply.inReplyToStatusId.flatMap { inReplyToStatusId =>

edges += RepliesToTweetsGraph.edge(inReplyToStatusId, tweet.id)

// only index conversationId if this is a reply to another tweet

TweetLenses.conversationId.get(tweet).map { conversationId =>

edges += ConversationGraph.edge(conversationId, tweet.id)

}

}

}

}

private[this] def addDirectedAtEdges(

tweet: Tweet,

edges: ListBuffer[ExecuteEdge[StatusGraph]]

): Unit = {

TweetLenses.directedAtUser.get(tweet).foreach { directedAtUser =>

edges += DirectedAtUserIdGraph.edge(directedAtUser.userId, tweet.id)

}

}

private[this] def addMentionEdges(

tweet: Tweet,

edges: ListBuffer[ExecuteEdge[StatusGraph]]

): Unit = {

getMentions(tweet)

.flatMap(\_.userId).foreach { mention =>

edges += MentionsGraph.edge(mention, tweet.id)

}

}

private[this] def addQTEdges(

tweet: Tweet,

edges: ListBuffer[ExecuteEdge[StatusGraph]],

futureEdges: ListBuffer[Future[Seq[ExecuteEdge[StatusGraph]]]],

isCreate: Boolean

): Unit = {

val userId = getUserId(tweet)

tweet.quotedTweet.foreach { quotedTweet =>

// Regardless of tweet creates/deletes, we add the corresponding edges to the

// following two graphs. Note that we're handling the case for

// the QuotersGraph slightly differently in the tweet delete case.

edges.append(QuotesGraph.edge(quotedTweet.userId, tweet.id))

edges.append(QuoteTweetsIndexGraph.edge(quotedTweet.tweetId, tweet.id))

if (isCreate) {

// As mentioned above, for tweet creates we go ahead and add an edge

// to the QuotersGraph without any additional checks.

edges.append(QuotersGraph.edge(quotedTweet.tweetId, userId))

} else {

// For tweet deletes, we only add an edge to be deleted from the

// QuotersGraph if the tweeting user isn't quoting the tweet anymore

// i.e. if a user has quoted a tweet multiple times, we only delete

// an edge from the QuotersGraph if they've deleted all the quotes,

// otherwise an edge should exist by definition of what the QuotersGraph

// represents.

// Note: There can be a potential edge case here due to a race condition

// in the following scenario.

// i) A quotes a tweet T twice resulting in tweets T1 and T2.

// ii) There should exist edges in the QuotersGraph from T -> A and T1 <-> T, T2 <-> T in

// the QuoteTweetsIndexGraph, but one of the edges haven't been written

// to the QuoteTweetsIndex graph in TFlock yet.

// iii) In this scenario, we shouldn't really be deleting an edge as we're doing below.

// The approach that we're taking below is a "best effort" approach similar to what we

// currently do for RTs.

// Find all the quotes of the quoted tweet from the quoting user

val quotesFromQuotingUser = QuoteTweetsIndexGraph

.from(quotedTweet.tweetId)

.intersect(UserTimelineGraph.from(userId))

futureEdges.append(

tflock

.count(quotesFromQuotingUser).flatMap { count =>

// If this is the last quote of the quoted tweet from the quoting user,

// we go ahead and delete the edge from the QuotersGraph.

if (count <= 1) {

tflock.selectAll(quotesFromQuotingUser).map { tweets =>

if (tweets.size <= 1) {

Seq(QuotersGraph.edge(quotedTweet.tweetId, userId))

} else {

Nil

}

}

} else {

FutureNil

}

}

)

}

}

}

private[this] def addCardEdges(

tweet: Tweet,

edges: ListBuffer[ExecuteEdge[StatusGraph]]

): Unit = {

// Note that we are indexing only the TOO "stored" cards

// (cardUri=card://<cardId>). Rest of the cards are ignored here.

tweet.cardReference

.collect {

case StoredCard(id) =>

edges.append(CardTweetsGraph.edge(id, tweet.id))

}.getOrElse(())

}

// Note: on undelete, this method restores all archived edges, including those that may have

// been archived prior to the delete. This is incorrect behavior but in practice rarely

// causes problems, as undeletes are so rare.

private[this] def addEdgesForDeleteOrUndelete(

tweet: Tweet,

edges: ListBuffer[ExecuteEdge[StatusGraph]]

): Unit = {

edges.appendAll(

Seq(

MentionsGraph.edges(tweet.id, None, Reverse),

RepliesToTweetsGraph.edges(tweet.id, None)

)

)

// When we delete or undelete a conversation control root Tweet we want to archive or restore

// all the edges in InvitedUsersGraph from the Tweet id.

if (hasConversationControl(tweet) && isConversationRoot(tweet)) {

edges.append(InvitedUsersGraph.edges(tweet.id, None))

}

}

private[this] def addSimpleEdges(

tweet: Tweet,

edges: ListBuffer[ExecuteEdge[StatusGraph]]

): Unit = {

if (TweetLenses.nullcast.get(tweet)) {

edges.append(NullcastedTweetsGraph.edge(getUserId(tweet), tweet.id))

} else if (TweetLenses.narrowcast.get(tweet).isDefined) {

edges.append(NarrowcastSentTweetsGraph.edge(getUserId(tweet), tweet.id))

} else {

edges.append(UserTimelineGraph.edge(getUserId(tweet), tweet.id))

if (hasMedia(tweet))

edges.append(MediaTimelineGraph.edge(getUserId(tweet), tweet.id))

// Index root creator subscription tweets.

// Ignore replies because those are not necessarily visible to a user who subscribes to tweet author

val isRootTweet: Boolean = tweet.coreData match {

case Some(c) => c.reply.isEmpty && c.share.isEmpty

case None => true

}

if (tweet.exclusiveTweetControl.isDefined && isRootTweet) {

edges.append(CreatorSubscriptionTimelineGraph.edge(getUserId(tweet), tweet.id))

if (hasMedia(tweet))

edges.append(CreatorSubscriptionMediaTimelineGraph.edge(getUserId(tweet), tweet.id))

}

}

}

/\*\*

\* Issues edges for each mention of user in a conversation-controlled tweet. This way InvitedUsers

\* graph accumulates complete set of ids for @mention-invited users, by conversation id.

\*/

private def invitedUsersEdgesForCreate(

tweet: Tweet,

edges: ListBuffer[ExecuteEdge[StatusGraph]]

): Unit = {

val conversationId: Long = getConversationId(tweet).getOrElse(tweet.id)

val mentions: Seq[UserId] = getMentions(tweet).flatMap(\_.userId)

edges.appendAll(mentions.map(userId => InvitedUsersGraph.edge(conversationId, userId)))

}

/\*\*

\* Issues edges of InviteUsersGraph that ought to be deleted for a conversation controlled reply.

\* These are mentions of users in the given tweet, only if the user was not mentioned elsewhere

\* in the conversation. This way for a conversation, InvitedUsersGraph would always hold a set

\* of all users invited to the conversation, and an edge is removed only after the last mention of

\* a user is deleted.

\*/

private def invitedUsersEdgesForDelete(

tweet: Tweet,

futureEdges: ListBuffer[Future[Seq[ExecuteEdge[StatusGraph]]]]

): Unit = {

getConversationId(tweet).foreach { conversationId: Long =>

val mentions: Seq[UserId] = getMentions(tweet).flatMap(\_.userId)

mentions.foreach { userId =>

val tweetIdsWithinConversation = ConversationGraph.from(conversationId)

val tweetIdsThatMentionUser = MentionsGraph.from(userId)

futureEdges.append(

tflock

.selectAll(

query = tweetIdsThatMentionUser.intersect(tweetIdsWithinConversation),

limit = Some(2), // Just need to know if it is >1 or <=1, so 2 are enough.

pageSize = None // Provide default, otherwise Mockito complains

).map { tweetIds: Seq[Long] =>

if (tweetIds.size <= 1) {

Seq(InvitedUsersGraph.edge(conversationId, userId))

} else {

Nil

}

}

)

}

}

}

private def hasInviteViaMention(tweet: Tweet): Boolean = {

tweet.conversationControl match {

case Some(ConversationControl.ByInvitation(controls)) =>

controls.inviteViaMention.getOrElse(false)

case Some(ConversationControl.Community(controls)) =>

controls.inviteViaMention.getOrElse(false)

case Some(ConversationControl.Followers(followers)) =>

followers.inviteViaMention.getOrElse(false)

case \_ =>

false

}

}

private def hasConversationControl(tweet: Tweet): Boolean =

tweet.conversationControl.isDefined

// If a Tweet has a ConversationControl, it must have a ConversationId associated with it so we

// can compare the ConversationId with the current Tweet ID to determine if it's the root of the

// conversation. See ConversationIdHydrator for more details

private def isConversationRoot(tweet: Tweet): Boolean =

getConversationId(tweet).get == tweet.id

private def addInvitedUsersEdges(

tweet: Tweet,

isCreate: Boolean,

isUndelete: Boolean,

edges: ListBuffer[ExecuteEdge[StatusGraph]],

futureEdges: ListBuffer[Future[Seq[ExecuteEdge[StatusGraph]]]]

): Unit = {

if (hasConversationControl(tweet)) {

if (isCreate) {

if (isConversationRoot(tweet) && !isUndelete) {

// For root Tweets, only add edges for original creates, not for undeletes.

// Undeletes are handled by addEdgesForDeleteOrUndelete.

invitedUsersEdgesForCreate(tweet, edges)

}

if (!isConversationRoot(tweet) && hasInviteViaMention(tweet)) {

// For replies, only add edges when the conversation control is in inviteViaMention mode.

invitedUsersEdgesForCreate(tweet, edges)

}

} else {

if (!isConversationRoot(tweet)) {

invitedUsersEdgesForDelete(tweet, futureEdges)

}

}

}

}

private[this] def getEdges(

tweet: Tweet,

isCreate: Boolean,

isDelete: Boolean,

isUndelete: Boolean

): Future[Seq[ExecuteEdge[StatusGraph]]] = {

val edges = ListBuffer[ExecuteEdge[StatusGraph]]()

val futureEdges = ListBuffer[Future[Seq[ExecuteEdge[StatusGraph]]]]()

addSimpleEdges(tweet, edges)

getShare(tweet) match {

case Some(share) => addRTEdges(tweet, share, isCreate, edges, futureEdges)

case \_ =>

addInvitedUsersEdges(tweet, isCreate, isUndelete, edges, futureEdges)

addReplyEdges(tweet, edges)

addDirectedAtEdges(tweet, edges)

addMentionEdges(tweet, edges)

addQTEdges(tweet, edges, futureEdges, isCreate)

addCardEdges(tweet, edges)

if (isDelete || isUndelete) {

addEdgesForDeleteOrUndelete(tweet, edges)

}

}

Future

.collect(futureEdges)

.map { moreEdges => (edges ++= moreEdges.flatten).toList }

}

}