package com.twitter.recosinjector.edges

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.store.TweetCreationTimeMHStore

import com.twitter.frigate.common.util.SnowflakeUtils

import com.twitter.recos.internal.thriftscala.{RecosUserTweetInfo, TweetType}

import com.twitter.recos.util.Action

import com.twitter.recosinjector.decider.RecosInjectorDecider

import com.twitter.recosinjector.decider.RecosInjectorDeciderConstants

import com.twitter.recosinjector.util.TweetCreateEventDetails

import com.twitter.util.{Future, Time}

class TweetEventToUserTweetEntityGraphBuilder(

userTweetEntityEdgeBuilder: UserTweetEntityEdgeBuilder,

tweetCreationStore: TweetCreationTimeMHStore,

decider: RecosInjectorDecider

)(

override implicit val statsReceiver: StatsReceiver)

extends EventToMessageBuilder[TweetCreateEventDetails, UserTweetEntityEdge] {

// TweetCreationStore counters

private val lastTweetTimeNotInMh = statsReceiver.counter("last\_tweet\_time\_not\_in\_mh")

private val tweetCreationStoreInserts = statsReceiver.counter("tweet\_creation\_store\_inserts")

private val numInvalidActionCounter = statsReceiver.counter("num\_invalid\_tweet\_action")

private val numTweetEdgesCounter = statsReceiver.counter("num\_tweet\_edge")

private val numRetweetEdgesCounter = statsReceiver.counter("num\_retweet\_edge")

private val numReplyEdgesCounter = statsReceiver.counter("num\_reply\_edge")

private val numQuoteEdgesCounter = statsReceiver.counter("num\_quote\_edge")

private val numIsMentionedEdgesCounter = statsReceiver.counter("num\_isMentioned\_edge")

private val numIsMediataggedEdgesCounter = statsReceiver.counter("num\_isMediatagged\_edge")

private val numIsDecider = statsReceiver.counter("num\_decider\_enabled")

private val numIsNotDecider = statsReceiver.counter("num\_decider\_not\_enabled")

override def shouldProcessEvent(event: TweetCreateEventDetails): Future[Boolean] = {

val isDecider = decider.isAvailable(

RecosInjectorDeciderConstants.TweetEventTransformerUserTweetEntityEdgesDecider

)

if (isDecider) {

numIsDecider.incr()

Future(true)

} else {

numIsNotDecider.incr()

Future(false)

}

}

/\*\*

\* Build edges Reply event. Reply event emits 2 edges:

\* author -> Reply -> SourceTweetId

\* author -> Tweet -> ReplyId

\* Do not associate entities in reply tweet to the source tweet

\*/

private def buildReplyEdge(event: TweetCreateEventDetails) = {

val userTweetEngagement = event.userTweetEngagement

val authorId = userTweetEngagement.engageUserId

val replyEdgeFut = event.sourceTweetDetails

.map { sourceTweetDetails =>

val sourceTweetId = sourceTweetDetails.tweet.id

val sourceTweetEntitiesMapFut = userTweetEntityEdgeBuilder.getEntitiesMapAndUpdateCache(

tweetId = sourceTweetId,

tweetDetails = Some(sourceTweetDetails)

)

sourceTweetEntitiesMapFut.map { sourceTweetEntitiesMap =>

val replyEdge = UserTweetEntityEdge(

sourceUser = authorId,

targetTweet = sourceTweetId,

action = Action.Reply,

metadata = Some(userTweetEngagement.tweetId),

cardInfo = Some(sourceTweetDetails.cardInfo.toByte),

entitiesMap = sourceTweetEntitiesMap,

tweetDetails = Some(sourceTweetDetails)

)

numReplyEdgesCounter.incr()

Some(replyEdge)

}

}.getOrElse(Future.None)

val tweetCreationEdgeFut =

if (decider.isAvailable(RecosInjectorDeciderConstants.EnableEmitTweetEdgeFromReply)) {

getAndUpdateLastTweetCreationTime(

authorId = authorId,

tweetId = userTweetEngagement.tweetId,

tweetType = TweetType.Reply

).map { lastTweetTime =>

val edge = UserTweetEntityEdge(

sourceUser = authorId,

targetTweet = userTweetEngagement.tweetId,

action = Action.Tweet,

metadata = lastTweetTime,

cardInfo = userTweetEngagement.tweetDetails.map(\_.cardInfo.toByte),

entitiesMap = None,

tweetDetails = userTweetEngagement.tweetDetails

)

numTweetEdgesCounter.incr()

Some(edge)

}

} else {

Future.None

}

Future.join(replyEdgeFut, tweetCreationEdgeFut).map {

case (replyEdgeOpt, tweetCreationEdgeOpt) =>

tweetCreationEdgeOpt.toSeq ++ replyEdgeOpt.toSeq

}

}

/\*\*

\* Build a Retweet UTEG edge: author -> RT -> SourceTweetId.

\*/

private def buildRetweetEdge(event: TweetCreateEventDetails) = {

val userTweetEngagement = event.userTweetEngagement

val tweetId = userTweetEngagement.tweetId

event.sourceTweetDetails

.map { sourceTweetDetails =>

val sourceTweetId = sourceTweetDetails.tweet.id // Id of the tweet being Retweeted

val sourceTweetEntitiesMapFut = userTweetEntityEdgeBuilder.getEntitiesMapAndUpdateCache(

tweetId = sourceTweetId,

tweetDetails = Some(sourceTweetDetails)

)

sourceTweetEntitiesMapFut.map { sourceTweetEntitiesMap =>

val edge = UserTweetEntityEdge(

sourceUser = userTweetEngagement.engageUserId,

targetTweet = sourceTweetId,

action = Action.Retweet,

metadata = Some(tweetId), // metadata is the tweetId

cardInfo = Some(sourceTweetDetails.cardInfo.toByte),

entitiesMap = sourceTweetEntitiesMap,

tweetDetails = Some(sourceTweetDetails)

)

numRetweetEdgesCounter.incr()

Seq(edge)

}

}.getOrElse(Future.Nil)

}

/\*\*

\* Build edges for a Quote event. Quote tweet emits 2 edges:

\* 1. A quote social proof: author -> Quote -> SourceTweetId

\* 2. A tweet creation edge: author -> Tweet -> QuoteTweetId

\*/

private def buildQuoteEdges(

event: TweetCreateEventDetails

): Future[Seq[UserTweetEntityEdge]] = {

val userTweetEngagement = event.userTweetEngagement

val tweetId = userTweetEngagement.tweetId

val authorId = userTweetEngagement.engageUserId

// do not associate entities in quote tweet to the source tweet,

// but associate entities to quote tweet in tweet creation event

val quoteTweetEdgeFut = event.sourceTweetDetails

.map { sourceTweetDetails =>

val sourceTweetId = sourceTweetDetails.tweet.id // Id of the tweet being quoted

val sourceTweetEntitiesMapFut = userTweetEntityEdgeBuilder.getEntitiesMapAndUpdateCache(

tweetId = sourceTweetId,

tweetDetails = event.sourceTweetDetails

)

sourceTweetEntitiesMapFut.map { sourceTweetEntitiesMap =>

val edge = UserTweetEntityEdge(

sourceUser = authorId,

targetTweet = sourceTweetId,

action = Action.Quote,

metadata = Some(tweetId), // metadata is tweetId

cardInfo = Some(sourceTweetDetails.cardInfo.toByte), // cardInfo of the source tweet

entitiesMap = sourceTweetEntitiesMap,

tweetDetails = Some(sourceTweetDetails)

)

numQuoteEdgesCounter.incr()

Seq(edge)

}

}.getOrElse(Future.Nil)

val tweetCreationEdgeFut = getAndUpdateLastTweetCreationTime(

authorId = authorId,

tweetId = tweetId,

tweetType = TweetType.Quote

).map { lastTweetTime =>

val metadata = lastTweetTime

val cardInfo = userTweetEngagement.tweetDetails.map(\_.cardInfo.toByte)

val edge = UserTweetEntityEdge(

sourceUser = authorId,

targetTweet = tweetId,

action = Action.Tweet,

metadata = metadata,

cardInfo = cardInfo,

entitiesMap = None,

tweetDetails = userTweetEngagement.tweetDetails

)

numTweetEdgesCounter.incr()

Seq(edge)

}

Future.join(quoteTweetEdgeFut, tweetCreationEdgeFut).map {

case (quoteEdge, creationEdge) =>

quoteEdge ++ creationEdge

}

}

/\*\*

\* Build edges for a Tweet event. A Tweet emits 3 tyes edges:

\* 1. A tweet creation edge: author -> Tweet -> TweetId

\* 2. IsMentioned edges: mentionedUserId -> IsMentioned -> TweetId

\* 3. IsMediatagged edges: mediataggedUserId -> IsMediatagged -> TweetId

\*/

private def buildTweetEdges(event: TweetCreateEventDetails): Future[Seq[UserTweetEntityEdge]] = {

val userTweetEngagement = event.userTweetEngagement

val tweetDetails = userTweetEngagement.tweetDetails

val tweetId = userTweetEngagement.tweetId

val authorId = userTweetEngagement.engageUserId

val cardInfo = tweetDetails.map(\_.cardInfo.toByte)

val entitiesMapFut = userTweetEntityEdgeBuilder.getEntitiesMapAndUpdateCache(

tweetId = tweetId,

tweetDetails = tweetDetails

)

val lastTweetTimeFut = getAndUpdateLastTweetCreationTime(

authorId = authorId,

tweetId = tweetId,

tweetType = TweetType.Tweet

)

Future.join(entitiesMapFut, lastTweetTimeFut).map {

case (entitiesMap, lastTweetTime) =>

val tweetCreationEdge = UserTweetEntityEdge(

sourceUser = authorId,

targetTweet = tweetId,

action = Action.Tweet,

metadata = lastTweetTime,

cardInfo = cardInfo,

entitiesMap = entitiesMap,

tweetDetails = userTweetEngagement.tweetDetails

)

numTweetEdgesCounter.incr()

val isMentionedEdges = event.validMentionUserIds

.map(\_.map { mentionedUserId =>

UserTweetEntityEdge(

sourceUser = mentionedUserId,

targetTweet = tweetId,

action = Action.IsMentioned,

metadata = Some(tweetId),

cardInfo = cardInfo,

entitiesMap = entitiesMap,

tweetDetails = userTweetEngagement.tweetDetails

)

}).getOrElse(Nil)

numIsMentionedEdgesCounter.incr(isMentionedEdges.size)

val isMediataggedEdges = event.validMediatagUserIds

.map(\_.map { mediataggedUserId =>

UserTweetEntityEdge(

sourceUser = mediataggedUserId,

targetTweet = tweetId,

action = Action.IsMediaTagged,

metadata = Some(tweetId),

cardInfo = cardInfo,

entitiesMap = entitiesMap,

tweetDetails = userTweetEngagement.tweetDetails

)

}).getOrElse(Nil)

numIsMediataggedEdgesCounter.incr(isMediataggedEdges.size)

Seq(tweetCreationEdge) ++ isMentionedEdges ++ isMediataggedEdges

}

}

/\*\*

\* For a given user, read the user's last time tweeted from the MH store, and

\* write the new tweet time into the MH store before returning.

\* Note this function is async, so the MH write operations will continue to execute on its own.

\* This might create a read/write race condition, but it's expected.

\*/

private def getAndUpdateLastTweetCreationTime(

authorId: Long,

tweetId: Long,

tweetType: TweetType

): Future[Option[Long]] = {

val newTweetInfo = RecosUserTweetInfo(

authorId,

tweetId,

tweetType,

SnowflakeUtils.tweetCreationTime(tweetId).map(\_.inMillis).getOrElse(Time.now.inMillis)

)

tweetCreationStore

.get(authorId)

.map(\_.map { previousTweetInfoSeq =>

val lastTweetTime = previousTweetInfoSeq

.filter(info => info.tweetType == TweetType.Tweet || info.tweetType == TweetType.Quote)

.map(\_.tweetTimestamp)

.sortBy(-\_)

.headOption // Fetch the latest time user Tweeted or Quoted

.getOrElse(

Time.Bottom.inMillis

) // Last tweet time never recorded in MH, default to oldest point in time

if (lastTweetTime == Time.Bottom.inMillis) lastTweetTimeNotInMh.incr()

lastTweetTime

})

.ensure {

tweetCreationStore

.put(authorId, newTweetInfo)

.onSuccess(\_ => tweetCreationStoreInserts.incr())

.onFailure { e =>

statsReceiver.counter("write\_failed\_with\_ex:" + e.getClass.getName).incr()

}

}

}

override def buildEdges(event: TweetCreateEventDetails): Future[Seq[UserTweetEntityEdge]] = {

val userTweetEngagement = event.userTweetEngagement

userTweetEngagement.action match {

case Action.Reply =>

buildReplyEdge(event)

case Action.Retweet =>

buildRetweetEdge(event)

case Action.Tweet =>

buildTweetEdges(event)

case Action.Quote =>

buildQuoteEdges(event)

case \_ =>

numInvalidActionCounter.incr()

Future.Nil

}

}

override def filterEdges(

event: TweetCreateEventDetails,

edges: Seq[UserTweetEntityEdge]

): Future[Seq[UserTweetEntityEdge]] = {

Future(edges) // No filtering for now. Add more if needed

}

}