package com.twitter.tweetypie

package hydrator

import com.twitter.tweetypie.core.\_

import com.twitter.tweetypie.repository.TweetQuery

import com.twitter.tweetypie.tweettext.TweetText

import com.twitter.tweetypie.thriftscala.\_

object CopyFromSourceTweet {

/\*\*

\* A `ValueHydrator` that copies and/or merges certain fields from a retweet's source

\* tweet into the retweet.

\*/

def hydrator: ValueHydrator[TweetData, TweetQuery.Options] =

ValueHydrator.map { (td, \_) =>

td.sourceTweetResult.map(\_.value.tweet) match {

case None => ValueState.unmodified(td)

case Some(src) => ValueState.modified(td.copy(tweet = copy(src, td.tweet)))

}

}

/\*\*

\* Updates `dst` with fields from `src`. This is more complicated than you would think, because:

\*

\* - the tweet has an extra mention entity due to the "RT @user" prefix;

\* - the retweet text may be truncated at the end, and doesn't necessarily contain all of the

\* the text from the source tweet. truncation may happen in the middle of entity.

\* - the text in the retweet may have a different unicode normalization, which affects

\* code point indices. this means entities aren't shifted by a fixed amount equal to

\* the RT prefix.

\* - url entities, when hydrated, may be converted to media entities; url entities may not

\* be hydrated in the retweet, so the source tweet may have a media entity that corresponds

\* to an unhydrated url entity in the retweet.

\* - there may be multiple media entities that map to a single url entity, because the tweet

\* may have multiple photos.

\*/

def copy(src: Tweet, dst: Tweet): Tweet = {

val srcCoreData = src.coreData.get

val dstCoreData = dst.coreData.get

// get the code point index of the end of the text

val max = getText(dst).codePointCount(0, getText(dst).length).toShort

// get all entities from the source tweet, merged into a single list sorted by fromIndex.

val srcEntities = getWrappedEntities(src)

// same for the retweet, but drop first @mention, add back later

val dstEntities = getWrappedEntities(dst).drop(1)

// merge indices from dst into srcEntities. at the end, resort entities back

// to their original ordering. for media entities, order matters to clients.

val mergedEntities = merge(srcEntities, dstEntities, max).sortBy(\_.position)

// extract entities back out by type

val mentions = mergedEntities.collect { case WrappedMentionEntity(e, \_) => e }

val hashtags = mergedEntities.collect { case WrappedHashtagEntity(e, \_) => e }

val cashtags = mergedEntities.collect { case WrappedCashtagEntity(e, \_) => e }

val urls = mergedEntities.collect { case WrappedUrlEntity(e, \_) => e }

val media = mergedEntities.collect { case WrappedMediaEntity(e, \_) => e }

// merge the updated entities back into the retweet, adding the RT @mention back in

dst.copy(

coreData = Some(

dstCoreData.copy(

hasMedia = srcCoreData.hasMedia,

hasTakedown = dstCoreData.hasTakedown || srcCoreData.hasTakedown

)

),

mentions = Some(getMentions(dst).take(1) ++ mentions),

hashtags = Some(hashtags),

cashtags = Some(cashtags),

urls = Some(urls),

media = Some(media.map(updateSourceStatusId(src.id, getUserId(src)))),

quotedTweet = src.quotedTweet,

card2 = src.card2,

cards = src.cards,

language = src.language,

mediaTags = src.mediaTags,

spamLabel = src.spamLabel,

takedownCountryCodes =

mergeTakedowns(Seq(src, dst).map(TweetLenses.takedownCountryCodes.get): \_\*),

conversationControl = src.conversationControl,

exclusiveTweetControl = src.exclusiveTweetControl

)

}

/\*\*

\* Merges one or more optional lists of takedowns. If no lists are defined, returns None.

\*/

private def mergeTakedowns(takedowns: Option[Seq[CountryCode]]\*): Option[Seq[CountryCode]] =

if (takedowns.exists(\_.isDefined)) {

Some(takedowns.flatten.flatten.distinct.sorted)

} else {

None

}

/\*\*

\* A retweet should never have media without a source\_status\_id or source\_user\_id

\*/

private def updateSourceStatusId(

srcTweetId: TweetId,

srcUserId: UserId

): MediaEntity => MediaEntity =

mediaEntity =>

if (mediaEntity.sourceStatusId.nonEmpty) {

// when sourceStatusId is set this indicates the media is "pasted media" so the values

// should already be correct (retweeting won't change sourceStatusId / sourceUserId)

mediaEntity

} else {

mediaEntity.copy(

sourceStatusId = Some(srcTweetId),

sourceUserId = Some(mediaEntity.sourceUserId.getOrElse(srcUserId))

)

}

/\*\*

\* Attempts to match up entities from the source tweet with entities from the retweet,

\* and to use the source tweet entities but shifted to the retweet entity indices. If an entity

\* got truncated at the end of the retweet text, we drop it and any following entities.

\*/

private def merge(

srcEntities: List[WrappedEntity],

rtEntities: List[WrappedEntity],

maxIndex: Short

): List[WrappedEntity] = {

(srcEntities, rtEntities) match {

case (Nil, Nil) =>

// successfully matched all entities!

Nil

case (Nil, \_) =>

// no more source tweet entities, but we still have remaining retweet entities.

// this can happen if a a text truncation turns something invalid like #tag1#tag2 or

// @mention1@mention2 into a valid entity. just drop all the remaining retweet entities.

Nil

case (\_, Nil) =>

// no more retweet entities, which means the remaining entities have been truncated.

Nil

case (srcHead :: srcTail, rtHead :: rtTail) =>

// we have more entities from the source tweet and the retweet. typically, we can

// match these entities because they have the same normalized text, but the retweet

// entity might be truncated, so we allow for a prefix match if the retweet entity

// ends at the end of the tweet.

val possiblyTruncated = rtHead.toIndex == maxIndex - 1

val exactMatch = srcHead.normalizedText == rtHead.normalizedText

if (exactMatch) {

// there could be multiple media entities for the same t.co url, so we need to find

// contiguous groupings of entities that share the same fromIndex.

val rtTail = rtEntities.dropWhile(\_.fromIndex == rtHead.fromIndex)

val srcGroup =

srcEntities

.takeWhile(\_.fromIndex == srcHead.fromIndex)

.map(\_.shift(rtHead.fromIndex, rtHead.toIndex))

val srcTail = srcEntities.drop(srcGroup.size)

srcGroup ++ merge(srcTail, rtTail, maxIndex)

} else {

// if we encounter a mismatch, it is most likely because of truncation,

// so we stop here.

Nil

}

}

}

/\*\*

\* Wraps all the entities with the appropriate WrappedEntity subclasses, merges them into

\* a single list, and sorts by fromIndex.

\*/

private def getWrappedEntities(tweet: Tweet): List[WrappedEntity] =

(getUrls(tweet).zipWithIndex.map { case (e, p) => WrappedUrlEntity(e, p) } ++

getMedia(tweet).zipWithIndex.map { case (e, p) => WrappedMediaEntity(e, p) } ++

getMentions(tweet).zipWithIndex.map { case (e, p) => WrappedMentionEntity(e, p) } ++

getHashtags(tweet).zipWithIndex.map { case (e, p) => WrappedHashtagEntity(e, p) } ++

getCashtags(tweet).zipWithIndex.map { case (e, p) => WrappedCashtagEntity(e, p) })

.sortBy(\_.fromIndex)

.toList

/\*\*

\* The thrift-entity classes don't share a common entity parent class, so we wrap

\* them with a class that allows us to mix entities together into a single list, and

\* to provide a generic interface for shifting indicies.

\*/

private sealed abstract class WrappedEntity(

val fromIndex: Short,

val toIndex: Short,

val rawText: String) {

/\*\* the original position of the entity within the entity group \*/

val position: Int

val normalizedText: String = TweetText.nfcNormalize(rawText).toLowerCase

def shift(fromIndex: Short, toIndex: Short): WrappedEntity

}

private case class WrappedUrlEntity(entity: UrlEntity, position: Int)

extends WrappedEntity(entity.fromIndex, entity.toIndex, entity.url) {

override def shift(fromIndex: Short, toIndex: Short): WrappedUrlEntity =

copy(entity.copy(fromIndex = fromIndex, toIndex = toIndex))

}

private case class WrappedMediaEntity(entity: MediaEntity, position: Int)

extends WrappedEntity(entity.fromIndex, entity.toIndex, entity.url) {

override def shift(fromIndex: Short, toIndex: Short): WrappedMediaEntity =

copy(entity.copy(fromIndex = fromIndex, toIndex = toIndex))

}

private case class WrappedMentionEntity(entity: MentionEntity, position: Int)

extends WrappedEntity(entity.fromIndex, entity.toIndex, entity.screenName) {

override def shift(fromIndex: Short, toIndex: Short): WrappedMentionEntity =

copy(entity.copy(fromIndex = fromIndex, toIndex = toIndex))

}

private case class WrappedHashtagEntity(entity: HashtagEntity, position: Int)

extends WrappedEntity(entity.fromIndex, entity.toIndex, entity.text) {

override def shift(fromIndex: Short, toIndex: Short): WrappedHashtagEntity =

copy(entity.copy(fromIndex = fromIndex, toIndex = toIndex))

}

private case class WrappedCashtagEntity(entity: CashtagEntity, position: Int)

extends WrappedEntity(entity.fromIndex, entity.toIndex, entity.text) {

override def shift(fromIndex: Short, toIndex: Short): WrappedCashtagEntity =

copy(entity.copy(fromIndex = fromIndex, toIndex = toIndex))

}

}