package com.twitter.tweetypie

package service

package observer

import com.twitter.snowflake.id.SnowflakeId

import com.twitter.tweetypie.additionalfields.AdditionalFields

import com.twitter.tweetypie.media.MediaKeyClassifier

import com.twitter.tweetypie.thriftscala.\_

import com.twitter.tweetypie.tweettext.TweetText.codePointLength

import com.twitter.conversions.DurationOps.\_

/\*\*

\* Observer can be used for storing

\* - one-off handler specific metrics with minor logic

\* - reusable Tweetypie service metrics for multiple handlers

\*/

private[service] object Observer {

val successStatusStates: Set[StatusState] = Set(

StatusState.Found,

StatusState.NotFound,

StatusState.DeactivatedUser,

StatusState.SuspendedUser,

StatusState.ProtectedUser,

StatusState.ReportedTweet,

StatusState.UnsupportedClient,

StatusState.Drop,

StatusState.Suppress,

StatusState.Deleted,

StatusState.BounceDeleted

)

def observeStatusStates(statsReceiver: StatsReceiver): Effect[StatusState] = {

val stats = statsReceiver.scope("status\_state")

val total = statsReceiver.counter("status\_results")

val foundCounter = stats.counter("found")

val notFoundCounter = stats.counter("not\_found")

val partialCounter = stats.counter("partial")

val timedOutCounter = stats.counter("timed\_out")

val failedCounter = stats.counter("failed")

val deactivatedCounter = stats.counter("deactivated")

val suspendedCounter = stats.counter("suspended")

val protectedCounter = stats.counter("protected")

val reportedCounter = stats.counter("reported")

val overCapacityCounter = stats.counter("over\_capacity")

val unsupportedClientCounter = stats.counter("unsupported\_client")

val dropCounter = stats.counter("drop")

val suppressCounter = stats.counter("suppress")

val deletedCounter = stats.counter("deleted")

val bounceDeletedCounter = stats.counter("bounce\_deleted")

Effect { st =>

total.incr()

st match {

case StatusState.Found => foundCounter.incr()

case StatusState.NotFound => notFoundCounter.incr()

case StatusState.Partial => partialCounter.incr()

case StatusState.TimedOut => timedOutCounter.incr()

case StatusState.Failed => failedCounter.incr()

case StatusState.DeactivatedUser => deactivatedCounter.incr()

case StatusState.SuspendedUser => suspendedCounter.incr()

case StatusState.ProtectedUser => protectedCounter.incr()

case StatusState.ReportedTweet => reportedCounter.incr()

case StatusState.OverCapacity => overCapacityCounter.incr()

case StatusState.UnsupportedClient => unsupportedClientCounter.incr()

case StatusState.Drop => dropCounter.incr()

case StatusState.Suppress => suppressCounter.incr()

case StatusState.Deleted => deletedCounter.incr()

case StatusState.BounceDeleted => bounceDeletedCounter.incr()

case \_ =>

}

}

}

def observeSetFieldsRequest(stats: StatsReceiver): Effect[SetAdditionalFieldsRequest] =

Effect { request =>

val tweet = request.additionalFields

AdditionalFields.nonEmptyAdditionalFieldIds(tweet).foreach { id =>

val fieldScope = "field\_%d".format(id)

val fieldCounter = stats.counter(fieldScope)

val sizeStats = stats.stat(fieldScope)

tweet.getFieldBlob(id).foreach { blob =>

fieldCounter.incr()

sizeStats.add(blob.content.length)

}

}

}

def observeSetRetweetVisibilityRequest(

stats: StatsReceiver

): Effect[SetRetweetVisibilityRequest] = {

val setInvisibleCounter = stats.counter("set\_invisible")

val setVisibleCounter = stats.counter("set\_visible")

Effect { request =>

if (request.visible) setVisibleCounter.incr() else setInvisibleCounter.incr()

}

}

def observeDeleteFieldsRequest(stats: StatsReceiver): Effect[DeleteAdditionalFieldsRequest] = {

val requestSizeStat = stats.stat("request\_size")

Effect { request =>

requestSizeStat.add(request.tweetIds.size)

request.fieldIds.foreach { id =>

val fieldScope = "field\_%d".format(id)

val fieldCounter = stats.counter(fieldScope)

fieldCounter.incr()

}

}

}

def observeDeleteTweetsRequest(stats: StatsReceiver): Effect[DeleteTweetsRequest] = {

val requestSizeStat = stats.stat("request\_size")

val userErasureTweetsStat = stats.counter("user\_erasure\_tweets")

val isBounceDeleteStat = stats.counter("is\_bounce\_delete\_tweets")

Effect {

case DeleteTweetsRequest(tweetIds, \_, \_, \_, isUserErasure, \_, isBounceDelete, \_, \_) =>

requestSizeStat.add(tweetIds.size)

if (isUserErasure) {

userErasureTweetsStat.incr(tweetIds.size)

}

if (isBounceDelete) {

isBounceDeleteStat.incr(tweetIds.size)

}

}

}

def observeRetweetRequest(stats: StatsReceiver): Effect[RetweetRequest] = {

val optionsScope = stats.scope("options")

val narrowcastCounter = optionsScope.counter("narrowcast")

val nullcastCounter = optionsScope.counter("nullcast")

val darkCounter = optionsScope.counter("dark")

val successOnDupCounter = optionsScope.counter("success\_on\_dup")

Effect { request =>

if (request.narrowcast.nonEmpty) narrowcastCounter.incr()

if (request.nullcast) nullcastCounter.incr()

if (request.dark) darkCounter.incr()

if (request.returnSuccessOnDuplicate) successOnDupCounter.incr()

}

}

def observeScrubGeo(stats: StatsReceiver): Effect[GeoScrub] = {

val optionsScope = stats.scope("options")

val hosebirdEnqueueCounter = optionsScope.counter("hosebird\_enqueue")

val requestSizeStat = stats.stat("request\_size")

Effect { request =>

requestSizeStat.add(request.statusIds.size)

if (request.hosebirdEnqueue) hosebirdEnqueueCounter.incr()

}

}

def observeEventOrRetry(stats: StatsReceiver, isRetry: Boolean): Unit = {

val statName = if (isRetry) "retry" else "event"

stats.counter(statName).incr()

}

def observeAsyncInsertRequest(stats: StatsReceiver): Effect[AsyncInsertRequest] = {

val insertScope = stats.scope("insert")

val ageStat = insertScope.stat("age")

Effect { request =>

observeEventOrRetry(insertScope, request.retryAction.isDefined)

ageStat.add(SnowflakeId.timeFromId(request.tweet.id).untilNow.inMillis)

}

}

def observeAsyncSetAdditionalFieldsRequest(

stats: StatsReceiver

): Effect[AsyncSetAdditionalFieldsRequest] = {

val setAdditionalFieldsScope = stats.scope("set\_additional\_fields")

Effect { request =>

observeEventOrRetry(setAdditionalFieldsScope, request.retryAction.isDefined)

}

}

def observeAsyncSetRetweetVisibilityRequest(

stats: StatsReceiver

): Effect[AsyncSetRetweetVisibilityRequest] = {

val setRetweetVisibilityScope = stats.scope("set\_retweet\_visibility")

Effect { request =>

observeEventOrRetry(setRetweetVisibilityScope, request.retryAction.isDefined)

}

}

def observeAsyncUndeleteTweetRequest(stats: StatsReceiver): Effect[AsyncUndeleteTweetRequest] = {

val undeleteTweetScope = stats.scope("undelete\_tweet")

Effect { request => observeEventOrRetry(undeleteTweetScope, request.retryAction.isDefined) }

}

def observeAsyncDeleteTweetRequest(stats: StatsReceiver): Effect[AsyncDeleteRequest] = {

val deleteTweetScope = stats.scope("delete\_tweet")

Effect { request => observeEventOrRetry(deleteTweetScope, request.retryAction.isDefined) }

}

def observeAsyncDeleteAdditionalFieldsRequest(

stats: StatsReceiver

): Effect[AsyncDeleteAdditionalFieldsRequest] = {

val deleteAdditionalFieldsScope = stats.scope("delete\_additional\_fields")

Effect { request =>

observeEventOrRetry(

deleteAdditionalFieldsScope,

request.retryAction.isDefined

)

}

}

def observeAsyncTakedownRequest(stats: StatsReceiver): Effect[AsyncTakedownRequest] = {

val takedownScope = stats.scope("takedown")

Effect { request => observeEventOrRetry(takedownScope, request.retryAction.isDefined) }

}

def observeAsyncUpdatePossiblySensitiveTweetRequest(

stats: StatsReceiver

): Effect[AsyncUpdatePossiblySensitiveTweetRequest] = {

val updatePossiblySensitiveTweetScope = stats.scope("update\_possibly\_sensitive\_tweet")

Effect { request =>

observeEventOrRetry(updatePossiblySensitiveTweetScope, request.action.isDefined)

}

}

def observeReplicatedInsertTweetRequest(stats: StatsReceiver): Effect[Tweet] = {

val ageStat = stats.stat("age") // in milliseconds

Effect { request => ageStat.add(SnowflakeId.timeFromId(request.id).untilNow.inMillis) }

}

def camelToUnderscore(str: String): String = {

val bldr = new StringBuilder

str.foldLeft(false) { (prevWasLowercase, c) =>

if (prevWasLowercase && c.isUpper) {

bldr += '\_'

}

bldr += c.toLower

c.isLower

}

bldr.result

}

def observeAdditionalFields(stats: StatsReceiver): Effect[Tweet] = {

val additionalScope = stats.scope("additional\_fields")

Effect { tweet =>

for (fieldId <- AdditionalFields.nonEmptyAdditionalFieldIds(tweet))

additionalScope.counter(fieldId.toString).incr()

}

}

/\*\*

\* We count how many tweets have each of these attributes so that we

\* can observe general trends, as well as for tracking down the

\* cause of behavior changes, like increased calls to certain

\* services.

\*/

def countTweetAttributes(stats: StatsReceiver, byClient: Boolean): Effect[Tweet] = {

val ageStat = stats.stat("age")

val tweetCounter = stats.counter("tweets")

val retweetCounter = stats.counter("retweets")

val repliesCounter = stats.counter("replies")

val inReplyToTweetCounter = stats.counter("in\_reply\_to\_tweet")

val selfRepliesCounter = stats.counter("self\_replies")

val directedAtCounter = stats.counter("directed\_at")

val mentionsCounter = stats.counter("mentions")

val mentionsStat = stats.stat("mentions")

val urlsCounter = stats.counter("urls")

val urlsStat = stats.stat("urls")

val hashtagsCounter = stats.counter("hashtags")

val hashtagsStat = stats.stat("hashtags")

val mediaCounter = stats.counter("media")

val mediaStat = stats.stat("media")

val photosCounter = stats.counter("media", "photos")

val gifsCounter = stats.counter("media", "animated\_gifs")

val videosCounter = stats.counter("media", "videos")

val cardsCounter = stats.counter("cards")

val card2Counter = stats.counter("card2")

val geoCoordsCounter = stats.counter("geo\_coordinates")

val placeCounter = stats.counter("place")

val quotedTweetCounter = stats.counter("quoted\_tweet")

val selfRetweetCounter = stats.counter("self\_retweet")

val languageScope = stats.scope("language")

val textLengthStat = stats.stat("text\_length")

val selfThreadCounter = stats.counter("self\_thread")

val communitiesTweetCounter = stats.counter("communities")

observeAdditionalFields(stats).also {

Effect[Tweet] { tweet =>

def coreDataField[T](f: TweetCoreData => T): Option[T] =

tweet.coreData.map(f)

def coreDataOptionField[T](f: TweetCoreData => Option[T]) =

coreDataField(f).flatten

(SnowflakeId.isSnowflakeId(tweet.id) match {

case true => Some(SnowflakeId.timeFromId(tweet.id))

case false => coreDataField(\_.createdAtSecs.seconds.afterEpoch)

}).foreach { createdAt => ageStat.add(createdAt.untilNow.inSeconds) }

if (!byClient) {

val mentions = getMentions(tweet)

val urls = getUrls(tweet)

val hashtags = getHashtags(tweet)

val media = getMedia(tweet)

val mediaKeys = media.flatMap(\_.mediaKey)

val share = coreDataOptionField(\_.share)

val selfThreadMetadata = getSelfThreadMetadata(tweet)

val communities = getCommunities(tweet)

tweetCounter.incr()

if (share.isDefined) retweetCounter.incr()

if (coreDataOptionField(\_.directedAtUser).isDefined) directedAtCounter.incr()

coreDataOptionField(\_.reply).foreach { reply =>

repliesCounter.incr()

if (reply.inReplyToStatusId.nonEmpty) {

// repliesCounter counts all Tweets with a Reply struct,

// but that includes both directed-at Tweets and

// conversational replies. Only conversational replies

// have inReplyToStatusId present, so this counter lets

// us split apart those two cases.

inReplyToTweetCounter.incr()

}

// Not all Tweet objects have CoreData yet isSelfReply() requires it. Thus, this

// invocation is guarded by the `coreDataOptionField(\_.reply)` above.

if (isSelfReply(tweet)) selfRepliesCounter.incr()

}

if (mentions.nonEmpty) mentionsCounter.incr()

if (urls.nonEmpty) urlsCounter.incr()

if (hashtags.nonEmpty) hashtagsCounter.incr()

if (media.nonEmpty) mediaCounter.incr()

if (selfThreadMetadata.nonEmpty) selfThreadCounter.incr()

if (communities.nonEmpty) communitiesTweetCounter.incr()

mentionsStat.add(mentions.size)

urlsStat.add(urls.size)

hashtagsStat.add(hashtags.size)

mediaStat.add(media.size)

if (mediaKeys.exists(MediaKeyClassifier.isImage(\_))) photosCounter.incr()

if (mediaKeys.exists(MediaKeyClassifier.isGif(\_))) gifsCounter.incr()

if (mediaKeys.exists(MediaKeyClassifier.isVideo(\_))) videosCounter.incr()

if (tweet.cards.exists(\_.nonEmpty)) cardsCounter.incr()

if (tweet.card2.nonEmpty) card2Counter.incr()

if (coreDataOptionField(\_.coordinates).nonEmpty) geoCoordsCounter.incr()

if (TweetLenses.place.get(tweet).nonEmpty) placeCounter.incr()

if (TweetLenses.quotedTweet.get(tweet).nonEmpty) quotedTweetCounter.incr()

if (share.exists(\_.sourceUserId == getUserId(tweet))) selfRetweetCounter.incr()

tweet.language

.map(\_.language)

.foreach(lang => languageScope.counter(lang).incr())

coreDataField(\_.text).foreach(text => textLengthStat.add(codePointLength(text)))

}

}

}

}

}