package com.twitter.timelineranker.util

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.logging.Level

import com.twitter.logging.Logger

import com.twitter.timelines.model.TweetId

import com.twitter.timelines.model.UserId

import com.twitter.timelines.model.tweet.HydratedTweet

import com.twitter.timelines.util.stats.BooleanObserver

import com.twitter.timelines.util.stats.RequestStats

import scala.collection.mutable

object TweetFilters extends Enumeration {

// Filters independent of users or their follow graph.

val DuplicateRetweets: Value = Value

val DuplicateTweets: Value = Value

val NullcastTweets: Value = Value

val Replies: Value = Value

val Retweets: Value = Value

// Filters that depend on users or their follow graph.

val DirectedAtNotFollowedUsers: Value = Value

val NonReplyDirectedAtNotFollowedUsers: Value = Value

val TweetsFromNotFollowedUsers: Value = Value

val ExtendedReplies: Value = Value

val NotQualifiedExtendedReplies: Value = Value

val NotValidExpandedExtendedReplies: Value = Value

val NotQualifiedReverseExtendedReplies: Value = Value

val RecommendedRepliesToNotFollowedUsers: Value = Value

val None: TweetFilters.ValueSet = ValueSet.empty

val UserDependent: ValueSet = ValueSet(

NonReplyDirectedAtNotFollowedUsers,

DirectedAtNotFollowedUsers,

TweetsFromNotFollowedUsers,

ExtendedReplies,

NotQualifiedExtendedReplies,

NotValidExpandedExtendedReplies,

NotQualifiedReverseExtendedReplies,

RecommendedRepliesToNotFollowedUsers

)

val UserIndependent: ValueSet = ValueSet(

DuplicateRetweets,

DuplicateTweets,

NullcastTweets,

Replies,

Retweets

)

require(

(UserDependent ++ UserIndependent) == TweetFilters.values,

"UserIndependent and UserDependent should contain all possible filters"

)

private[util] type FilterMethod =

(HydratedTweet, TweetsPostFilterParams, MutableState) => Boolean

case class MutableState(

seenTweetIds: mutable.Map[TweetId, Int] = mutable.Map.empty[TweetId, Int].withDefaultValue(0)) {

def isSeen(tweetId: TweetId): Boolean = {

val seen = seenTweetIds(tweetId) >= 1

incrementIf0(tweetId)

seen

}

def incrementIf0(key: TweetId): Unit = {

if (seenTweetIds(key) == 0) {

seenTweetIds(key) = 1

}

}

def incrementThenGetCount(key: TweetId): Int = {

seenTweetIds(key) += 1

seenTweetIds(key)

}

}

}

case class TweetsPostFilterParams(

userId: UserId,

followedUserIds: Seq[UserId],

inNetworkUserIds: Seq[UserId],

mutedUserIds: Set[UserId],

numRetweetsAllowed: Int,

loggingPrefix: String = "",

sourceTweets: Seq[HydratedTweet] = Nil) {

lazy val sourceTweetsById: Map[TweetId, HydratedTweet] =

sourceTweets.map(tweet => tweet.tweetId -> tweet).toMap

}

/\*\*

\* Performs post-filtering on tweets obtained from search.

\*

\* Search currently does not perform certain steps or performs them inadequately.

\* This class addresses those shortcomings by post-processing hydrated search results.

\*/

abstract class TweetsPostFilterBase(

filters: TweetFilters.ValueSet,

logger: Logger,

statsReceiver: StatsReceiver)

extends RequestStats {

import TweetFilters.FilterMethod

import TweetFilters.MutableState

private[this] val baseScope = statsReceiver.scope("filter")

private[this] val directedAtNotFollowedCounter = baseScope.counter("directedAtNotFollowed")

private[this] val nonReplyDirectedAtNotFollowedObserver =

BooleanObserver(baseScope.scope("nonReplyDirectedAtNotFollowed"))

private[this] val dupRetweetCounter = baseScope.counter("dupRetweet")

private[this] val dupTweetCounter = baseScope.counter("dupTweet")

private[this] val notFollowedCounter = baseScope.counter("notFollowed")

private[this] val nullcastCounter = baseScope.counter("nullcast")

private[this] val repliesCounter = baseScope.counter("replies")

private[this] val retweetsCounter = baseScope.counter("retweets")

private[this] val extendedRepliesCounter = baseScope.counter("extendedReplies")

private[this] val notQualifiedExtendedRepliesObserver =

BooleanObserver(baseScope.scope("notQualifiedExtendedReplies"))

private[this] val notValidExpandedExtendedRepliesObserver =

BooleanObserver(baseScope.scope("notValidExpandedExtendedReplies"))

private[this] val notQualifiedReverseExtendedRepliesCounter =

baseScope.counter("notQualifiedReverseExtendedReplies")

private[this] val recommendedRepliesToNotFollowedUsersObserver =

BooleanObserver(baseScope.scope("recommendedRepliesToNotFollowedUsers"))

private[this] val totalCounter = baseScope.counter(Total)

private[this] val resultCounter = baseScope.counter("result")

// Used for debugging. Its values should remain false for prod use.

private[this] val alwaysLog = false

val applicableFilters: Seq[FilterMethod] = Filters.getApplicableFilters(filters)

protected def filter(

tweets: Seq[HydratedTweet],

params: TweetsPostFilterParams

): Seq[HydratedTweet] = {

val invocationState = MutableState()

val result = tweets.reverseIterator

.filterNot { tweet => applicableFilters.exists(\_(tweet, params, invocationState)) }

.toSeq

.reverse

totalCounter.incr(tweets.size)

resultCounter.incr(result.size)

result

}

object Filters {

case class FilterData(kind: TweetFilters.Value, method: FilterMethod)

private val allFilters = Seq[FilterData](

FilterData(TweetFilters.DuplicateTweets, isDuplicateTweet),

FilterData(TweetFilters.DuplicateRetweets, isDuplicateRetweet),

FilterData(TweetFilters.DirectedAtNotFollowedUsers, isDirectedAtNonFollowedUser),

FilterData(

TweetFilters.NonReplyDirectedAtNotFollowedUsers,

isNonReplyDirectedAtNonFollowedUser

),

FilterData(TweetFilters.NullcastTweets, isNullcast),

FilterData(TweetFilters.Replies, isReply),

FilterData(TweetFilters.Retweets, isRetweet),

FilterData(TweetFilters.TweetsFromNotFollowedUsers, isFromNonFollowedUser),

FilterData(TweetFilters.ExtendedReplies, isExtendedReply),

FilterData(TweetFilters.NotQualifiedExtendedReplies, isNotQualifiedExtendedReply),

FilterData(TweetFilters.NotValidExpandedExtendedReplies, isNotValidExpandedExtendedReply),

FilterData(

TweetFilters.NotQualifiedReverseExtendedReplies,

isNotQualifiedReverseExtendedReply),

FilterData(

TweetFilters.RecommendedRepliesToNotFollowedUsers,

isRecommendedRepliesToNotFollowedUsers)

)

def getApplicableFilters(filters: TweetFilters.ValueSet): Seq[FilterMethod] = {

require(allFilters.map(\_.kind).toSet == TweetFilters.values)

allFilters.filter(data => filters.contains(data.kind)).map(\_.method)

}

private def isNullcast(

tweet: HydratedTweet,

params: TweetsPostFilterParams,

invocationState: MutableState

): Boolean = {

if (tweet.isNullcast) {

nullcastCounter.incr()

log(

Level.ERROR,

() => s"${params.loggingPrefix}:: Found nullcast tweet: tweet-id: ${tweet.tweetId}"

)

true

} else {

false

}

}

private def isReply(

tweet: HydratedTweet,

params: TweetsPostFilterParams,

invocationState: MutableState

): Boolean = {

if (tweet.hasReply) {

repliesCounter.incr()

log(Level.OFF, () => s"${params.loggingPrefix}:: Removed reply: tweet-id: ${tweet.tweetId}")

true

} else {

false

}

}

private def isRetweet(

tweet: HydratedTweet,

params: TweetsPostFilterParams,

invocationState: MutableState

): Boolean = {

if (tweet.isRetweet) {

retweetsCounter.incr()

log(

Level.OFF,

() => s"${params.loggingPrefix}:: Removed retweet: tweet-id: ${tweet.tweetId}"

)

true

} else {

false

}

}

private def isFromNonFollowedUser(

tweet: HydratedTweet,

params: TweetsPostFilterParams,

invocationState: MutableState

): Boolean = {

if ((tweet.userId != params.userId) && !params.inNetworkUserIds.contains(tweet.userId)) {

notFollowedCounter.incr()

log(

Level.ERROR,

() =>

s"${params.loggingPrefix}:: Found tweet from not-followed user: ${tweet.tweetId} from ${tweet.userId}"

)

true

} else {

false

}

}

private def isDirectedAtNonFollowedUser(

tweet: HydratedTweet,

params: TweetsPostFilterParams,

invocationState: MutableState

): Boolean = {

tweet.directedAtUser.exists { directedAtUserId =>

val shouldFilterOut = (tweet.userId != params.userId) && !params.inNetworkUserIds

.contains(directedAtUserId)

// We do not log here because search is known to not handle this case.

if (shouldFilterOut) {

log(

Level.OFF,

() =>

s"${params.loggingPrefix}:: Found tweet: ${tweet.tweetId} directed-at not-followed user: $directedAtUserId"

)

directedAtNotFollowedCounter.incr()

}

shouldFilterOut

}

}

private def isNonReplyDirectedAtNonFollowedUser(

tweet: HydratedTweet,

params: TweetsPostFilterParams,

invocationState: MutableState

): Boolean = {

tweet.directedAtUser.exists { directedAtUserId =>

val shouldFilterOut = !tweet.hasReply &&

(tweet.userId != params.userId) &&

!params.inNetworkUserIds.contains(directedAtUserId)

// We do not log here because search is known to not handle this case.

if (nonReplyDirectedAtNotFollowedObserver(shouldFilterOut)) {

log(

Level.OFF,

() =>

s"${params.loggingPrefix}:: Found non-reply tweet: ${tweet.tweetId} directed-at not-followed user: $directedAtUserId"

)

}

shouldFilterOut

}

}

/\*\*

\* Determines whether the given tweet has already been seen.

\*/

private def isDuplicateTweet(

tweet: HydratedTweet,

params: TweetsPostFilterParams,

invocationState: MutableState

): Boolean = {

val shouldFilterOut = invocationState.isSeen(tweet.tweetId)

if (shouldFilterOut) {

dupTweetCounter.incr()

log(Level.ERROR, () => s"${params.loggingPrefix}:: Duplicate tweet found: ${tweet.tweetId}")

}

shouldFilterOut

}

/\*\*

\* If the given tweet is a retweet, determines whether the source tweet

\* of that retweet has already been seen.

\*/

private def isDuplicateRetweet(

tweet: HydratedTweet,

params: TweetsPostFilterParams,

invocationState: MutableState

): Boolean = {

invocationState.incrementIf0(tweet.tweetId)

tweet.sourceTweetId.exists { sourceTweetId =>

val seenCount = invocationState.incrementThenGetCount(sourceTweetId)

val shouldFilterOut = seenCount > params.numRetweetsAllowed

if (shouldFilterOut) {

// We do not log here because search is known to not handle this case.

dupRetweetCounter.incr()

log(

Level.OFF,

() =>

s"${params.loggingPrefix}:: Found dup retweet: ${tweet.tweetId} (source tweet: $sourceTweetId), count: $seenCount"

)

}

shouldFilterOut

}

}

private def isExtendedReply(

tweet: HydratedTweet,

params: TweetsPostFilterParams,

invocationState: MutableState

): Boolean = {

val shouldFilterOut = ExtendedRepliesFilter.isExtendedReply(

tweet,

params.followedUserIds

)

if (shouldFilterOut) {

extendedRepliesCounter.incr()

log(

Level.DEBUG,

() => s"${params.loggingPrefix}:: extended reply to be filtered: ${tweet.tweetId}"

)

}

shouldFilterOut

}

private def isNotQualifiedExtendedReply(

tweet: HydratedTweet,

params: TweetsPostFilterParams,

invocationState: MutableState

): Boolean = {

val shouldFilterOut = ExtendedRepliesFilter.isNotQualifiedExtendedReply(

tweet,

params.userId,

params.followedUserIds,

params.mutedUserIds,

params.sourceTweetsById

)

if (notQualifiedExtendedRepliesObserver(shouldFilterOut)) {

log(

Level.DEBUG,

() =>

s"${params.loggingPrefix}:: non qualified extended reply to be filtered: ${tweet.tweetId}"

)

}

shouldFilterOut

}

private def isNotValidExpandedExtendedReply(

tweet: HydratedTweet,

params: TweetsPostFilterParams,

invocationState: MutableState

): Boolean = {

val shouldFilterOut = ExtendedRepliesFilter.isNotValidExpandedExtendedReply(

tweet,

params.userId,

params.followedUserIds,

params.mutedUserIds,

params.sourceTweetsById

)

if (notValidExpandedExtendedRepliesObserver(shouldFilterOut)) {

log(

Level.DEBUG,

() =>

s"${params.loggingPrefix}:: non qualified extended reply to be filtered: ${tweet.tweetId}"

)

}

shouldFilterOut

}

private def isRecommendedRepliesToNotFollowedUsers(

tweet: HydratedTweet,

params: TweetsPostFilterParams,

invocationState: MutableState

): Boolean = {

val shouldFilterOut = RecommendedRepliesFilter.isRecommendedReplyToNotFollowedUser(

tweet,

params.userId,

params.followedUserIds,

params.mutedUserIds

)

if (recommendedRepliesToNotFollowedUsersObserver(shouldFilterOut)) {

log(

Level.DEBUG,

() =>

s"${params.loggingPrefix}:: non qualified recommended reply to be filtered: ${tweet.tweetId}"

)

}

shouldFilterOut

}

//For now this filter is meant to be used only with reply tweets from the inReplyToUserId query

private def isNotQualifiedReverseExtendedReply(

tweet: HydratedTweet,

params: TweetsPostFilterParams,

invocationState: MutableState

): Boolean = {

val shouldFilterOut = !ReverseExtendedRepliesFilter.isQualifiedReverseExtendedReply(

tweet,

params.userId,

params.followedUserIds,

params.mutedUserIds,

params.sourceTweetsById

)

if (shouldFilterOut) {

notQualifiedReverseExtendedRepliesCounter.incr()

log(

Level.DEBUG,

() =>

s"${params.loggingPrefix}:: non qualified reverse extended reply to be filtered: ${tweet.tweetId}"

)

}

shouldFilterOut

}

private def log(level: Level, message: () => String): Unit = {

if (alwaysLog || ((level != Level.OFF) && logger.isLoggable(level))) {

val updatedLevel = if (alwaysLog) Level.INFO else level

logger.log(updatedLevel, message())

}

}

}

}

class TweetsPostFilter(filters: TweetFilters.ValueSet, logger: Logger, statsReceiver: StatsReceiver)

extends TweetsPostFilterBase(filters, logger, statsReceiver) {

def apply(

userId: UserId,

followedUserIds: Seq[UserId],

inNetworkUserIds: Seq[UserId],

mutedUserIds: Set[UserId],

tweets: Seq[HydratedTweet],

numRetweetsAllowed: Int = 1,

sourceTweets: Seq[HydratedTweet] = Nil

): Seq[HydratedTweet] = {

val loggingPrefix = s"userId: $userId"

val params = TweetsPostFilterParams(

userId = userId,

followedUserIds = followedUserIds,

inNetworkUserIds = inNetworkUserIds,

mutedUserIds = mutedUserIds,

numRetweetsAllowed = numRetweetsAllowed,

loggingPrefix = loggingPrefix,

sourceTweets = sourceTweets

)

super.filter(tweets, params)

}

}

class TweetsPostFilterUserIndependent(

filters: TweetFilters.ValueSet,

logger: Logger,

statsReceiver: StatsReceiver)

extends TweetsPostFilterBase(filters, logger, statsReceiver) {

require(

(filters -- TweetFilters.UserIndependent).isEmpty,

"Only user independent filters are supported"

)

def apply(tweets: Seq[HydratedTweet], numRetweetsAllowed: Int = 1): Seq[HydratedTweet] = {

val params = TweetsPostFilterParams(

userId = 0L,

followedUserIds = Seq.empty,

inNetworkUserIds = Seq.empty,

mutedUserIds = Set.empty,

numRetweetsAllowed

)

super.filter(tweets, params)

}

}