package com.twitter.timelineranker.util

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.logging.Level

import com.twitter.logging.Logger

import com.twitter.search.earlybird.thriftscala.ThriftSearchResult

import com.twitter.timelines.model.TweetId

import com.twitter.timelines.model.UserId

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

import scala.collection.mutable

object TweetFiltersBasedOnSearchMetadata extends Enumeration {

val DuplicateRetweets: Value = Value

val DuplicateTweets: Value = Value

val None: TweetFiltersBasedOnSearchMetadata.ValueSet = ValueSet.empty

private[util] type FilterBasedOnSearchMetadataMethod =

(ThriftSearchResult, TweetsPostFilterBasedOnSearchMetadataParams, 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 TweetsPostFilterBasedOnSearchMetadataParams(

userId: UserId,

inNetworkUserIds: Seq[UserId],

numRetweetsAllowed: Int,

loggingPrefix: String = "")

/\*\*

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

\*

\* Search currently does not perform certain steps while searching, so this class addresses those

\* shortcomings by post-processing search results using the returned metadata.

\*/

class TweetsPostFilterBasedOnSearchMetadata(

filters: TweetFiltersBasedOnSearchMetadata.ValueSet,

logger: Logger,

statsReceiver: StatsReceiver)

extends RequestStats {

import TweetFiltersBasedOnSearchMetadata.FilterBasedOnSearchMetadataMethod

import TweetFiltersBasedOnSearchMetadata.MutableState

private[this] val baseScope = statsReceiver.scope("filter\_based\_on\_search\_metadata")

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

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

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[FilterBasedOnSearchMetadataMethod] =

FiltersBasedOnSearchMetadata.getApplicableFilters(filters)

def apply(

userId: UserId,

inNetworkUserIds: Seq[UserId],

tweets: Seq[ThriftSearchResult],

numRetweetsAllowed: Int = 1

): Seq[ThriftSearchResult] = {

val loggingPrefix = s"userId: $userId"

val params = TweetsPostFilterBasedOnSearchMetadataParams(

userId = userId,

inNetworkUserIds = inNetworkUserIds,

numRetweetsAllowed = numRetweetsAllowed,

loggingPrefix = loggingPrefix,

)

filter(tweets, params)

}

protected def filter(

tweets: Seq[ThriftSearchResult],

params: TweetsPostFilterBasedOnSearchMetadataParams

): Seq[ThriftSearchResult] = {

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 FiltersBasedOnSearchMetadata {

case class FilterData(

kind: TweetFiltersBasedOnSearchMetadata.Value,

method: FilterBasedOnSearchMetadataMethod)

private val allFilters = Seq[FilterData](

FilterData(TweetFiltersBasedOnSearchMetadata.DuplicateTweets, isDuplicateTweet),

FilterData(TweetFiltersBasedOnSearchMetadata.DuplicateRetweets, isDuplicateRetweet)

)

def getApplicableFilters(

filters: TweetFiltersBasedOnSearchMetadata.ValueSet

): Seq[FilterBasedOnSearchMetadataMethod] = {

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

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

}

/\*\*

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

\*/

private def isDuplicateTweet(

tweet: ThriftSearchResult,

params: TweetsPostFilterBasedOnSearchMetadataParams,

invocationState: MutableState

): Boolean = {

val shouldFilterOut = invocationState.isSeen(tweet.id)

if (shouldFilterOut) {

dupTweetCounter.incr()

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

}

shouldFilterOut

}

/\*\*

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

\* of that retweet has already been seen.

\*/

private def isDuplicateRetweet(

tweet: ThriftSearchResult,

params: TweetsPostFilterBasedOnSearchMetadataParams,

invocationState: MutableState

): Boolean = {

invocationState.incrementIf0(tweet.id)

SearchResultUtil.getRetweetSourceTweetId(tweet).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.id} (source tweet: $sourceTweetId), count: $seenCount"

)

}

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())

}

}

}

}