package com.twitter.follow\_recommendations.blenders

import com.google.common.annotations.VisibleForTesting

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.follow\_recommendations.common.base.Transform

import com.twitter.follow\_recommendations.common.models.AdMetadata

import com.twitter.follow\_recommendations.common.models.Recommendation

import com.twitter.inject.Logging

import com.twitter.stitch.Stitch

import javax.inject.Inject

import javax.inject.Singleton

@Singleton

class PromotedAccountsBlender @Inject() (statsReceiver: StatsReceiver)

extends Transform[Int, Recommendation]

with Logging {

import PromotedAccountsBlender.\_

val stats = statsReceiver.scope(Name)

val inputOrganicAccounts = stats.counter(InputOrganic)

val inputPromotedAccounts = stats.counter(InputPromoted)

val outputOrganicAccounts = stats.counter(OutputOrganic)

val outputPromotedAccounts = stats.counter(OutputPromoted)

val promotedAccountsStats = stats.scope(NumPromotedAccounts)

override def transform(

maxResults: Int,

items: Seq[Recommendation]

): Stitch[Seq[Recommendation]] = {

val (promoted, organic) = items.partition(\_.isPromotedAccount)

val promotedIds = promoted.map(\_.id).toSet

val dedupedOrganic = organic.filterNot(u => promotedIds.contains(u.id))

val blended = blendPromotedAccount(dedupedOrganic, promoted, maxResults)

val (outputPromoted, outputOrganic) = blended.partition(\_.isPromotedAccount)

inputOrganicAccounts.incr(dedupedOrganic.size)

inputPromotedAccounts.incr(promoted.size)

outputOrganicAccounts.incr(outputOrganic.size)

val size = outputPromoted.size

outputPromotedAccounts.incr(size)

if (size <= 5) {

promotedAccountsStats.counter(outputPromoted.size.toString).incr()

} else {

promotedAccountsStats.counter(MoreThan5Promoted).incr()

}

Stitch.value(blended)

}

/\*\*

\* Merge Promoted results and organic results. Promoted result dictates the position

\* in the merge list.

\*

\* merge a list of positioned users, aka. promoted, and a list of organic

\* users. The positioned promoted users are pre-sorted with regards to their

\* position ascendingly. Only requirement about position is to be within the

\* range, i.e, can not exceed the combined length if merge is successful, ok

\* to be at the last position, but not beyond.

\* For more detailed description of location position:

\* http://confluence.local.twitter.com/display/ADS/Promoted+Tweets+in+Timeline+Design+Document

\*/

@VisibleForTesting

private[blenders] def mergePromotedAccounts(

organicUsers: Seq[Recommendation],

promotedUsers: Seq[Recommendation]

): Seq[Recommendation] = {

def mergeAccountWithIndex(

organicUsers: Seq[Recommendation],

promotedUsers: Seq[Recommendation],

index: Int

): Stream[Recommendation] = {

if (promotedUsers.isEmpty) organicUsers.toStream

else {

val promotedHead = promotedUsers.head

val promotedTail = promotedUsers.tail

promotedHead.adMetadata match {

case Some(AdMetadata(position, \_)) =>

if (position < 0) mergeAccountWithIndex(organicUsers, promotedTail, index)

else if (position == index)

promotedHead #:: mergeAccountWithIndex(organicUsers, promotedTail, index)

else if (organicUsers.isEmpty) organicUsers.toStream

else {

val organicHead = organicUsers.head

val organicTail = organicUsers.tail

organicHead #:: mergeAccountWithIndex(organicTail, promotedUsers, index + 1)

}

case \_ =>

logger.error("Unknown Candidate type in mergePromotedAccounts")

Stream.empty

}

}

}

mergeAccountWithIndex(organicUsers, promotedUsers, 0)

}

private[this] def blendPromotedAccount(

organic: Seq[Recommendation],

promoted: Seq[Recommendation],

maxResults: Int

): Seq[Recommendation] = {

val merged = mergePromotedAccounts(organic, promoted)

val mergedServed = merged.take(maxResults)

val promotedServed = promoted.intersect(mergedServed)

if (isBlendPromotedNeeded(

mergedServed.size - promotedServed.size,

promotedServed.size,

maxResults

)) {

mergedServed

} else {

organic.take(maxResults)

}

}

@VisibleForTesting

private[blenders] def isBlendPromotedNeeded(

organicSize: Int,

promotedSize: Int,

maxResults: Int

): Boolean = {

(organicSize > 1) &&

(promotedSize > 0) &&

(promotedSize < organicSize) &&

(promotedSize <= 2) &&

(maxResults > 1)

}

}

object PromotedAccountsBlender {

val Name = "promoted\_accounts\_blender"

val InputOrganic = "input\_organic\_accounts"

val InputPromoted = "input\_promoted\_accounts"

val OutputOrganic = "output\_organic\_accounts"

val OutputPromoted = "output\_promoted\_accounts"

val NumPromotedAccounts = "num\_promoted\_accounts"

val MoreThan5Promoted = "more\_than\_5"

}