package com.twitter.cr\_mixer.filter

import com.twitter.cr\_mixer.model.CandidateGeneratorQuery

import com.twitter.cr\_mixer.model.InitialCandidate

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.util.Future

import javax.inject.Inject

import javax.inject.Singleton

@Singleton

class PreRankFilterRunner @Inject() (

impressedTweetListFilter: ImpressedTweetlistFilter,

tweetAgeFilter: TweetAgeFilter,

videoTweetFilter: VideoTweetFilter,

tweetReplyFilter: ReplyFilter,

globalStats: StatsReceiver) {

private val scopedStats = globalStats.scope(this.getClass.getCanonicalName)

/\*\*\*

\* The order of the filters does not matter as long as we do not apply .take(N) truncation

\* across all filters. In other words, it is fine that we first do tweetAgeFilter, and then

\* we do impressedTweetListFilter, or the other way around.

\* Same idea applies to the signal based filter - it is ok that we apply signal based filters

\* before impressedTweetListFilter.

\*

\* We move all signal based filters before tweetAgeFilter and impressedTweetListFilter

\* as a set of early filters.

\*/

val orderedFilters = Seq(

tweetAgeFilter,

impressedTweetListFilter,

videoTweetFilter,

tweetReplyFilter

)

def runSequentialFilters[CGQueryType <: CandidateGeneratorQuery](

request: CGQueryType,

candidates: Seq[Seq[InitialCandidate]],

): Future[Seq[Seq[InitialCandidate]]] = {

PreRankFilterRunner.runSequentialFilters(

request,

candidates,

orderedFilters,

scopedStats

)

}

}

object PreRankFilterRunner {

private def recordCandidateStatsBeforeFilter(

candidates: Seq[Seq[InitialCandidate]],

statsReceiver: StatsReceiver

): Unit = {

statsReceiver

.counter("empty\_sources", "before").incr(

candidates.count { \_.isEmpty }

)

candidates.foreach { candidate =>

statsReceiver.counter("candidates", "before").incr(candidate.size)

}

}

private def recordCandidateStatsAfterFilter(

candidates: Seq[Seq[InitialCandidate]],

statsReceiver: StatsReceiver

): Unit = {

statsReceiver

.counter("empty\_sources", "after").incr(

candidates.count { \_.isEmpty }

)

candidates.foreach { candidate =>

statsReceiver.counter("candidates", "after").incr(candidate.size)

}

}

/\*

Helper function for running some candidates through a sequence of filters

\*/

private[filter] def runSequentialFilters[CGQueryType <: CandidateGeneratorQuery](

request: CGQueryType,

candidates: Seq[Seq[InitialCandidate]],

filters: Seq[FilterBase],

statsReceiver: StatsReceiver

): Future[Seq[Seq[InitialCandidate]]] =

filters.foldLeft(Future.value(candidates)) {

case (candsFut, filter) =>

candsFut.flatMap { cands =>

recordCandidateStatsBeforeFilter(cands, statsReceiver.scope(filter.name))

filter

.filter(cands, filter.requestToConfig(request))

.map { filteredCands =>

recordCandidateStatsAfterFilter(filteredCands, statsReceiver.scope(filter.name))

filteredCands

}

}

}

}