package com.twitter.product\_mixer.component\_library.selector

import com.twitter.product\_mixer.core.functional\_component.common.CandidateScope

import com.twitter.product\_mixer.core.functional\_component.common.SpecificPipelines

import com.twitter.product\_mixer.core.functional\_component.selector.Selector

import com.twitter.product\_mixer.core.functional\_component.selector.SelectorResult

import com.twitter.product\_mixer.core.model.common.identifier.CandidatePipelineIdentifier

import com.twitter.product\_mixer.core.model.common.presentation.CandidateWithDetails

import com.twitter.product\_mixer.core.pipeline.PipelineQuery

import com.twitter.timelines.configapi.Param

import scala.annotation.tailrec

import scala.collection.mutable

import scala.util.Random

/\*\*

\* Select candidates and add them according to the ratio assigned for each [[Bucket]]

\* For instance, if given `Set((A, 0.8), (B, 0.2))` then candidates will randomly be added to the

\* results with an 80% chance of any candidate being from `A` and 20% from`B`. If there are no more

\* candidates from a given `CandidatePipeline` then it's simply skipped, so if we run out of `A`

\* candidates the rest will be `B`. The end result is all candidates from all [[candidatePipelines]]s

\* provided will end up in the result.

\*

\* For example, an output may look like `Seq(A, A, B, A, A)`, `Seq(A, A, A, A, B)`. If we eventually

\* run out of `A` candidates then we would end up with the remaining candidates at the end,

\* `Seq(A, A, B, A, A, A, B, A, A, A [run out of A], B, B, B, B, B, B)`

\*

\* @note the ratios provided are proportional to the sum of all ratios, so if you give 0.3 and 0.7,

\* they will be function as to 30% and 70%, and the same for if you provided 3000 and 7000 for

\* ratios.

\*

\* @note Its important to be sure to update all [[Param]]s when changing the ratio for 1 of them

\* otherwise you may get unexpected results. For instance, of you have 0.3 and 0.7 which

\* correspond to 30% and 70%, and you change `0.7 -> 0.9`, then the total sum of the ratios is

\* now 1.2, so you have 25% and 75% when you intended to have 10% and 90%. To prevent this,

\* be sure to update all [[Param]]s together, so `0.3 -> 0.1` and `0.7 -> 0.9` so the total

\* remains the same.

\*/

case class InsertAppendRatioResults[-Query <: PipelineQuery, Bucket](

candidatePipelines: Set[CandidatePipelineIdentifier],

bucketer: Bucketer[Bucket],

ratios: Map[Bucket, Param[Double]],

random: Random = new Random(0))

extends Selector[Query] {

require(ratios.nonEmpty, "bucketRatios must be non-empty")

override val pipelineScope: CandidateScope = SpecificPipelines(candidatePipelines)

private sealed trait PatternResult

private case object NotASelectedCandidatePipeline extends PatternResult

private case object NotABucketInThePattern extends PatternResult

private case class Bucketed(bucket: Bucket) extends PatternResult

override def apply(

query: Query,

remainingCandidates: Seq[CandidateWithDetails],

result: Seq[CandidateWithDetails]

): SelectorResult = {

val groupedCandidates: Map[PatternResult, Seq[CandidateWithDetails]] =

remainingCandidates.groupBy { candidateWithDetails =>

if (pipelineScope.contains(candidateWithDetails)) {

// if a candidate's Bucket doesnt appear in the pattern it's backfilled at the end

val bucket = bucketer(candidateWithDetails)

if (ratios.contains(bucket)) {

Bucketed(bucket)

} else {

NotABucketInThePattern

}

} else {

NotASelectedCandidatePipeline

}

}

val otherCandidates =

groupedCandidates.getOrElse(NotASelectedCandidatePipeline, Seq.empty)

val notABucketInThePattern =

groupedCandidates.getOrElse(NotABucketInThePattern, Seq.empty)

val groupedCandidatesIterators = groupedCandidates.collect {

case (Bucketed(bucket), candidatesWithDetails) => (bucket, candidatesWithDetails.iterator)

}

// using a LinkedHashMap and sorting by descending ratio

// the highest ratios will always be checked first when iterating

// mutable so we can remove finished ratios when they are finished to optimize looping for large numbers of ratios

val currentBucketRatios: mutable.Map[Bucket, Double] = {

val bucketsAndRatiosSortedByRatio =

ratios.iterator

.map {

case (bucket, param) =>

val ratio = query.params(param)

require(

ratio >= 0,

"The ratio for an InsertAppendRatioResults selector can not be negative")

(bucket, ratio)

}.toSeq

.sortBy { case (\_, ratio) => ratio }(Ordering.Double.reverse)

mutable.LinkedHashMap(bucketsAndRatiosSortedByRatio: \_\*)

}

// keep track of the sum of all ratios so we can look only at random values between 0 and that

var ratioSum = currentBucketRatios.valuesIterator.sum

// add candidates to `newResults` until all remaining candidates are for a single bucket

val newResult = new mutable.ArrayBuffer[CandidateWithDetails]()

while (currentBucketRatios.size > 1) {

// random number between 0 and the sum of the ratios of all params

val randomValue = random.nextDouble() \* ratioSum

val currentBucketRatiosIterator: Iterator[(Bucket, Double)] =

currentBucketRatios.iterator

val (currentBucket, ratio) = currentBucketRatiosIterator.next()

val componentToTakeFrom = findBucketToTakeFrom(

randomValue = randomValue,

cumulativeSumOfRatios = ratio,

bucket = currentBucket,

bucketRatiosIterator = currentBucketRatiosIterator

)

groupedCandidatesIterators.get(componentToTakeFrom) match {

case Some(iteratorForBucket) if iteratorForBucket.nonEmpty =>

newResult += iteratorForBucket.next()

case \_ =>

ratioSum -= currentBucketRatios(componentToTakeFrom)

currentBucketRatios.remove(componentToTakeFrom)

}

}

// with only have 1 source remaining, we can skip all the above work and insert them in bulk

val remainingBucketInRatio =

currentBucketRatios.keysIterator.flatMap(groupedCandidatesIterators.get).flatten

SelectorResult(

remainingCandidates = otherCandidates,

result = result ++ newResult ++ remainingBucketInRatio ++ notABucketInThePattern)

}

/\*\*

\* iterates through the `bucketRatiosIterator` until it finds a the

\* [[Bucket]] that corresponds with the current `randomValue`.

\*

\* This method expects that `0 <= randomValue <= sum of all ratios`

\*

\* @example If the given ratios are `Seq(A -> 0.2, B -> 0.35, C -> 0.45)`

\* check if the given `randomValue` is

\* - `< 0.45`, if not then check

\* - `< 0.8` (0.45 + 0.35), if not then check

\* - `< 1.0` (0.45 + 0.35 + 0.2)

\*

\* and return the corresponding [[Bucket]]

\*/

@tailrec private def findBucketToTakeFrom(

randomValue: Double,

cumulativeSumOfRatios: Double,

bucket: Bucket,

bucketRatiosIterator: Iterator[(Bucket, Double)]

): Bucket = {

if (randomValue < cumulativeSumOfRatios || bucketRatiosIterator.isEmpty) {

bucket

} else {

val (nextBucket, ratio) = bucketRatiosIterator.next()

findBucketToTakeFrom(

randomValue,

cumulativeSumOfRatios + ratio,

nextBucket,

bucketRatiosIterator)

}

}

}