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

import com.twitter.product\_mixer.component\_library.selector.InsertRandomPositionResults.randomIndices

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

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

import com.twitter.product\_mixer.core.functional\_component.configapi.StaticParam

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.presentation.CandidateWithDetails

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

import com.twitter.timelines.configapi.Param

import scala.collection.mutable

import scala.util.Random

object InsertRandomPositionResults {

/\*\*

\* Iterator containing random index between `startIndex` and `endIndex` + `n`

\* where `n` is the number of times `next` has been called on the iterator

\* without duplication

\*/

private[selector] def randomIndices(

resultLength: Int,

startIndex: Int,

endIndex: Int,

random: Random

): Iterator[Int] = {

/\*\* exclusive because [[Random.nextInt]]'s bound is exclusive \*/

val indexUpperBound = Math.min(endIndex, resultLength)

/\*\*

\* keep track of the available indices, `O(n)` space where `n` is `min(endIndex, resultLength) - max(startIndex, 0)`

\* this ensures fairness which duplicate indices could otherwise skew

\*/

val values = mutable.ArrayBuffer(Math.max(0, startIndex) to indexUpperBound: \_\*)

/\*\*

\* Iterator that starts at 1 above the last valid index, [[indexUpperBound]] + 1, and increments monotonically

\* representing the new highest index possible in the results for the next call

\*/

Iterator

.from(indexUpperBound + 1)

.map { indexUpperBound =>

/\*\*

\* pick a random index-to-insert-candidate-into-results from [[values]] replacing the value at

\* the chosen index with the new highest index from [[indexUpperBound]], this results in

\* constant time for picking the random index and adding the new highest valid index instead

\* of removing the item from the middle and appending the new, which would be `O(n)` to shift

\* all indices after the removal point

\*/

val i = random.nextInt(values.length)

val randomIndexToUse = values(i)

// override the value at i with the new `upperBoundExclusive` to account for the new index value in the next iteration

values(i) = indexUpperBound

randomIndexToUse

}

}

}

sealed trait InsertedCandidateOrder

/\*\*

\* Candidates from the `remainingCandidates` side will be inserted in a random order into the `result`

\*

\* @example if inserting `[ x, y, z ]` into the `result` then the relative positions of `x`, `y` and `z`

\* to each other is random, e.g. `y` could come before `x` in the result.

\*/

case object UnstableOrderingOfInsertedCandidates extends InsertedCandidateOrder

/\*\*

\* Candidates from the `remainingCandidates` side will be inserted in their original order into the `result`

\*

\* @example if inserting `[ x, y, z ]` into the `result` then the relative positions of `x`, `y` and `z`

\* to each other will remain the same, e.g. `x` is always before `y` is always before `z` in the final result

\*/

case object StableOrderingOfInsertedCandidates extends InsertedCandidateOrder

/\*\*

\* Insert `remainingCandidates` into a random position between the specified indices (inclusive)

\*

\* @example let `result` = `[ a, b, c, d ]` and we want to insert randomly `[ x, y, z ]`

\* with `startIndex` = 1, `endIndex` = 2, and [[UnstableOrderingOfInsertedCandidates]].

\* We can expect a result that looks like `[ a, ... , d ]` where `...` is

\* a random insertion of `x`, `y`, and `z` into `[ b, c ]`. So this could look like

\* `[ a, y, b, x, c, z, d ]`, note that the inserted elements are randomly distributed

\* among the elements that were originally between the specified indices.

\* This functions like taking a slice of the original `result` between the indices,

\* e.g. `[ b, c ]`, then randomly inserting into the slice, e.g. `[ y, b, x, c, z ]`,

\* before reassembling the `result`, e.g. `[ a ] ++ [ y, b, x, c, z ] ++ [ d ]`.

\*

\* @example let `result` = `[ a, b, c, d ]` and we want to insert randomly `[ x, y, z ]`

\* with `startIndex` = 1, `endIndex` = 2, and [[StableOrderingOfInsertedCandidates]].

\* We can expect a result that looks something like `[ a, x, b, y, c, z, d ]`,

\* where `x` is before `y` which is before `z`

\*

\* @param startIndex an inclusive index which starts the range where the candidates will be inserted

\* @param endIndex an inclusive index which ends the range where the candidates will be inserted

\*/

case class InsertRandomPositionResults[-Query <: PipelineQuery](

pipelineScope: CandidateScope,

remainingCandidateOrder: InsertedCandidateOrder,

startIndex: Param[Int] = StaticParam(0),

endIndex: Param[Int] = StaticParam(Int.MaxValue),

random: Random = new Random(0))

extends Selector[Query] {

override def apply(

query: Query,

remainingCandidates: Seq[CandidateWithDetails],

result: Seq[CandidateWithDetails]

): SelectorResult = {

val PartitionedCandidates(candidatesInScope, candidatesOutOfScope) =

pipelineScope.partition(remainingCandidates)

val randomIndexIterator = {

val randomIndexIterator =

randomIndices(result.length, query.params(startIndex), query.params(endIndex), random)

remainingCandidateOrder match {

case StableOrderingOfInsertedCandidates =>

randomIndexIterator.take(candidatesInScope.length).toSeq.sorted.iterator

case UnstableOrderingOfInsertedCandidates =>

randomIndexIterator

}

}

val mergedResult = DynamicPositionSelector.mergeByIndexIntoResult(

candidatesToInsertByIndex = randomIndexIterator.zip(candidatesInScope.iterator).toSeq,

result = result,

DynamicPositionSelector.AbsoluteIndices

)

SelectorResult(remainingCandidates = candidatesOutOfScope, result = mergedResult)

}

}