package com.twitter.graph\_feature\_service.util

import com.twitter.graph\_feature\_service.thriftscala.{

FeatureType,

IntersectionValue,

WorkerIntersectionValue

}

import java.nio.ByteBuffer

import scala.collection.mutable.ArrayBuffer

/\*\*

\* Functions for computing feature values based on the values returned by constantDB.

\*/

object IntersectionValueCalculator {

/\*\*

\* Compute the size of the array in a ByteBuffer.

\* Note that this function assumes the ByteBuffer is encoded using Injections.seqLong2ByteBuffer

\*/

def computeArraySize(x: ByteBuffer): Int = {

x.remaining() >> 3 // divide 8

}

/\*\*

\*

\*/

def apply(x: ByteBuffer, y: ByteBuffer, intersectionIdLimit: Int): WorkerIntersectionValue = {

val xSize = computeArraySize(x)

val ySize = computeArraySize(y)

val largerArray = if (xSize > ySize) x else y

val smallerArray = if (xSize > ySize) y else x

if (intersectionIdLimit == 0) {

val result = computeIntersectionUsingBinarySearchOnLargerByteBuffer(smallerArray, largerArray)

WorkerIntersectionValue(result, xSize, ySize)

} else {

val (result, ids) = computeIntersectionWithIds(smallerArray, largerArray, intersectionIdLimit)

WorkerIntersectionValue(result, xSize, ySize, ids)

}

}

/\*\*

\* Note that this function assumes the ByteBuffer is encoded using Injections.seqLong2ByteBuffer

\*

\*/

def computeIntersectionUsingBinarySearchOnLargerByteBuffer(

smallArray: ByteBuffer,

largeArray: ByteBuffer

): Int = {

var res: Int = 0

var i: Int = 0

while (i < smallArray.remaining()) {

if (binarySearch(largeArray, smallArray.getLong(i)) >= 0) {

res += 1

}

i += 8

}

res

}

def computeIntersectionWithIds(

smallArray: ByteBuffer,

largeArray: ByteBuffer,

intersectionLimit: Int

): (Int, Seq[Long]) = {

var res: Int = 0

var i: Int = 0

// Most of the intersectionLimit is smaller than default size: 16

val idBuffer = ArrayBuffer[Long]()

while (i < smallArray.remaining()) {

val value = smallArray.getLong(i)

if (binarySearch(largeArray, value) >= 0) {

res += 1

// Always get the smaller ids

if (idBuffer.size < intersectionLimit) {

idBuffer += value

}

}

i += 8

}

(res, idBuffer)

}

/\*\*

\* Note that this function assumes the ByteBuffer is encoded using Injections.seqLong2ByteBuffer

\*

\*/

private[util] def binarySearch(arr: ByteBuffer, value: Long): Int = {

var start = 0

var end = arr.remaining()

while (start <= end && start < arr.remaining()) {

val mid = ((start + end) >> 1) & ~7 // take mid - mid % 8

if (arr.getLong(mid) == value) {

return mid // return the index of the value

} else if (arr.getLong(mid) < value) {

start = mid + 8

} else {

end = mid - 1

}

}

// if not existed, return -1

-1

}

/\*\*

\* TODO: for now it only computes intersection size. Will add more feature types (e.g., dot

\* product, maximum value).

\*

\* NOTE that this function assumes both x and y are SORTED arrays.

\* In graph feature service, the sorting is done in the offline Scalding job.

\*

\* @param x source user's array

\* @param y candidate user's array

\* @param featureType feature type

\* @return

\*/

def apply(x: Array[Long], y: Array[Long], featureType: FeatureType): IntersectionValue = {

val xSize = x.length

val ySize = y.length

val intersection =

if (xSize.min(ySize) \* math.log(xSize.max(ySize)) < (xSize + ySize).toDouble) {

if (xSize < ySize) {

computeIntersectionUsingBinarySearchOnLargerArray(x, y)

} else {

computeIntersectionUsingBinarySearchOnLargerArray(y, x)

}

} else {

computeIntersectionUsingListMerging(x, y)

}

IntersectionValue(

featureType,

Some(intersection.toInt),

None, // return None for now

Some(xSize),

Some(ySize)

)

}

/\*\*

\* Function for computing the intersections of two SORTED arrays by list merging.

\*

\* @param x one array

\* @param y another array

\* @param ordering ordering function for comparing values of T

\* @tparam T type

\* @return The intersection size and the list of intersected elements

\*/

private[util] def computeIntersectionUsingListMerging[T](

x: Array[T],

y: Array[T]

)(

implicit ordering: Ordering[T]

): Int = {

var res: Int = 0

var i: Int = 0

var j: Int = 0

while (i < x.length && j < y.length) {

val comp = ordering.compare(x(i), y(j))

if (comp > 0) j += 1

else if (comp < 0) i += 1

else {

res += 1

i += 1

j += 1

}

}

res

}

/\*\*

\* Function for computing the intersections of two arrays by binary search on the larger array.

\* Note that the larger array MUST be SORTED.

\*

\* @param smallArray smaller array

\* @param largeArray larger array

\* @param ordering ordering function for comparing values of T

\* @tparam T type

\*

\* @return The intersection size and the list of intersected elements

\*/

private[util] def computeIntersectionUsingBinarySearchOnLargerArray[T](

smallArray: Array[T],

largeArray: Array[T]

)(

implicit ordering: Ordering[T]

): Int = {

var res: Int = 0

var i: Int = 0

while (i < smallArray.length) {

val currentValue: T = smallArray(i)

if (binarySearch(largeArray, currentValue) >= 0) {

res += 1

}

i += 1

}

res

}

/\*\*

\* Function for doing the binary search

\*

\* @param arr array

\* @param value the target value for searching

\* @param ordering ordering function

\* @tparam T type

\* @return the index of element in the larger array.

\* If there is no such element in the array, return -1.

\*/

private[util] def binarySearch[T](

arr: Array[T],

value: T

)(

implicit ordering: Ordering[T]

): Int = {

var start = 0

var end = arr.length - 1

while (start <= end) {

val mid = (start + end) >> 1

val comp = ordering.compare(arr(mid), value)

if (comp == 0) {

return mid // return the index of the value

} else if (comp < 0) {

start = mid + 1

} else {

end = mid - 1

}

}

// if not existed, return -1

-1

}

}