package com.twitter.simclusters\_v2.scalding

import com.twitter.algebird.Max

import com.twitter.algebird.Monoid

import com.twitter.bijection.scrooge.BinaryScalaCodec

import com.twitter.hermit.candidate.thriftscala.Candidate

import com.twitter.hermit.candidate.thriftscala.Candidates

import com.twitter.logging.Logger

import com.twitter.pluck.source.cassowary.FollowingsCosineSimilaritiesManhattanSource

import com.twitter.sbf.core.AlgorithmConfig

import com.twitter.sbf.core.MHAlgorithm

import com.twitter.sbf.core.PredictionStat

import com.twitter.sbf.core.SparseBinaryMatrix

import com.twitter.sbf.core.SparseRealMatrix

import com.twitter.sbf.graph.Graph

import com.twitter.scalding.\_

import com.twitter.scalding.commons.source.VersionedKeyValSource

import com.twitter.scalding\_internal.dalv2.DAL

import com.twitter.scalding\_internal.job.TwitterExecutionApp

import com.twitter.scalding\_internal.source.lzo\_scrooge.FixedPathLzoScrooge

import com.twitter.simclusters\_v2.scalding.common.TypedRichPipe.\_

import com.twitter.usersource.snapshot.flat.UsersourceFlatScalaDataset

import com.twitter.usersource.snapshot.flat.thriftscala.FlatUser

import com.twitter.wtf.scalding.sims.thriftscala.SimilarUserPair

import java.io.PrintWriter

import java.text.DecimalFormat

import java.util

import org.apache.hadoop.conf.Configuration

import org.apache.hadoop.fs.FileSystem

import org.apache.hadoop.fs.Path

import scala.collection.JavaConverters.\_

case class TopUser(id: Long, activeFollowerCount: Int, screenName: String)

case class TopUserWithMappedId(topUser: TopUser, mappedId: Int)

case class AdjList(sourceId: Long, neighbors: List[(Long, Float)])

object TopUsersSimilarityGraph {

val log = Logger()

def topUsers(

userSourcePipe: TypedPipe[FlatUser],

minActiveFollowers: Int,

topK: Int

): TypedPipe[TopUser] = {

userSourcePipe

.collect {

case f: FlatUser

if f.activeFollowers.exists(\_ >= minActiveFollowers)

&& f.followers.isDefined && f.id.isDefined && f.screenName.isDefined

&& !f.deactivated.contains(true) && !f.suspended.contains(true) =>

TopUser(f.id.get, f.activeFollowers.get.toInt, f.screenName.get)

}

.groupAll

.sortedReverseTake(topK)(Ordering.by(\_.activeFollowerCount))

.values

.flatten

}

/\*\*

\* This function returns the top most followed userIds truncated to topK

\* Offers the same functionality as TopUsersSimilarityGraph.topUsers but more efficient

\* as we donot store screennames while grouping and sorting the users

\*/

def topUserIds(

userSourcePipe: TypedPipe[FlatUser],

minActiveFollowers: Int,

topK: Int

): TypedPipe[Long] = {

userSourcePipe

.collect {

case f: FlatUser

if f.activeFollowers.exists(\_ >= minActiveFollowers)

&& f.followers.isDefined && f.id.isDefined && f.screenName.isDefined

&& !f.deactivated.contains(true) && !f.suspended.contains(true) =>

(f.id.get, f.activeFollowers.get)

}

.groupAll

.sortedReverseTake(topK)(Ordering.by(\_.\_2))

.values

.flatten

.keys

}

def topUsersWithMappedIds(

userSourcePipe: TypedPipe[FlatUser],

minActiveFollowers: Int

): TypedPipe[TopUserWithMappedId] = {

userSourcePipe

.collect {

case f: FlatUser

if f.activeFollowers.exists(\_ >= minActiveFollowers)

&& f.followers.isDefined && f.id.isDefined && f.screenName.isDefined

&& !f.deactivated.contains(true) && !f.suspended.contains(true) =>

TopUser(f.id.get, f.activeFollowers.get.toInt, f.screenName.get)

}

.groupAll

.mapGroup {

case (\_, topUserIter) =>

topUserIter.zipWithIndex.map {

case (topUser, id) =>

TopUserWithMappedId(topUser, id)

}

}

.values

}

def topUsersWithMappedIdsTopK(

userSourcePipe: TypedPipe[FlatUser],

minActiveFollowers: Int,

topK: Int

): TypedPipe[TopUserWithMappedId] = {

userSourcePipe

.collect {

case f: FlatUser

if f.activeFollowers.exists(\_ >= minActiveFollowers)

&& f.followers.isDefined && f.id.isDefined && f.screenName.isDefined

&& !f.deactivated.contains(true) && !f.suspended.contains(true) =>

TopUser(f.id.get, f.activeFollowers.get.toInt, f.screenName.get)

}

.groupAll

.sortedReverseTake(topK)(Ordering.by(\_.activeFollowerCount))

.map {

case (\_, topUserIter) =>

topUserIter.zipWithIndex.map {

case (topUser, id) =>

TopUserWithMappedId(topUser, id)

}

}

.flatten

}

/\*\*

\* This function returns the top most followed and verified userIds truncated to topK

\*/

def vits(

userSourcePipe: TypedPipe[FlatUser],

minActiveFollowers: Int,

topK: Int

): TypedPipe[Long] = {

userSourcePipe

.collect {

case f: FlatUser

if f.verified.contains(true) && f.id.isDefined &&

f.screenName.isDefined && !f.deactivated.contains(true) && !f.suspended.contains(

true) &&

f.activeFollowers.exists(\_ >= minActiveFollowers) =>

(f.id.get, f.activeFollowers.get)

}

.groupAll

.sortedReverseTake(topK)(Ordering.by(\_.\_2))

.values

.flatten

.keys

}

def topUsersInMemory(

userSourcePipe: TypedPipe[FlatUser],

minActiveFollowers: Int,

topK: Int

): Execution[List[TopUserWithMappedId]] = {

log.info(s"Will fetch top $topK users with at least $minActiveFollowers many active followers")

topUsers(userSourcePipe, minActiveFollowers, topK).toIterableExecution

.map { idFollowersList =>

idFollowersList.toList.sortBy(\_.id).zipWithIndex.map {

case (topuser, index) =>

TopUserWithMappedId(topuser, index)

}

}

}

def addSelfLoop(

input: TypedPipe[(Long, Map[Long, Float])],

maxToSelfLoopWeight: Float => Float

): TypedPipe[(Long, Map[Long, Float])] = {

input

.map {

case (nodeId, neighborMap) if neighborMap.nonEmpty =>

val maxEntry = neighborMap.values.max

val selfLoopWeight = maxToSelfLoopWeight(maxEntry)

(nodeId, neighborMap ++ Map(nodeId -> selfLoopWeight))

case (nodeId, emptyMap) =>

(nodeId, emptyMap)

}

}

def makeGraph(

backfillPipe: TypedPipe[(Long, Map[Long, Float])],

dirToReadFromOrSaveTo: String

): Execution[TypedPipe[(Long, Map[Long, Float])]] = {

backfillPipe

.map {

case (nodeId, nbrMap) =>

val cands = nbrMap.toList.map { case (nId, wt) => Candidate(nId, wt) }

Candidates(nodeId, candidates = cands)

}

.make(new FixedPathLzoScrooge(dirToReadFromOrSaveTo, Candidates))

.map { tp =>

tp.map {

case Candidates(nodeId, cands) =>

(nodeId, cands.map { case Candidate(nId, wt, \_) => (nId, wt.toFloat) }.toMap)

}

}

}

def getSubgraphFromUserGroupedInput(

fullGraph: TypedPipe[Candidates],

usersToInclude: TypedPipe[Long],

maxNeighborsPerNode: Int,

degreeThresholdForStat: Int

)(

implicit uniqId: UniqueID

): TypedPipe[(Long, Map[Long, Float])] = {

val numUsersWithZeroEdges = Stat("num\_users\_with\_zero\_edges")

val numUsersWithSmallDegree = Stat("num\_users\_with\_degree\_lt\_" + degreeThresholdForStat)

val numUsersWithEnoughDegree = Stat("num\_users\_with\_degree\_gte\_" + degreeThresholdForStat)

fullGraph

.map { cands =>

(

cands.userId,

// These candidates are already sorted, but leaving it in just in case the behavior changes upstream

cands.candidates

.map { c => (c.userId, c.score) }.sortBy(-\_.\_2).take(maxNeighborsPerNode).toMap

)

}

.rightJoin(usersToInclude.asKeys)

// uncomment for adhoc job

//.withReducers(110)

.mapValues(\_.\_1) // discard the Unit

.toTypedPipe

.count("num\_sims\_records\_from\_top\_users")

.flatMap {

case (nodeId, Some(neighborMap)) =>

neighborMap.flatMap {

case (neighborId, edgeWt) =>

List(

(nodeId, Map(neighborId -> Max(edgeWt.toFloat))),

(neighborId, Map(nodeId -> Max(edgeWt.toFloat)))

)

}

case (nodeId, None) => List((nodeId, Map.empty[Long, Max[Float]]))

}

.sumByKey

// uncomment for adhoc job

//.withReducers(150)

.toTypedPipe

.mapValues(\_.mapValues(\_.get)) // get the max for each value in each map

.count("num\_sims\_records\_after\_symmetrization\_before\_keeping\_only\_top\_users")

.join(usersToInclude.asKeys) // only keep records for top users

// uncomment for adhoc job

//.withReducers(100)

.mapValues(\_.\_1)

.toTypedPipe

.map {

case (nodeId, neighborsMap) =>

if (neighborsMap.nonEmpty) {

if (neighborsMap.size < degreeThresholdForStat) {

numUsersWithSmallDegree.inc()

} else {

numUsersWithEnoughDegree.inc()

}

} else {

numUsersWithZeroEdges.inc()

}

(nodeId, neighborsMap)

}

.count("num\_sims\_records\_after\_symmetrization\_only\_top\_users")

}

def getSubgraphFromUserGroupedInput(

fullGraph: TypedPipe[Candidates],

usersToInclude: Set[Long],

maxNeighborsPerNode: Int

)(

implicit uniqId: UniqueID

): TypedPipe[(Long, Map[Long, Float])] = {

val numUsersWithZeroEdges = Stat("num\_users\_with\_zero\_edges")

val numUsersWithDegreeLessThan10 = Stat("num\_users\_with\_degree\_less\_than\_10")

val (intIdsToIncludeSorted: Array[Int], longIdsToIncludeSorted: Array[Long]) =

setToSortedArrays(usersToInclude)

log.info("Size of intArray " + intIdsToIncludeSorted.length)

log.info("Size of longArray " + longIdsToIncludeSorted.length)

fullGraph

.collect {

case candidates

if isIdInIntOrLongArray(

candidates.userId,

intIdsToIncludeSorted,

longIdsToIncludeSorted) =>

val sourceId = candidates.userId

val toKeep = candidates.candidates.collect {

case neighbor

if isIdInIntOrLongArray(

neighbor.userId,

intIdsToIncludeSorted,

longIdsToIncludeSorted) =>

(neighbor.userId, neighbor.score.toFloat)

}.toList

val toKeepLength = toKeep.size

if (toKeep.isEmpty) {

numUsersWithZeroEdges.inc()

} else if (toKeepLength < 10) {

numUsersWithDegreeLessThan10.inc()

}

val knn = if (toKeepLength > maxNeighborsPerNode) {

toKeep.sortBy(\_.\_2).takeRight(maxNeighborsPerNode)

} else toKeep

knn.flatMap {

case (nbrId, wt) =>

List(

(sourceId, Map(nbrId -> Max(wt))),

(nbrId, Map(sourceId -> Max(wt)))

)

}

}

.flatten

.sumByKey

.toTypedPipe

.mapValues(\_.mapValues(\_.get)) // get the max for each value in each map

}

def getInMemorySubgraphFromUserGroupedInput(

fullGraph: TypedPipe[Candidates],

usersToInclude: Set[Long],

maxNeighborsPerNode: Int

)(

implicit uniqId: UniqueID

): Execution[Iterable[AdjList]] = {

getSubgraphFromUserGroupedInput(fullGraph, usersToInclude, maxNeighborsPerNode).map {

case (sourceId, weightedNeighbors) =>

AdjList(

sourceId,

weightedNeighbors.toList.sortBy(\_.\_1)

)

}.toIterableExecution

}

def isIdInIntOrLongArray(

id: Long,

intArraySorted: Array[Int],

longArraySorted: Array[Long]

): Boolean = {

if (id < Integer.MAX\_VALUE) {

util.Arrays.binarySearch(intArraySorted, id.toInt) >= 0

} else {

util.Arrays.binarySearch(longArraySorted, id.toLong) >= 0

}

}

/\*\*

\* Creates two sorted arrays out of a set, one with ints and one with longs.

\* Sorted arrays are only slightly more expensive to search in, but empirically I've found

\* that the MapReduce job runs more reliably using them than using Set directly.

\*

\* @param inSet

\*

\* @return

\*/

def setToSortedArrays(inSet: Set[Long]): (Array[Int], Array[Long]) = {

val (intArrayUnconvertedSorted, longArraySorted) =

inSet.toArray.sorted.partition { l => l < Integer.MAX\_VALUE }

(intArrayUnconvertedSorted.map(\_.toInt), longArraySorted)

}

def getInMemorySubgraph(

fullGraph: TypedPipe[SimilarUserPair],

usersToInclude: Set[Long],

maxNeighborsPerNode: Int

)(

implicit uniqId: UniqueID

): Execution[Iterable[AdjList]] = {

val numValidEdges = Stat("num\_valid\_edges")

val numInvalidEdges = Stat("num\_invalid\_edges")

val (intIdsToIncludeSorted: Array[Int], longIdsToIncludeSorted: Array[Long]) =

setToSortedArrays(usersToInclude)

log.info("Size of intArray " + intIdsToIncludeSorted.length)

log.info("Size of longArray " + longIdsToIncludeSorted.length)

fullGraph

.filter { edge =>

val res =

isIdInIntOrLongArray(edge.sourceId, intIdsToIncludeSorted, longIdsToIncludeSorted) &&

isIdInIntOrLongArray(edge.destinationId, intIdsToIncludeSorted, longIdsToIncludeSorted)

if (res) {

numValidEdges.inc()

} else {

numInvalidEdges.inc()

}

res

}

.map { edge => (edge.sourceId, (edge.destinationId, edge.cosineScore.toFloat)) }

.group

.sortedReverseTake(maxNeighborsPerNode)(Ordering.by(\_.\_2))

.toTypedPipe

.flatMap {

case (sourceId, weightedNeighbors) =>

weightedNeighbors.flatMap {

case (destId, wt) =>

/\*

By default, a k-nearest neighbor graph need not be symmetric, since if u is in v's

k nearest neighbors, that doesn't guarantee that v is in u's.

This step adds edges in both directions, but having a Map ensures that each neighbor

only appears once and not twice. Using Max() operator from Algebird, we take the max

weight of (u, v) and (v, u) - it is expected that the two will be pretty much the same.

Example illustrating how Map and Max work together:

Map(1 -> Max(2)) + Map(1 -> Max(3)) = Map(1 -> Max(3))

\*/

List(

(sourceId, Map(destId -> Max(wt))),

(destId, Map(sourceId -> Max(wt)))

)

}

}

.sumByKey

.map {

case (sourceId, weightedNeighbors) =>

AdjList(

sourceId,

weightedNeighbors.toList.map { case (id, maxWt) => (id, maxWt.get) }.sortBy(\_.\_1)

)

}

.toIterableExecution

}

def convertIterableToGraph(

adjList: Iterable[AdjList],

verticesMapping: Map[Long, Int],

wtExponent: Float

): Graph = {

val n = verticesMapping.size

val neighbors: Array[Array[Int]] = new Array[Array[Int]](n)

val wts: Array[Array[Float]] = new Array[Array[Float]](n)

var numEdges = 0L

var numVertices = 0

val iter = adjList.iterator

val verticesWithAtleastOneEdgeBuilder = Set.newBuilder[Long]

while (iter.hasNext) {

val AdjList(originalId, wtedNeighbors) = iter.next()

val wtedNeighborsSize = wtedNeighbors.size

val newId = verticesMapping(originalId) // throw exception if originalId not in map

if (newId < 0 || newId >= n) {

throw new IllegalStateException(

s"$originalId has been mapped to $newId, which is outside" +

s"the expected range [0, " + (n - 1) + "]")

}

verticesWithAtleastOneEdgeBuilder += originalId

neighbors(newId) = new Array[Int](wtedNeighborsSize)

wts(newId) = new Array[Float](wtedNeighborsSize)

wtedNeighbors.zipWithIndex.foreach {

case ((nbrId, wt), index) =>

neighbors(newId)(index) = verticesMapping(nbrId)

wts(newId)(index) = wt

numEdges += 1

}

if (math.abs(wtExponent - 1.0) > 1e-5) {

var maxWt = Float.MinValue

for (index <- wts(newId).indices) {

wts(newId)(index) = math.pow(wts(newId)(index), wtExponent).toFloat

if (wts(newId)(index) > maxWt) {

maxWt = wts(newId)(index)

}

}

}

numVertices += 1

if (numVertices % 100000 == 0) {

log.info(s"Done with $numVertices many vertices.")

}

}

val verticesWithAtleastOneEdge = verticesWithAtleastOneEdgeBuilder.result()

val verticesWithZeroEdges = verticesMapping.keySet.diff(verticesWithAtleastOneEdge)

verticesWithZeroEdges.foreach { originalId =>

neighbors(verticesMapping(originalId)) = new Array[Int](0)

wts(verticesMapping(originalId)) = new Array[Float](0)

}

log.info("Number of vertices with zero edges " + verticesWithZeroEdges.size)

log.info("Number of edges " + numEdges)

if (verticesWithZeroEdges.nonEmpty) {

log.info("The vertices with zero edges: " + verticesWithZeroEdges.mkString(","))

}

new Graph(n, numEdges / 2, neighbors, wts)

}

def run(

userSourcePipe: TypedPipe[FlatUser],

minActiveFollowers: Int,

topK: Int,

getSubgraphFn: Set[Long] => Execution[Iterable[AdjList]],

wtExponent: Float

)(

implicit id: UniqueID

): Execution[(List[TopUserWithMappedId], Graph)] = {

topUsersInMemory(

userSourcePipe,

minActiveFollowers,

topK

).flatMap { topUsers =>

val idMap = topUsers.map { topUser => (topUser.topUser.id, topUser.mappedId) }.toMap

log.info("Got idMap with " + idMap.size + " entries.")

getSubgraphFn(idMap.keySet)

.map { iterableAdjLists =>

log.info("Going to convert iterable to graph")

val tic = System.currentTimeMillis()

val graph = convertIterableToGraph(

iterableAdjLists,

idMap,

wtExponent

)

val toc = System.currentTimeMillis()

val seconds = (toc - tic) \* 1.0 / 1e6

log.info("Took %.2f seconds to convert iterable to graph".format(seconds))

(topUsers, graph)

}

}

}

def runUsingJoin(

mappedUsers: TypedPipe[(Long, Int)],

allEdges: TypedPipe[Candidates],

maxNeighborsPerNode: Int

)(

implicit uniqueID: UniqueID

): TypedPipe[(Int, String)] = {

val numEdgesAfterFirstJoin = Stat("num\_edges\_after\_first\_join")

val numEdgesAfterSecondJoin = Stat("num\_edges\_after\_second\_join")

val numEdgesLostTopKTruncated = Stat("num\_edges\_lost\_topk\_truncated")

val finalNumEdges = Stat("final\_num\_edges")

allEdges

.map { cs => (cs.userId, cs.candidates) }

.join(mappedUsers)

.withReducers(6000)

.flatMap {

case (id, (neighbors, mappedId)) =>

val before = neighbors.size

val topKNeighbors = neighbors.sortBy(-\_.score).take(maxNeighborsPerNode)

val after = topKNeighbors.size

numEdgesLostTopKTruncated.incBy(before - after)

topKNeighbors.map { candidate =>

numEdgesAfterFirstJoin.inc()

(candidate.userId, (mappedId, candidate.score.toFloat))

}

}

.join(mappedUsers)

.withReducers(9000)

.flatMap {

case (id, ((mappedNeighborId, score), mappedId)) =>

numEdgesAfterSecondJoin.inc()

List(

(mappedId, Map(mappedNeighborId -> Max(score))),

(mappedNeighborId, Map(mappedId -> Max(score)))

)

}

.sumByKey

.withReducers(9100)

.map {

case (id, nbrMap) =>

val sorted = nbrMap.mapValues(\_.get).toList.sortBy(-\_.\_2)

finalNumEdges.incBy(sorted.size)

val str = sorted.map { case (nbrId, wt) => "%d %.2f".format(nbrId, wt) }.mkString(" ")

(id, str)

}

}

def writeToHDFSFile(lines: Iterator[String], conf: Configuration, outputFile: String): Unit = {

val fs = FileSystem.newInstance(conf)

val outputStream = fs.create(new Path(outputFile))

log.info("Will write to " + outputFile)

var numLines = 0

val tic = System.currentTimeMillis()

try {

val writer = new PrintWriter(outputStream)

while (lines.hasNext) {

writer.println(lines.next())

numLines += 1

if (numLines % 1000000 == 0) {

log.info(s"Done writing $numLines lines")

}

}

writer.flush()

writer.close()

} finally {

outputStream.close()

}

val toc = System.currentTimeMillis()

val seconds = (toc - tic) \* 1.0 / 1e6

log.info(

"Finished writing %d lines to %s. Took %.2f seconds".format(numLines, outputFile, seconds))

}

def writeToHDFSIfHDFS(lines: Iterator[String], mode: Mode, outputFile: String): Unit = {

mode match {

case Hdfs(\_, conf) =>

writeToHDFSFile(lines, conf, outputFile)

case \_ => ()

}

}

def writeTopUsers(topUsers: List[TopUserWithMappedId], mode: Mode, outputFile: String): Unit = {

val topUsersLines =

topUsers.map { topUser =>

// Add 1 to mappedId so as to get 1-indexed ids, which are friendlier to humans.

List(

topUser.topUser.id,

topUser.mappedId + 1,

topUser.topUser.screenName,

topUser.topUser.activeFollowerCount

).mkString("\t")

}.iterator

writeToHDFSIfHDFS(topUsersLines, mode, outputFile)

}

def readSimsInput(isKeyValSource: Boolean, inputDir: String): TypedPipe[Candidates] = {

if (isKeyValSource) {

log.info("Will treat " + inputDir + " as SequenceFiles input")

val rawInput = FollowingsCosineSimilaritiesManhattanSource(path = inputDir)

TypedPipe.from(rawInput).map(\_.\_2)

} else {

log.info("Will treat " + inputDir + " as LzoScrooge input")

TypedPipe.from(new FixedPathLzoScrooge(inputDir, Candidates))

}

}

}

/\*\*

\* ./bazel bundle src/scala/com/twitter/simclusters\_v2/scalding:top\_users\_only && \

\* oscar hdfs --hadoop-client-memory 120000 --user cassowary --host atla-aor-08-sr1 \

\* --bundle top\_users\_only --tool com.twitter.simclusters\_v2.scalding.ClusterHdfsGraphApp \

\* --screen --screen-detached --tee ldap\_logs/SBFOnSubGraphOf100MTopusersWithMappedIds\_120GB\_RAM \

\* -- --inputDir adhoc/ldap\_subgraphOf100MTopUsersWithMappedIds --numNodesPerCommunity 200 \

\* --outputDir adhoc/ldap\_SBFOnSubGraphOf100MTopusersWithMappedIds\_k500K\_120GB\_RAM --assumedNumberOfNodes 100200000

\*/

object ClusterHdfsGraphApp extends TwitterExecutionApp {

def job: Execution[Unit] =

Execution.getConfigMode.flatMap {

case (config, mode) =>

Execution.withId { implicit uniqueId =>

val args = config.getArgs

val inputDir = args("inputDir")

val numNodesPerCommunity = args.int("numNodesPerCommunity", 200)

val outputDir = args("outputDir")

val assumedNumberOfNodes = args.int("assumedNumberOfNodes")

//val useEdgeWeights = args.boolean("useEdgeWeights")

val input = TypedPipe.from(TypedTsv[(Int, String)](inputDir)).map {

case (id, nbrStr) =>

val nbrsWithWeights = nbrStr.split(" ")

val nbrsArray = nbrsWithWeights.zipWithIndex

.collect {

case (str, index) if index % 2 == 0 =>

str.toInt

}

(id, nbrsArray.sorted)

}

println("Gonna assume total number of nodes is " + assumedNumberOfNodes)

input.toIterableExecution.flatMap { adjListsIter =>

val nbrs: Array[Array[Int]] = new Array[Array[Int]](assumedNumberOfNodes)

var numEdges = 0L

var numVertices = 0

var maxVertexId = 0

val tic = System.currentTimeMillis

adjListsIter.foreach {

case (id, nbrArray) =>

if (id >= assumedNumberOfNodes) {

throw new IllegalStateException(

s"Yikes! Entry with id $id, >= assumedNumberOfNodes")

}

nbrs(id) = nbrArray

if (id > maxVertexId) {

maxVertexId = id

}

numEdges += nbrArray.length

numVertices += 1

if (numVertices % 100000 == 0) {

println(s"Done loading $numVertices many vertices. Edges so far: $numEdges")

}

}

(0 until assumedNumberOfNodes).foreach { i =>

if (nbrs(i) == null) {

nbrs(i) = Array[Int]()

}

}

val toc = System.currentTimeMillis()

println(

"maxVertexId is " + maxVertexId + ", assumedNumberOfNodes is " + assumedNumberOfNodes)

println(

s"Done loading graph with $assumedNumberOfNodes nodes and $numEdges edges (counting each edge twice)")

println("Number of nodes with at least neighbor is " + numVertices)

println("Time to load the graph " + (toc - tic) / 1000.0 / 60.0 + " minutes")

val graph = new Graph(assumedNumberOfNodes, numEdges / 2, nbrs, null)

val k = assumedNumberOfNodes / numNodesPerCommunity

println("Will set number of communities to " + k)

val algoConfig = new AlgorithmConfig()

.withCpu(16).withK(k)

.withWtCoeff(10.0).withMaxEpoch(5)

var z = new SparseBinaryMatrix(assumedNumberOfNodes, k)

val err = new PrintWriter(System.err)

println("Going to initalize from random neighborhoods")

z.initFromBestNeighborhoods(

graph,

(gr: Graph, i: Integer) => algoConfig.rng.nextDouble,

false,

err)

println("Done initializing from random neighborhoods")

val prec0 = MHAlgorithm.clusterPrecision(graph, z, 0, 1000, algoConfig.rng)

println("Precision of cluster 0:" + prec0.precision)

val prec1 = MHAlgorithm.clusterPrecision(graph, z, 1, 1000, algoConfig.rng)

println("Precision of cluster 1:" + prec1.precision)

println(

"Fraction of empty rows after initializing from random neighborhoods: " + z.emptyRowProportion)

val tic2 = System.currentTimeMillis

val algo = new MHAlgorithm(algoConfig, graph, z, err)

val optimizedZ = algo.optimize

val toc2 = System.currentTimeMillis

println("Time to optimize: %.2f seconds\n".format((toc2 - tic2) / 1000.0))

println("Time to initialize & optimize: %.2f seconds\n".format((toc2 - toc) / 1000.0))

val srm = MHAlgorithm.heuristicallyScoreClusterAssignments(graph, optimizedZ)

val outputIter = (0 to srm.getNumRows).map { rowId =>

val rowWithIndices = srm.getColIdsForRow(rowId)

val rowWithScores = srm.getValuesForRow(rowId)

val str = rowWithIndices

.zip(rowWithScores).map {

case (colId, score) =>

"%d:%.2g".format(colId + 1, score)

}.mkString(" ")

"%d %s".format(rowId, str)

}

TypedPipe.from(outputIter).writeExecution(TypedTsv(outputDir))

}

}

}

}

/\*\*

\* ./bazel bundle src/scala/com/twitter/simclusters\_v2/scalding:top\_users\_only && \

\* oscar hdfs --hadoop-client-memory 60000 --user cassowary --host atla-aor-08-sr1 \

\* --bundle top\_users\_only --tool com.twitter.simclusters\_v2.scalding.ScalableTopUsersSimilarityGraphApp \

\* --screen --screen-detached --tee ldap\_logs/SubGraphOf100MTopusersWithMappedIds \

\* -- --mappedUsersDir adhoc/ldap\_top100M\_mappedUsers \

\* --inputDir adhoc/ldap\_approximate\_cosine\_similarity\_follow \

\* --outputDir adhoc/ldap\_subgraphOf100MTopUsersWithMappedIds\_correct\_topK

\*/

object ScalableTopUsersSimilarityGraphApp extends TwitterExecutionApp {

implicit val tz: java.util.TimeZone = DateOps.UTC

implicit val dp = DateParser.default

val log = Logger()

def job: Execution[Unit] =

Execution.getConfigMode.flatMap {

case (config, mode) =>

Execution.withId { implicit uniqueId =>

val args = config.getArgs

val inputDir = args("inputDir")

val mappedUsersDir = args("mappedUsersDir")

val maxNeighbors = args.int("maxNeighbors", 100)

val outputDir = args("outputDir")

val mappedUsers = TypedPipe

.from(TypedTsv[(Long, Int, String, Int)](mappedUsersDir))

.map {

case (id, \_, \_, mappedId) =>

(id, mappedId)

}

.shard(200)

val sims = TypedPipe

.from(FollowingsCosineSimilaritiesManhattanSource(path = inputDir))

.map(\_.\_2)

TopUsersSimilarityGraph

.runUsingJoin(

mappedUsers,

sims,

maxNeighbors

).writeExecution(TypedTsv(args("outputDir")))

}

}

}

/\*\*

\* Scalding app using Executions that does the following:

\*

\* 1. Get the top N most followed users on Twitter

\* (also maps them to ids 1 -> N in int space for easier processing)

\* 2. For each user from the step above, get the top K most similar users for this user from the

\* list of N users from the step above.

\* 3. Construct an undirected graph by setting an edge between (u, v) if

\* either v is in u's top-K similar users list, or u is in v's top-K similar user's list.

\* 4. The weight for the (u, v) edge is set to be the cosine similarity between u and v's

\* follower lists, raised to some exponent > 1.

\* This last step is a heuristic reweighting procedure to give more importance to edges involving

\* more similar users.

\* 5. Write the above graph to HDFS in Metis format,

\* i.e. one line per node, with the line for each node specifying the list of neighbors along

\* with their weights. The first line specifies the number of nodes and the number of edges.

\*

\* I've tested this Scalding job for values of topK upto 20M.

\*

\* Example invocation:

\* $ ./bazel bundle src/scala/com/twitter/simclusters\_v2/scalding:top\_users\_similarity\_graph && \

\* oscar hdfs --hadoop-client-memory 60000 --host atla-amw-03-sr1 --bundle top\_users\_similarity\_graph \

\* --tool com.twitter.simclusters\_v2.scalding.TopUsersSimilarityGraphApp \

\* --hadoop-properties "elephantbird.use.combine.input.format=true;elephantbird.combine.split.size=468435456;mapred.min.split.size=468435456;mapreduce.reduce.memory.mb=5096;mapreduce.reduce.java.opts=-Xmx4400m" \

\* --screen --screen-detached --tee logs/20MSubGraphExecution -- --date 2017-10-24 \

\* --minActiveFollowers 300 --topK 20000000 \

\* --inputUserGroupedDir /user/cassowary/manhattan\_sequence\_files/approximate\_cosine\_similarity\_follow/ \

\* --groupedInputInSequenceFiles \

\* --maxNeighborsPerNode 100 --wtExponent 2 \

\* --outputTopUsersDir /user/your\_ldap/simclusters\_graph\_prep\_q42017/top20MUsers \

\* --outputGraphDir /user/your\_ldap/simclusters\_graph\_prep\_q42017/top20Musers\_exp2\_100neighbors\_metis\_graph

\*

\*/

object TopUsersSimilarityGraphApp extends TwitterExecutionApp {

implicit val tz: java.util.TimeZone = DateOps.UTC

implicit val dp = DateParser.default

val log = Logger()

def job: Execution[Unit] =

Execution.getConfigMode.flatMap {

case (config, mode) =>

Execution.withId { implicit uniqueId =>

val args = config.getArgs

val minActiveFollowers = args.int("minActiveFollowers", 100000)

val topK = args.int("topK")

val date = DateRange.parse(args("date"))

val inputSimilarPairsDir = args.optional("inputSimilarPairsDir")

val inputUserGroupedDir = args.optional("inputUserGroupedDir")

val isGroupedInputSequenceFiles = args.boolean("groupedInputInSequenceFiles")

val outputTopUsersDir = args("outputTopUsersDir")

val maxNeighborsPerNode = args.int("maxNeighborsPerNode", 300)

val wtExponent = args.float("wtExponent", 3.5f)

val outputGraphDir = args("outputGraphDir")

val userSource = DAL.readMostRecentSnapshot(UsersourceFlatScalaDataset, date).toTypedPipe

val exception = new IllegalStateException(

"Please specify only one of inputSimilarPairsDir or inputUserGroupedDir"

)

(inputSimilarPairsDir, inputUserGroupedDir) match {

case (Some(\_), Some(\_)) => throw exception

case (None, None) => throw exception

case \_ => // no-op

}

def getSubgraphFn(usersToInclude: Set[Long]) = {

(inputSimilarPairsDir, inputUserGroupedDir) match {

case (Some(similarPairs), None) =>

val similarUserPairs: TypedPipe[SimilarUserPair] =

TypedPipe.from(

new FixedPathLzoScrooge(

inputSimilarPairsDir.get,

SimilarUserPair

))

TopUsersSimilarityGraph.getInMemorySubgraph(

similarUserPairs,

usersToInclude,

maxNeighborsPerNode)

case (None, Some(groupedInput)) =>

val candidatesPipe =

TopUsersSimilarityGraph.readSimsInput(isGroupedInputSequenceFiles, groupedInput)

TopUsersSimilarityGraph.getInMemorySubgraphFromUserGroupedInput(

candidatesPipe,

usersToInclude,

maxNeighborsPerNode

)

case \_ => Execution.from(Nil) // we should never get here

}

}

TopUsersSimilarityGraph

.run(

userSource,

minActiveFollowers,

topK,

getSubgraphFn,

wtExponent

).flatMap {

case (topUsersList, graph) =>

// We're writing to HDFS ourselves, from the submitter node.

// When we use TypedPipe.write, it's failing for large topK, e.g.10M.

// We can make the submitter node have a lot of memory, but it's

// difficult and suboptimal to give this much memory to all mappers.

val topUsersExec = Execution.from(

TopUsersSimilarityGraph

.writeTopUsers(topUsersList, mode, outputTopUsersDir + "/all")

)

// We want to make sure the write of the topUsers succeeds, and

// only then write out the graph. A graph without the topUsers is useless.

topUsersExec.map { \_ =>

// We're writing to HDFS ourselves, from the submitter node.

// When we use TypedPipe.write, it fails due to OOM on the mappers.

// We can make the submitter node have a lot of memory, but it's difficult

// and suboptimal to give this much memory to all mappers.

TopUsersSimilarityGraph.writeToHDFSIfHDFS(

graph

.iterableStringRepresentation(new DecimalFormat("#.###")).iterator().asScala,

mode,

outputGraphDir + "/all"

)

}

}

}

}

}

/\*\*

\* App that only outputs the topK users on Twitter by active follower count. Example invocation:

\* $ ./bazel bundle src/scala/com/twitter/simclusters\_v2/scalding:top\_users\_only && \

\* oscar hdfs --hadoop-client-memory 60000 --host atla-aor-08-sr1 --bundle top\_users\_only \

\* --tool com.twitter.simclusters\_v2.scalding.TopUsersOnlyApp \

\* #are these hadoop-properties needed for this job?

\* #--hadoop-properties "scalding.with.reducers.set.explicitly=true;elephantbird.use.combine.input.format=true;elephantbird.combine.split.size=468435456;mapred.min.split.size=468435456" \

\* --screen --screen-detached --tee logs/10MTopusersOnlyExecution -- --date 2017-10-20 \

\* --minActiveFollowers 500 --topK 10000000 \

\* --outputTopUsersDir /user/your\_ldap/simclusters\_graph\_prep\_q42017/top10MUsers

\*

\* ./bazel bundle src/scala/com/twitter/simclusters\_v2/scalding:top\_users\_only && \

\* oscar hdfs --hadoop-client-memory 60000 --user cassowary --host atla-aor-08-sr1 \

\* --bundle top\_users\_only --tool com.twitter.simclusters\_v2.scalding.TopUsersOnlyApp \

\* --screen --screen-detached --tee ldap\_logs/100MTopusersWithMappedIds \

\* -- --date 2019-10-11 --minActiveFollowers 67 --outputTopUsersDir adhoc/ldap\_top100M\_mappedUsers \

\* --includeMappedIds

\*/

object TopUsersOnlyApp extends TwitterExecutionApp {

implicit val tz: java.util.TimeZone = DateOps.UTC

implicit val dp = DateParser.default

val log = Logger()

def job: Execution[Unit] =

Execution.getConfigMode.flatMap {

case (config, mode) =>

Execution.withId { implicit uniqueId =>

val args = config.getArgs

val minActiveFollowers = args.int("minActiveFollowers", 100000)

val topK = args.int("topK", 20000000)

val date = DateRange.parse(args("date"))

val outputTopUsersDir = args("outputTopUsersDir")

val includeMappedIds = args.boolean("includeMappedIds")

if (includeMappedIds) {

println("Going to include mappedIds in output")

TopUsersSimilarityGraph

.topUsersWithMappedIds(

DAL.readMostRecentSnapshot(UsersourceFlatScalaDataset, date).toTypedPipe,

minActiveFollowers

)

.map {

case TopUserWithMappedId(TopUser(id, activeFollowerCount, screenName), mappedId) =>

(id, activeFollowerCount, screenName, mappedId)

}

.writeExecution(TypedTsv(outputTopUsersDir))

} else {

TopUsersSimilarityGraph

.topUsersInMemory(

DAL.readMostRecentSnapshot(UsersourceFlatScalaDataset, date).toTypedPipe,

minActiveFollowers,

topK

).map { topUsersList =>

TopUsersSimilarityGraph.writeTopUsers(

topUsersList,

mode,

outputTopUsersDir + "/all")

}

}

}

}

}