package com.twitter.simclusters\_v2.scio.multi\_type\_graph.assemble\_multi\_type\_graph

import com.spotify.scio.ScioContext

import com.spotify.scio.coders.Coder

import com.spotify.scio.values.SCollection

import com.twitter.beam.io.dal.DAL

import com.twitter.beam.io.fs.multiformat.DiskFormat

import com.twitter.beam.io.fs.multiformat.PathLayout

import com.twitter.beam.job.DateRangeOptions

import com.twitter.dal.client.dataset.KeyValDALDataset

import com.twitter.dal.client.dataset.SnapshotDALDataset

import com.twitter.frigate.data\_pipeline.magicrecs.magicrecs\_notifications\_lite.thriftscala.MagicRecsNotificationLite

import com.twitter.iesource.thriftscala.InteractionEvent

import com.twitter.iesource.thriftscala.InteractionType

import com.twitter.iesource.thriftscala.ReferenceTweet

import com.twitter.scalding\_internal.multiformat.format.keyval.KeyVal

import com.twitter.scio\_internal.coders.ThriftStructLazyBinaryScroogeCoder

import com.twitter.scio\_internal.job.ScioBeamJob

import com.twitter.scrooge.ThriftStruct

import com.twitter.simclusters\_v2.common.Country

import com.twitter.simclusters\_v2.common.Language

import com.twitter.simclusters\_v2.common.TopicId

import com.twitter.simclusters\_v2.common.TweetId

import com.twitter.simclusters\_v2.common.UserId

import com.twitter.simclusters\_v2.hdfs\_sources.MultiTypeGraphForTopKRightNodesThriftScioScalaDataset

import com.twitter.simclusters\_v2.hdfs\_sources.TopKRightNounsScioScalaDataset

import com.twitter.simclusters\_v2.hdfs\_sources.TruncatedMultiTypeGraphScioScalaDataset

import com.twitter.simclusters\_v2.scio.common.ExternalDataSources

import com.twitter.simclusters\_v2.scio.multi\_type\_graph.assemble\_multi\_type\_graph.Config.GlobalDefaultMinFrequencyOfRightNodeType

import com.twitter.simclusters\_v2.scio.multi\_type\_graph.assemble\_multi\_type\_graph.Config.HalfLifeInDaysForFavScore

import com.twitter.simclusters\_v2.scio.multi\_type\_graph.assemble\_multi\_type\_graph.Config.NumTopNounsForUnknownRightNodeType

import com.twitter.simclusters\_v2.scio.multi\_type\_graph.assemble\_multi\_type\_graph.Config.SampledEmployeeIds

import com.twitter.simclusters\_v2.scio.multi\_type\_graph.assemble\_multi\_type\_graph.Config.TopKConfig

import com.twitter.simclusters\_v2.scio.multi\_type\_graph.assemble\_multi\_type\_graph.Config.TopKRightNounsForMHDump

import com.twitter.simclusters\_v2.scio.multi\_type\_graph.common.MultiTypeGraphUtil

import com.twitter.simclusters\_v2.thriftscala.EdgeWithDecayedWeights

import com.twitter.simclusters\_v2.thriftscala.LeftNode

import com.twitter.simclusters\_v2.thriftscala.MultiTypeGraphEdge

import com.twitter.simclusters\_v2.thriftscala.Noun

import com.twitter.simclusters\_v2.thriftscala.NounWithFrequency

import com.twitter.simclusters\_v2.thriftscala.NounWithFrequencyList

import com.twitter.simclusters\_v2.thriftscala.RightNode

import com.twitter.simclusters\_v2.thriftscala.RightNodeType

import com.twitter.simclusters\_v2.thriftscala.RightNodeTypeStruct

import com.twitter.simclusters\_v2.thriftscala.RightNodeWithEdgeWeight

import com.twitter.simclusters\_v2.thriftscala.RightNodeWithEdgeWeightList

import com.twitter.twadoop.user.gen.thriftscala.CombinedUser

import com.twitter.util.Duration

import java.time.Instant

import org.joda.time.Interval

/\*\*

\* Scio version of

\* src/scala/com/twitter/simclusters\_v2/scalding/multi\_type\_graph/assemble\_multi\_type\_graph/AssembleMultiTypeGraph.scala

\*/

trait AssembleMultiTypeGraphScioBaseApp extends ScioBeamJob[DateRangeOptions] {

// Provides an implicit binary thrift scrooge coder by default.

override implicit def scroogeCoder[T <: ThriftStruct: Manifest]: Coder[T] =

ThriftStructLazyBinaryScroogeCoder.scroogeCoder

val isAdhoc: Boolean

val rootMHPath: String

val rootThriftPath: String

val truncatedMultiTypeGraphMHOutputDir: String =

Config.truncatedMultiTypeGraphMHOutputDir

val truncatedMultiTypeGraphThriftOutputDir: String =

Config.truncatedMultiTypeGraphThriftOutputDir

val topKRightNounsMHOutputDir: String = Config.topKRightNounsMHOutputDir

val topKRightNounsOutputDir: String = Config.topKRightNounsOutputDir

val fullMultiTypeGraphThriftOutputDir: String =

Config.fullMultiTypeGraphThriftOutputDir

val truncatedMultiTypeGraphKeyValDataset: KeyValDALDataset[

KeyVal[LeftNode, RightNodeWithEdgeWeightList]

] = TruncatedMultiTypeGraphScioScalaDataset

val topKRightNounsKeyValDataset: KeyValDALDataset[

KeyVal[RightNodeTypeStruct, NounWithFrequencyList]

] = TopKRightNounsScioScalaDataset

val topKRightNounsMHKeyValDataset: KeyValDALDataset[

KeyVal[RightNodeTypeStruct, NounWithFrequencyList]

] = TopKRightNounsMhScioScalaDataset

val fullMultiTypeGraphSnapshotDataset: SnapshotDALDataset[MultiTypeGraphEdge] =

FullMultiTypeGraphScioScalaDataset

val multiTypeGraphTopKForRightNodesSnapshotDataset: SnapshotDALDataset[

MultiTypeGraphEdge

] =

MultiTypeGraphForTopKRightNodesThriftScioScalaDataset

def getValidUsers(

input: SCollection[CombinedUser]

): SCollection[UserId] = {

input

.flatMap { u =>

for {

user <- u.user

if user.id != 0

safety <- user.safety

if !(safety.suspended || safety.deactivated)

} yield {

user.id

}

}

}

def filterInvalidUsers(

flockEdges: SCollection[(UserId, UserId)],

validUsers: SCollection[UserId]

): SCollection[(UserId, UserId)] = {

val validUsersWithValues = validUsers.map(userId => (userId, ()))

flockEdges

.join(validUsersWithValues)

.map {

case (srcId, (destId, \_)) =>

(destId, srcId)

}

.join(validUsersWithValues)

.map {

case (destId, (srcId, \_)) =>

(srcId, destId)

}

}

def getFavEdges(

input: SCollection[EdgeWithDecayedWeights],

halfLifeInDaysForFavScore: Int,

): SCollection[(Long, Long, Double)] = {

input

.flatMap { edge =>

if (edge.weights.halfLifeInDaysToDecayedSums.contains(halfLifeInDaysForFavScore)) {

Some(

(

edge.sourceId,

edge.destinationId,

edge.weights.halfLifeInDaysToDecayedSums(halfLifeInDaysForFavScore)))

} else {

None

}

}

}

def leftRightTuple(

leftNodeUserId: UserId,

rightNodeType: RightNodeType,

rightNoun: Noun,

weight: Double = 1.0

): (LeftNode, RightNodeWithEdgeWeight) = {

(

LeftNode.UserId(leftNodeUserId),

RightNodeWithEdgeWeight(

rightNode = RightNode(rightNodeType = rightNodeType, noun = rightNoun),

weight = weight))

}

def getUserFavGraph(

userUserFavEdges: SCollection[(UserId, UserId, Double)]

): SCollection[(LeftNode, RightNodeWithEdgeWeight)] = {

userUserFavEdges.map {

case (srcId, destId, edgeWt) =>

leftRightTuple(srcId, RightNodeType.FavUser, Noun.UserId(destId), edgeWt)

}

}

def getUserFollowGraph(

userUserFollowEdges: SCollection[(UserId, UserId)]

): SCollection[(LeftNode, RightNodeWithEdgeWeight)] = {

userUserFollowEdges.map {

case (srcId, destId) =>

leftRightTuple(srcId, RightNodeType.FollowUser, Noun.UserId(destId), 1.0)

}

}

def getUserBlockGraph(

userUserBlockEdges: SCollection[(UserId, UserId)]

): SCollection[(LeftNode, RightNodeWithEdgeWeight)] = {

userUserBlockEdges.map {

case (srcId, destId) =>

leftRightTuple(srcId, RightNodeType.BlockUser, Noun.UserId(destId), 1.0)

}

}

def getUserAbuseReportGraph(

userUserAbuseReportEdges: SCollection[(UserId, UserId)]

): SCollection[(LeftNode, RightNodeWithEdgeWeight)] = {

userUserAbuseReportEdges.map {

case (srcId, destId) =>

leftRightTuple(srcId, RightNodeType.AbuseReportUser, Noun.UserId(destId), 1.0)

}

}

def getUserSpamReportGraph(

userUserSpamReportEdges: SCollection[(UserId, UserId)]

): SCollection[(LeftNode, RightNodeWithEdgeWeight)] = {

userUserSpamReportEdges.map {

case (srcId, destId) =>

leftRightTuple(srcId, RightNodeType.SpamReportUser, Noun.UserId(destId), 1.0)

}

}

def getUserTopicFollowGraph(

topicUserFollowedByEdges: SCollection[(TopicId, UserId)]

): SCollection[(LeftNode, RightNodeWithEdgeWeight)] = {

topicUserFollowedByEdges.map {

case (topicId, userId) =>

leftRightTuple(userId, RightNodeType.FollowTopic, Noun.TopicId(topicId), 1.0)

}

}

def getUserSignUpCountryGraph(

userSignUpCountryEdges: SCollection[(UserId, Country)]

): SCollection[(LeftNode, RightNodeWithEdgeWeight)] = {

userSignUpCountryEdges.map {

case (userId, country) =>

leftRightTuple(userId, RightNodeType.SignUpCountry, Noun.Country(country), 1.0)

}

}

def getMagicRecsNotifOpenOrClickTweetsGraph(

userMRNotifOpenOrClickEvents: SCollection[MagicRecsNotificationLite]

): SCollection[(LeftNode, RightNodeWithEdgeWeight)] = {

userMRNotifOpenOrClickEvents.flatMap { entry =>

for {

userId <- entry.targetUserId

tweetId <- entry.tweetId

} yield {

leftRightTuple(userId, RightNodeType.NotifOpenOrClickTweet, Noun.TweetId(tweetId), 1.0)

}

}

}

def getUserConsumedLanguagesGraph(

userConsumedLanguageEdges: SCollection[(UserId, Seq[(Language, Double)])]

): SCollection[(LeftNode, RightNodeWithEdgeWeight)] = {

userConsumedLanguageEdges.flatMap {

case (userId, langWithWeights) =>

langWithWeights.map {

case (lang, weight) =>

leftRightTuple(userId, RightNodeType.ConsumedLanguage, Noun.Language(lang), 1.0)

}

}

}

def getSearchGraph(

userSearchQueryEdges: SCollection[(UserId, String)]

): SCollection[(LeftNode, RightNodeWithEdgeWeight)] = {

userSearchQueryEdges.map {

case (userId, query) =>

leftRightTuple(userId, RightNodeType.SearchQuery, Noun.Query(query), 1.0)

}

}

def getUserTweetInteractionGraph(

tweetInteractionEvents: SCollection[InteractionEvent],

): SCollection[(LeftNode, RightNodeWithEdgeWeight)] = {

val userTweetInteractionsByType: SCollection[((UserId, TweetId), RightNodeType)] =

tweetInteractionEvents

.flatMap { event =>

val referenceTweet: Option[ReferenceTweet] = event.referenceTweet

val targetId: Long = event.targetId

val userId: Long = event.engagingUserId

// To find the id of the tweet that was interacted with

// For likes, this is the targetId; for retweet or reply, it is the referenceTweet's id

// One thing to note is that for likes, referenceTweet is empty

val (tweetIdOpt, rightNodeTypeOpt) = {

event.interactionType match {

case Some(InteractionType.Favorite) =>

// Only allow favorites on original tweets, not retweets, to avoid double-counting

// because we have retweet-type tweets in the data source as well

(

if (referenceTweet.isEmpty) {

Some(targetId)

} else None,

Some(RightNodeType.FavTweet))

case Some(InteractionType.Reply) =>

(referenceTweet.map(\_.tweetId), Some(RightNodeType.ReplyTweet))

case Some(InteractionType.Retweet) =>

(referenceTweet.map(\_.tweetId), Some(RightNodeType.RetweetTweet))

case \_ => (None, None)

}

}

for {

tweetId <- tweetIdOpt

rightNodeType <- rightNodeTypeOpt

} yield {

((userId, tweetId), rightNodeType)

}

}

userTweetInteractionsByType

.mapValues(Set(\_))

.sumByKey

.flatMap {

case ((userId, tweetId), rightNodeTypeSet) =>

rightNodeTypeSet.map { rightNodeType =>

leftRightTuple(userId, rightNodeType, Noun.TweetId(tweetId), 1.0)

}

}

}

def getTopKRightNounsWithFrequencies(

fullGraph: SCollection[(LeftNode, RightNodeWithEdgeWeight)],

topKConfig: Map[RightNodeType, Int],

minFrequency: Int,

): SCollection[(RightNodeType, Seq[(Noun, Double)])] = {

val maxAcrossRightNounType: Int = topKConfig.valuesIterator.max

fullGraph

.map {

case (leftNode, rightNodeWithWeight) =>

(rightNodeWithWeight.rightNode, 1.0)

}

.sumByKey

.filter(\_.\_2 >= minFrequency)

.map {

case (rightNode, freq) =>

(rightNode.rightNodeType, (rightNode.noun, freq))

}

.topByKey(maxAcrossRightNounType)(Ordering.by(\_.\_2))

.map {

case (rightNodeType, nounsListWithFreq) =>

val truncatedList = nounsListWithFreq.toSeq

.sortBy(-\_.\_2)

.take(topKConfig.getOrElse(rightNodeType, NumTopNounsForUnknownRightNodeType))

(rightNodeType, truncatedList)

}

}

def getTruncatedGraph(

fullGraph: SCollection[(LeftNode, RightNodeWithEdgeWeight)],

topKWithFrequency: SCollection[(RightNodeType, Seq[(Noun, Double)])]

): SCollection[(LeftNode, RightNodeWithEdgeWeight)] = {

val topNouns = topKWithFrequency

.flatMap {

case (rightNodeType, nounsList) =>

nounsList

.map {

case (nounVal, aggregatedFrequency) =>

RightNode(rightNodeType, nounVal)

}

}.map(nouns => (nouns, ()))

fullGraph

.map {

case (leftNode, rightNodeWithWeight) =>

(rightNodeWithWeight.rightNode, (leftNode, rightNodeWithWeight))

}

.hashJoin(topNouns)

.map {

case (rightNode, ((left, rightNodeWithWeight), \_)) =>

(left, rightNodeWithWeight)

}

}

def buildEmployeeGraph(

graph: SCollection[(LeftNode, RightNodeWithEdgeWeight)]

): SCollection[(LeftNode, RightNodeWithEdgeWeight)] = {

val employeeIds = SampledEmployeeIds

graph

.collect {

case (LeftNode.UserId(userId), rightNodeWithWeight) if employeeIds.contains(userId) =>

(LeftNode.UserId(userId), rightNodeWithWeight)

}

}

override def configurePipeline(sc: ScioContext, opts: DateRangeOptions): Unit = {

// Define the implicit ScioContext to read datasets from ExternalDataSources

implicit def scioContext: ScioContext = sc

// DAL.Environment variable for WriteExecs

val dalEnv = if (isAdhoc) DAL.Environment.Dev else DAL.Environment.Prod

// Define date intervals

val interval\_7days =

new Interval(opts.interval.getEnd.minusWeeks(1), opts.interval.getEnd.minusMillis(1))

val interval\_14days =

new Interval(opts.interval.getEnd.minusWeeks(2), opts.interval.getEnd.minusMillis(1))

/\*

\* Dataset read operations

\*/

// Get list of valid UserIds - to filter out deactivated or suspended user accounts

val validUsers = getValidUsers(ExternalDataSources.userSource(Duration.fromDays(7)))

// ieSource tweet engagements data for tweet favs, replies, retweets - from last 14 days

val tweetSource = ExternalDataSources.ieSourceTweetEngagementsSource(interval\_14days)

// Read TFlock datasets

val flockFollowSource = ExternalDataSources.flockFollowSource(Duration.fromDays(7))

val flockBlockSource = ExternalDataSources.flockBlockSource(Duration.fromDays(7))

val flockReportAsAbuseSource =

ExternalDataSources.flockReportAsAbuseSource(Duration.fromDays(7))

val flockReportAsSpamSource =

ExternalDataSources.flockReportAsSpamSource(Duration.fromDays(7))

// user-user fav edges

val userUserFavSource = ExternalDataSources.userUserFavSource(Duration.fromDays(14))

val userUserFavEdges = getFavEdges(userUserFavSource, HalfLifeInDaysForFavScore)

// user-user follow edges

val userUserFollowEdges = filterInvalidUsers(flockFollowSource, validUsers)

// user-user block edges

val userUserBlockEdges = filterInvalidUsers(flockBlockSource, validUsers)

// user-user abuse report edges

val userUserAbuseReportEdges = filterInvalidUsers(flockReportAsAbuseSource, validUsers)

// user-user spam report edges

val userUserSpamReportEdges = filterInvalidUsers(flockReportAsSpamSource, validUsers)

// user-signup country edges

val userSignUpCountryEdges = ExternalDataSources

.userCountrySource(Duration.fromDays(7))

// user-consumed language edges

val userConsumedLanguageEdges =

ExternalDataSources.inferredUserConsumedLanguageSource(Duration.fromDays(7))

// user-topic follow edges

val topicUserFollowedByEdges =

ExternalDataSources.topicFollowGraphSource(Duration.fromDays(7))

// user-MRNotifOpenOrClick events from last 7 days

val userMRNotifOpenOrClickEvents =

ExternalDataSources.magicRecsNotficationOpenOrClickEventsSource(interval\_7days)

// user-searchQuery strings from last 7 days

val userSearchQueryEdges =

ExternalDataSources.adaptiveSearchScribeLogsSource(interval\_7days)

/\*

\* Generate the full graph

\*/

val fullGraph =

getUserTweetInteractionGraph(tweetSource) ++

getUserFavGraph(userUserFavEdges) ++

getUserFollowGraph(userUserFollowEdges) ++

getUserBlockGraph(userUserBlockEdges) ++

getUserAbuseReportGraph(userUserAbuseReportEdges) ++

getUserSpamReportGraph(userUserSpamReportEdges) ++

getUserSignUpCountryGraph(userSignUpCountryEdges) ++

getUserConsumedLanguagesGraph(userConsumedLanguageEdges) ++

getUserTopicFollowGraph(topicUserFollowedByEdges) ++

getMagicRecsNotifOpenOrClickTweetsGraph(userMRNotifOpenOrClickEvents) ++

getSearchGraph(userSearchQueryEdges)

// Get Top K RightNodes

val topKRightNodes: SCollection[(RightNodeType, Seq[(Noun, Double)])] =

getTopKRightNounsWithFrequencies(

fullGraph,

TopKConfig,

GlobalDefaultMinFrequencyOfRightNodeType)

// key transformation - topK nouns, keyed by the RightNodeNounType

val topKNounsKeyedByType: SCollection[(RightNodeTypeStruct, NounWithFrequencyList)] =

topKRightNodes

.map {

case (rightNodeType, rightNounsWithScoresList) =>

val nounsListWithFrequency: Seq[NounWithFrequency] = rightNounsWithScoresList

.map {

case (noun, aggregatedFrequency) =>

NounWithFrequency(noun, aggregatedFrequency)

}

(RightNodeTypeStruct(rightNodeType), NounWithFrequencyList(nounsListWithFrequency))

}

// Get Truncated graph based on the top K RightNodes

val truncatedGraph: SCollection[(LeftNode, RightNodeWithEdgeWeight)] =

getTruncatedGraph(fullGraph, topKRightNodes)

// key transformations - truncated graph, keyed by LeftNode

// Note: By wrapping and unwrapping with the LeftNode.UserId, we don't have to deal

// with defining our own customer ordering for LeftNode type

val truncatedGraphKeyedBySrc: SCollection[(LeftNode, RightNodeWithEdgeWeightList)] =

truncatedGraph

.collect {

case (LeftNode.UserId(userId), rightNodeWithWeight) =>

userId -> List(rightNodeWithWeight)

}

.sumByKey

.map {

case (userId, rightNodeWithWeightList) =>

(LeftNode.UserId(userId), RightNodeWithEdgeWeightList(rightNodeWithWeightList))

}

// WriteExecs

// Write TopK RightNodes to DAL - save all the top K nodes for the clustering step

topKNounsKeyedByType

.map {

case (engagementType, rightList) =>

KeyVal(engagementType, rightList)

}

.saveAsCustomOutput(

name = "WriteTopKNouns",

DAL.writeVersionedKeyVal(

topKRightNounsKeyValDataset,

PathLayout.VersionedPath(prefix =

rootMHPath + topKRightNounsOutputDir),

instant = Instant.ofEpochMilli(opts.interval.getEndMillis - 1L),

environmentOverride = dalEnv,

)

)

// Write TopK RightNodes to DAL - only take TopKRightNounsForMHDump RightNodes for MH dump

topKNounsKeyedByType

.map {

case (engagementType, rightList) =>

val rightListMH =

NounWithFrequencyList(rightList.nounWithFrequencyList.take(TopKRightNounsForMHDump))

KeyVal(engagementType, rightListMH)

}

.saveAsCustomOutput(

name = "WriteTopKNounsToMHForDebugger",

DAL.writeVersionedKeyVal(

topKRightNounsMHKeyValDataset,

PathLayout.VersionedPath(prefix =

rootMHPath + topKRightNounsMHOutputDir),

instant = Instant.ofEpochMilli(opts.interval.getEndMillis - 1L),

environmentOverride = dalEnv,

)

)

// Write truncated graph (MultiTypeGraphTopKForRightNodes) to DAL in KeyVal format

truncatedGraphKeyedBySrc

.map {

case (leftNode, rightNodeWithWeightList) =>

KeyVal(leftNode, rightNodeWithWeightList)

}.saveAsCustomOutput(

name = "WriteTruncatedMultiTypeGraph",

DAL.writeVersionedKeyVal(

truncatedMultiTypeGraphKeyValDataset,

PathLayout.VersionedPath(prefix =

rootMHPath + truncatedMultiTypeGraphMHOutputDir),

instant = Instant.ofEpochMilli(opts.interval.getEndMillis - 1L),

environmentOverride = dalEnv,

)

)

// Write truncated graph (MultiTypeGraphTopKForRightNodes) to DAL in thrift format

truncatedGraph

.map {

case (leftNode, rightNodeWithWeight) =>

MultiTypeGraphEdge(leftNode, rightNodeWithWeight)

}.saveAsCustomOutput(

name = "WriteTruncatedMultiTypeGraphThrift",

DAL.writeSnapshot(

multiTypeGraphTopKForRightNodesSnapshotDataset,

PathLayout.FixedPath(rootThriftPath + truncatedMultiTypeGraphThriftOutputDir),

Instant.ofEpochMilli(opts.interval.getEndMillis - 1L),

DiskFormat.Thrift(),

environmentOverride = dalEnv

)

)

// Write full graph to DAL

fullGraph

.map {

case (leftNode, rightNodeWithWeight) =>

MultiTypeGraphEdge(leftNode, rightNodeWithWeight)

}

.saveAsCustomOutput(

name = "WriteFullMultiTypeGraph",

DAL.writeSnapshot(

fullMultiTypeGraphSnapshotDataset,

PathLayout.FixedPath(rootThriftPath + fullMultiTypeGraphThriftOutputDir),

Instant.ofEpochMilli(opts.interval.getEndMillis - 1L),

DiskFormat.Thrift(),

environmentOverride = dalEnv

)

)

}

}