package com.twitter.simclusters\_v2.scalding

package multi\_type\_graph.assemble\_multi\_type\_graph

import com.twitter.bijection.scrooge.BinaryScalaCodec

import com.twitter.scalding\_internal.job.RequiredBinaryComparators.ordSer

import com.twitter.scalding.typed.TypedPipe

import com.twitter.scalding.{DateRange, Days, Stat, UniqueID}

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

import com.twitter.simclusters\_v2.scalding.embedding.common.ExternalDataSources

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

LeftNode,

Noun,

RightNode,

RightNodeType,

RightNodeWithEdgeWeight

}

import java.util.TimeZone

import com.twitter.iesource.thriftscala.{InteractionEvent, InteractionType, ReferenceTweet}

import com.twitter.simclusters\_v2.common.{Country, Language, TopicId, TweetId, UserId}

import com.twitter.usersource.snapshot.combined.UsersourceScalaDataset

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

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

object AssembleMultiTypeGraph {

import Config.\_

implicit val nounOrdering: Ordering[Noun] = new Ordering[Noun] {

// We define an ordering for each noun type as specified in simclusters\_v2/multi\_type\_graph.thrift

// Please make sure we don't remove anything here that's still a part of the union Noun thrift and

// vice versa, if we add a new noun type to thrift, an ordering for it needs to added here as well.

def nounTypeOrder(noun: Noun): Int = noun match {

case \_: Noun.UserId => 0

case \_: Noun.Country => 1

case \_: Noun.Language => 2

case \_: Noun.Query => 3

case \_: Noun.TopicId => 4

case \_: Noun.TweetId => 5

}

override def compare(x: Noun, y: Noun): Int = (x, y) match {

case (Noun.UserId(a), Noun.UserId(b)) => a compare b

case (Noun.Country(a), Noun.Country(b)) => a compare b

case (Noun.Language(a), Noun.Language(b)) => a compare b

case (Noun.Query(a), Noun.Query(b)) => a compare b

case (Noun.TopicId(a), Noun.TopicId(b)) => a compare b

case (Noun.TweetId(a), Noun.TweetId(b)) => a compare b

case (nounA, nounB) => nounTypeOrder(nounA) compare nounTypeOrder(nounB)

}

}

implicit val rightNodeTypeOrdering: Ordering[RightNodeType] = ordSer[RightNodeType]

implicit val rightNodeTypeWithNounOrdering: Ordering[RightNode] =

new Ordering[RightNode] {

override def compare(x: RightNode, y: RightNode): Int = {

Ordering

.Tuple2(rightNodeTypeOrdering, nounOrdering)

.compare((x.rightNodeType, x.noun), (y.rightNodeType, y.noun))

}

}

def getUserTweetInteractionGraph(

tweetInteractionEvents: TypedPipe[InteractionEvent],

)(

implicit uniqueID: UniqueID

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

val numUserTweetInteractionEntries = Stat("num\_user\_tweet\_interaction\_entries")

val numDistinctUserTweetInteractionEntries = Stat("num\_distinct\_user\_tweet\_interaction\_entries")

val numFavedTweets = Stat("num\_faved\_tweets")

val numRepliedTweets = Stat("num\_replied\_tweets")

val numRetweetedTweets = Stat("num\_retweeted\_tweets")

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

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) {

numFavedTweets.inc()

Some(targetId)

} else None,

Some(RightNodeType.FavTweet))

case Some(InteractionType.Reply) =>

numRepliedTweets.inc()

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

case Some(InteractionType.Retweet) =>

numRetweetedTweets.inc()

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

case \_ => (None, None)

}

}

for {

tweetId <- tweetIdOpt

rightNodeType <- rightNodeTypeOpt

} yield {

numUserTweetInteractionEntries.inc()

((userId, rightNodeType), tweetId)

}

}

userTweetInteractionsByType

.mapValues(Set(\_))

.sumByKey

.flatMap {

case ((userId, rightNodeType), tweetIdSet) =>

tweetIdSet.map { tweetId =>

numDistinctUserTweetInteractionEntries.inc()

(

LeftNode.UserId(userId),

RightNodeWithEdgeWeight(

rightNode = RightNode(rightNodeType = rightNodeType, noun = Noun.TweetId(tweetId)),

weight = 1.0))

}

}

}

def getUserFavGraph(

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

)(

implicit uniqueID: UniqueID

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

val numInputFavEdges = Stat("num\_input\_fav\_edges")

userUserFavEdges.map {

case (srcId, destId, edgeWt) =>

numInputFavEdges.inc()

(

LeftNode.UserId(srcId),

RightNodeWithEdgeWeight(

rightNode =

RightNode(rightNodeType = RightNodeType.FavUser, noun = Noun.UserId(destId)),

weight = edgeWt))

}

}

def getUserFollowGraph(

userUserFollowEdges: TypedPipe[(UserId, UserId)]

)(

implicit uniqueID: UniqueID

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

val numFlockFollowEdges = Stat("num\_flock\_follow\_edges")

userUserFollowEdges.map {

case (srcId, destId) =>

numFlockFollowEdges.inc()

(

LeftNode.UserId(srcId),

RightNodeWithEdgeWeight(

rightNode =

RightNode(rightNodeType = RightNodeType.FollowUser, noun = Noun.UserId(destId)),

weight = 1.0))

}

}

def getUserBlockGraph(

userUserBlockEdges: TypedPipe[(UserId, UserId)]

)(

implicit uniqueID: UniqueID

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

val numFlockBlockEdges = Stat("num\_flock\_block\_edges")

userUserBlockEdges.map {

case (srcId, destId) =>

numFlockBlockEdges.inc()

(

LeftNode.UserId(srcId),

RightNodeWithEdgeWeight(

rightNode =

RightNode(rightNodeType = RightNodeType.BlockUser, noun = Noun.UserId(destId)),

weight = 1.0))

}

}

def getUserAbuseReportGraph(

userUserAbuseReportEdges: TypedPipe[(UserId, UserId)]

)(

implicit uniqueID: UniqueID

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

val numFlockAbuseEdges = Stat("num\_flock\_abuse\_edges")

userUserAbuseReportEdges.map {

case (srcId, destId) =>

numFlockAbuseEdges.inc()

(

LeftNode.UserId(srcId),

RightNodeWithEdgeWeight(

rightNode =

RightNode(rightNodeType = RightNodeType.AbuseReportUser, noun = Noun.UserId(destId)),

weight = 1.0))

}

}

def filterInvalidUsers(

flockEdges: TypedPipe[(UserId, UserId)],

validUsers: TypedPipe[UserId]

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

flockEdges

.join(validUsers.asKeys)

// .withReducers(10000)

.map {

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

(destId, srcId)

}

.join(validUsers.asKeys)

// .withReducers(10000)

.map {

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

(srcId, destId)

}

}

def getUserSpamReportGraph(

userUserSpamReportEdges: TypedPipe[(UserId, UserId)]

)(

implicit uniqueID: UniqueID

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

val numFlockSpamEdges = Stat("num\_flock\_spam\_edges")

userUserSpamReportEdges.map {

case (srcId, destId) =>

numFlockSpamEdges.inc()

(

LeftNode.UserId(srcId),

RightNodeWithEdgeWeight(

rightNode =

RightNode(rightNodeType = RightNodeType.SpamReportUser, noun = Noun.UserId(destId)),

weight = 1.0))

}

}

def getUserTopicFollowGraph(

topicUserFollowedByEdges: TypedPipe[(TopicId, UserId)]

)(

implicit uniqueID: UniqueID

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

val numTFGEdges = Stat("num\_tfg\_edges")

topicUserFollowedByEdges.map {

case (topicId, userId) =>

numTFGEdges.inc()

(

LeftNode.UserId(userId),

RightNodeWithEdgeWeight(

rightNode =

RightNode(rightNodeType = RightNodeType.FollowTopic, noun = Noun.TopicId(topicId)),

weight = 1.0)

)

}

}

def getUserSignUpCountryGraph(

userSignUpCountryEdges: TypedPipe[(UserId, (Country, Language))]

)(

implicit uniqueID: UniqueID

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

val numUserSourceEntriesRead = Stat("num\_user\_source\_entries")

userSignUpCountryEdges.map {

case (userId, (country, lang)) =>

numUserSourceEntriesRead.inc()

(

LeftNode.UserId(userId),

RightNodeWithEdgeWeight(

rightNode =

RightNode(rightNodeType = RightNodeType.SignUpCountry, noun = Noun.Country(country)),

weight = 1.0))

}

}

def getMagicRecsNotifOpenOrClickTweetsGraph(

userMRNotifOpenOrClickEvents: TypedPipe[MagicRecsNotificationLite]

)(

implicit uniqueID: UniqueID

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

val numNotifOpenOrClickEntries = Stat("num\_notif\_open\_or\_click")

userMRNotifOpenOrClickEvents.flatMap { entry =>

numNotifOpenOrClickEntries.inc()

for {

userId <- entry.targetUserId

tweetId <- entry.tweetId

} yield {

(

LeftNode.UserId(userId),

RightNodeWithEdgeWeight(

rightNode = RightNode(

rightNodeType = RightNodeType.NotifOpenOrClickTweet,

noun = Noun.TweetId(tweetId)),

weight = 1.0))

}

}

}

def getUserConsumedLanguagesGraph(

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

)(

implicit uniqueID: UniqueID

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

val numPenguinSourceEntriesRead = Stat("num\_penguin\_source\_entries")

userConsumedLanguageEdges.flatMap {

case (userId, langWithWeights) =>

numPenguinSourceEntriesRead.inc()

langWithWeights.map {

case (lang, weight) =>

(

LeftNode.UserId(userId),

RightNodeWithEdgeWeight(

rightNode = RightNode(

rightNodeType = RightNodeType.ConsumedLanguage,

noun = Noun.Language(lang)),

weight = weight))

}

}

}

def getSearchGraph(

userSearchQueryEdges: TypedPipe[(UserId, String)]

)(

implicit uniqueID: UniqueID

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

val numSearchQueries = Stat("num\_search\_queries")

userSearchQueryEdges.map {

case (userId, query) =>

numSearchQueries.inc()

(

LeftNode.UserId(userId),

RightNodeWithEdgeWeight(

rightNode =

RightNode(rightNodeType = RightNodeType.SearchQuery, noun = Noun.Query(query)),

weight = 1.0))

}

}

def buildEmployeeGraph(

fullGraph: TypedPipe[(LeftNode, RightNodeWithEdgeWeight)]

)(

implicit uniqueID: UniqueID

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

val numEmployeeEdges = Stat("num\_employee\_edges")

val employeeIds = Config.SampledEmployeeIds

fullGraph

.collect {

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

numEmployeeEdges.inc()

(LeftNode.UserId(userId), rightNodeWithWeight)

}

}

def getTruncatedGraph(

fullGraph: TypedPipe[(LeftNode, RightNodeWithEdgeWeight)],

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

)(

implicit uniqueID: UniqueID

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

val numEntriesTruncatedGraph = Stat("num\_entries\_truncated\_graph")

val numTopKTruncatedNouns = Stat("num\_topk\_truncated\_nouns")

implicit val rightNodeSer: RightNode => Array[Byte] = BinaryScalaCodec(RightNode)

val topNouns: TypedPipe[RightNode] = topKWithFrequency

.flatMap {

case (rightNodeType, nounsList) =>

nounsList

.map {

case (nounVal, aggregatedFrequency) =>

numTopKTruncatedNouns.inc()

RightNode(rightNodeType, nounVal)

}

}

fullGraph

.map {

case (leftNode, rightNodeWithWeight) =>

(rightNodeWithWeight.rightNode, (leftNode, rightNodeWithWeight))

}

.sketch(reducers = 5000)

.join(topNouns.asKeys.toTypedPipe)

.map {

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

numEntriesTruncatedGraph.inc()

(left, rightNodeWithWeight)

}

}

def getTopKRightNounsWithFrequencies(

fullGraph: TypedPipe[(LeftNode, RightNodeWithEdgeWeight)],

topKConfig: Map[RightNodeType, Int],

minFrequency: Int

)(

implicit uniqueID: UniqueID

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

val maxAcrossRightNounType: Int = topKConfig.valuesIterator.max

fullGraph

.map {

case (leftNode, rightNodeWithWeight) =>

(rightNodeWithWeight.rightNode, 1.0)

}

.sumByKey

// .withReducers(20000)

.toTypedPipe

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

.map {

case (rightNode, freq) =>

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

}

.group(rightNodeTypeOrdering)

// Note: if maxAcrossRightNounType is >15M, it might result in OOM on reducer

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

// An alternative to using group followed by sortedReverseTake is to define TopKMonoids,

// one for each RightNodeType to get the most frequent rightNouns

.map {

case (rightNodeType, nounsListWithFreq) =>

val truncatedList = nounsListWithFreq

.sortBy(-\_.\_2)

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

(rightNodeType, truncatedList)

}

}

def getValidUsers(

userSource: TypedPipe[CombinedUser]

)(

implicit uniqueID: UniqueID

): TypedPipe[UserId] = {

val numValidUsers = Stat("num\_valid\_users")

userSource

.flatMap { u =>

for {

user <- u.user

if user.id != 0

safety <- user.safety

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

} yield {

numValidUsers.inc()

user.id

}

}

}

def getFullGraph(

)(

implicit dateRange: DateRange,

timeZone: TimeZone,

uniqueID: UniqueID

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

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

val userSource: TypedPipe[CombinedUser] =

DAL

.readMostRecentSnapshotNoOlderThan(UsersourceScalaDataset, Days(7)).toTypedPipe

val validUsers: TypedPipe[UserId] = getValidUsers(userSource).forceToDisk

//Dataset read operations

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

val tweetSource: TypedPipe[InteractionEvent] =

ExternalDataSources.ieSourceTweetEngagementsSource(dateRange =

DateRange(dateRange.end - Days(14), dateRange.end))

// user-user fav edges

val userUserFavEdges: TypedPipe[(UserId, UserId, Double)] =

ExternalDataSources.getFavEdges(HalfLifeInDaysForFavScore)

// user-user follow edges

val userUserFollowEdges: TypedPipe[(UserId, UserId)] =

filterInvalidUsers(ExternalDataSources.flockFollowsSource, validUsers)

// user-user block edges

val userUserBlockEdges: TypedPipe[(UserId, UserId)] =

filterInvalidUsers(ExternalDataSources.flockBlocksSource, validUsers)

// user-user abuse report edges

val userUserAbuseReportEdges: TypedPipe[(UserId, UserId)] =

filterInvalidUsers(ExternalDataSources.flockReportAsAbuseSource, validUsers)

// user-user spam report edges

val userUserSpamReportEdges: TypedPipe[(UserId, UserId)] =

filterInvalidUsers(ExternalDataSources.flockReportAsSpamSource, validUsers)

// user-signup country edges

val userSignUpCountryEdges: TypedPipe[(UserId, (Country, Language))] =

ExternalDataSources.userSource

// user-consumed language edges

val userConsumedLanguageEdges: TypedPipe[(UserId, Seq[(Language, Double)])] =

ExternalDataSources.inferredUserConsumedLanguageSource

// user-topic follow edges

val topicUserFollowedByEdges: TypedPipe[(TopicId, UserId)] =

ExternalDataSources.topicFollowGraphSource

// user-MRNotifOpenOrClick events from last 7 days

val userMRNotifOpenOrClickEvents: TypedPipe[MagicRecsNotificationLite] =

ExternalDataSources.magicRecsNotficationOpenOrClickEventsSource(dateRange =

DateRange(dateRange.end - Days(7), dateRange.end))

// user-searchQuery strings from last 7 days

val userSearchQueryEdges: TypedPipe[(UserId, String)] =

ExternalDataSources.adaptiveSearchScribeLogsSource(dateRange =

DateRange(dateRange.end - Days(7), dateRange.end))

getUserTweetInteractionGraph(tweetSource) ++

getUserFavGraph(userUserFavEdges) ++

getUserFollowGraph(userUserFollowEdges) ++

getUserBlockGraph(userUserBlockEdges) ++

getUserAbuseReportGraph(userUserAbuseReportEdges) ++

getUserSpamReportGraph(userUserSpamReportEdges) ++

getUserSignUpCountryGraph(userSignUpCountryEdges) ++

getUserConsumedLanguagesGraph(userConsumedLanguageEdges) ++

getUserTopicFollowGraph(topicUserFollowedByEdges) ++

getMagicRecsNotifOpenOrClickTweetsGraph(userMRNotifOpenOrClickEvents) ++

getSearchGraph(userSearchQueryEdges)

}

}