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

import com.twitter.bijection.Injection

import com.twitter.frigate.common.constdb\_util.Injections

import com.twitter.frigate.common.constdb\_util.ScaldingUtil

import com.twitter.graph\_feature\_service.common.Configs

import com.twitter.graph\_feature\_service.common.Configs.\_

import com.twitter.interaction\_graph.scio.agg\_all.InteractionGraphHistoryAggregatedEdgeSnapshotScalaDataset

import com.twitter.interaction\_graph.scio.ml.scores.RealGraphInScoresScalaDataset

import com.twitter.interaction\_graph.thriftscala.FeatureName

import com.twitter.interaction\_graph.thriftscala.{EdgeFeature => TEdgeFeature}

import com.twitter.pluck.source.user\_audits.UserAuditFinalScalaDataset

import com.twitter.scalding.DateRange

import com.twitter.scalding.Days

import com.twitter.scalding.Execution

import com.twitter.scalding.Stat

import com.twitter.scalding.UniqueID

import com.twitter.scalding.typed.TypedPipe

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

import com.twitter.scalding\_internal.dalv2.remote\_access.AllowCrossClusterSameDC

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

import com.twitter.util.Time

import com.twitter.wtf.candidate.thriftscala.CandidateSeq

import java.nio.ByteBuffer

import java.util.TimeZone

trait GraphFeatureServiceMainJob extends GraphFeatureServiceBaseJob {

// keeping hdfsPath as a separate variable in order to override it in unit tests

protected val hdfsPath: String = BaseHdfsPath

protected def getShardIdForUser(userId: Long): Int = shardForUser(userId)

protected implicit val keyInj: Injection[Long, ByteBuffer] = Injections.long2Varint

protected implicit val valueInj: Injection[Long, ByteBuffer] = Injections.long2ByteBuffer

protected val bufferSize: Int = 1 << 26

protected val maxNumKeys: Int = 1 << 24

protected val numReducers: Int = NumGraphShards

protected val outputStreamBufferSize: Int = 1 << 26

protected final val shardingByKey = { (k: Long, \_: Long) =>

getShardIdForUser(k)

}

protected final val shardingByValue = { (\_: Long, v: Long) =>

getShardIdForUser(v)

}

private def writeGraphToDB(

graph: TypedPipe[(Long, Long)],

shardingFunction: (Long, Long) => Int,

path: String

)(

implicit dateRange: DateRange

): Execution[TypedPipe[(Int, Unit)]] = {

ScaldingUtil

.writeConstDB[Long, Long](

graph.withDescription(s"sharding $path"),

shardingFunction,

shardId =>

getTimedHdfsShardPath(

shardId,

getHdfsPath(path, Some(hdfsPath)),

Time.fromMilliseconds(dateRange.end.timestamp)

),

Int.MaxValue,

bufferSize,

maxNumKeys,

numReducers,

outputStreamBufferSize

)(

keyInj,

valueInj,

Ordering[(Long, Long)]

)

.forceToDiskExecution

}

def extractFeature(

featureList: Seq[TEdgeFeature],

featureName: FeatureName

): Option[Float] = {

featureList

.find(\_.name == featureName)

.map(\_.tss.ewma.toFloat)

.filter(\_ > 0.0)

}

/\*\*

\* Function to extract a subgraph (e.g., follow graph) from real graph and take top K by real graph

\* weight.

\*

\* @param input input real graph

\* @param edgeFilter filter function to only get the edges needed (e.g., only follow edges)

\* @param counter counter

\* @return a subgroup that contains topK, e.g., follow graph for each user.

\*/

private def getSubGraph(

input: TypedPipe[(Long, Long, EdgeFeature)],

edgeFilter: EdgeFeature => Boolean,

counter: Stat

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

input

.filter(c => edgeFilter(c.\_3))

.map {

case (srcId, destId, features) =>

(srcId, (destId, features.realGraphScore))

}

.group

// auto reducer estimation only allocates 15 reducers, so setting an explicit number here

.withReducers(2000)

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

.flatMap {

case (srcId, topKNeighbors) =>

counter.inc()

topKNeighbors.map {

case (destId, \_) =>

(srcId, destId)

}

}

}

def getMauIds()(implicit dateRange: DateRange, uniqueID: UniqueID): TypedPipe[Long] = {

val numMAUs = Stat("NUM\_MAUS")

val uniqueMAUs = Stat("UNIQUE\_MAUS")

DAL

.read(UserAuditFinalScalaDataset)

.withRemoteReadPolicy(AllowCrossClusterSameDC)

.toTypedPipe

.collect {

case user\_audit if user\_audit.isValid =>

numMAUs.inc()

user\_audit.userId

}

.distinct

.map { u =>

uniqueMAUs.inc()

u

}

}

def getRealGraphWithMAUOnly(

implicit dateRange: DateRange,

timeZone: TimeZone,

uniqueID: UniqueID

): TypedPipe[(Long, Long, EdgeFeature)] = {

val numMAUs = Stat("NUM\_MAUS")

val uniqueMAUs = Stat("UNIQUE\_MAUS")

val monthlyActiveUsers = DAL

.read(UserAuditFinalScalaDataset)

.withRemoteReadPolicy(AllowCrossClusterSameDC)

.toTypedPipe

.collect {

case user\_audit if user\_audit.isValid =>

numMAUs.inc()

user\_audit.userId

}

.distinct

.map { u =>

uniqueMAUs.inc()

u

}

.asKeys

val realGraphAggregates = DAL

.readMostRecentSnapshot(

InteractionGraphHistoryAggregatedEdgeSnapshotScalaDataset,

dateRange.embiggen(Days(5)))

.withRemoteReadPolicy(AllowCrossClusterSameDC)

.toTypedPipe

.map { edge =>

val featureList = edge.features

val edgeFeature = EdgeFeature(

edge.weight.getOrElse(0.0).toFloat,

extractFeature(featureList, FeatureName.NumMutualFollows),

extractFeature(featureList, FeatureName.NumFavorites),

extractFeature(featureList, FeatureName.NumRetweets),

extractFeature(featureList, FeatureName.NumMentions)

)

(edge.sourceId, (edge.destinationId, edgeFeature))

}

.join(monthlyActiveUsers)

.map {

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

(destId, (srcId, feature))

}

.join(monthlyActiveUsers)

.map {

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

(srcId, destId, feature)

}

realGraphAggregates

}

def getTopKFollowGraph(

implicit dateRange: DateRange,

timeZone: TimeZone,

uniqueID: UniqueID

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

val followGraphMauStat = Stat("NumFollowEdges\_MAU")

val mau: TypedPipe[Long] = getMauIds()

DAL

.readMostRecentSnapshot(RealGraphInScoresScalaDataset, dateRange.embiggen(Days(7)))

.withRemoteReadPolicy(AllowCrossClusterSameDC)

.toTypedPipe

.groupBy(\_.key)

.join(mau.asKeys)

.withDescription("filtering srcId by mau")

.flatMap {

case (\_, (KeyVal(srcId, CandidateSeq(candidates)), \_)) =>

followGraphMauStat.inc()

val topK = candidates.sortBy(-\_.score).take(TopKRealGraph)

topK.map { c => (srcId, c.userId) }

}

}

override def runOnDateRange(

enableValueGraphs: Option[Boolean],

enableKeyGraphs: Option[Boolean]

)(

implicit dateRange: DateRange,

timeZone: TimeZone,

uniqueID: UniqueID

): Execution[Unit] = {

val processValueGraphs = enableValueGraphs.getOrElse(Configs.EnableValueGraphs)

val processKeyGraphs = enableKeyGraphs.getOrElse(Configs.EnableKeyGraphs)

if (!processKeyGraphs && !processValueGraphs) {

// Skip the batch job

Execution.unit

} else {

// val favoriteGraphStat = Stat("NumFavoriteEdges")

// val retweetGraphStat = Stat("NumRetweetEdges")

// val mentionGraphStat = Stat("NumMentionEdges")

// val realGraphAggregates = getRealGraphWithMAUOnly

val followGraph = getTopKFollowGraph

// val mutualFollowGraph = followGraph.asKeys.join(followGraph.swap.asKeys).keys

// val favoriteGraph =

// getSubGraph(realGraphAggregates, \_.favoriteScore.isDefined, favoriteGraphStat)

// val retweetGraph =

// getSubGraph(realGraphAggregates, \_.retweetScore.isDefined, retweetGraphStat)

// val mentionGraph =

// getSubGraph(realGraphAggregates, \_.mentionScore.isDefined, mentionGraphStat)

val writeValDataSetExecutions = if (processValueGraphs) {

Seq(

(followGraph, shardingByValue, FollowOutValPath),

(followGraph.swap, shardingByValue, FollowInValPath)

// (mutualFollowGraph, shardingByValue, MutualFollowValPath),

// (favoriteGraph, shardingByValue, FavoriteOutValPath),

// (favoriteGraph.swap, shardingByValue, FavoriteInValPath),

// (retweetGraph, shardingByValue, RetweetOutValPath),

// (retweetGraph.swap, shardingByValue, RetweetInValPath),

// (mentionGraph, shardingByValue, MentionOutValPath),

// (mentionGraph.swap, shardingByValue, MentionInValPath)

)

} else {

Seq.empty

}

val writeKeyDataSetExecutions = if (processKeyGraphs) {

Seq(

(followGraph, shardingByKey, FollowOutKeyPath),

(followGraph.swap, shardingByKey, FollowInKeyPath)

// (favoriteGraph, shardingByKey, FavoriteOutKeyPath),

// (favoriteGraph.swap, shardingByKey, FavoriteInKeyPath),

// (retweetGraph, shardingByKey, RetweetOutKeyPath),

// (retweetGraph.swap, shardingByKey, RetweetInKeyPath),

// (mentionGraph, shardingByKey, MentionOutKeyPath),

// (mentionGraph.swap, shardingByKey, MentionInKeyPath),

// (mutualFollowGraph, shardingByKey, MutualFollowKeyPath)

)

} else {

Seq.empty

}

Execution

.sequence((writeValDataSetExecutions ++ writeKeyDataSetExecutions).map {

case (graph, shardingMethod, path) =>

writeGraphToDB(graph, shardingMethod, path)

}).unit

}

}

}