package com.twitter.simclusters\_v2.scalding

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

import com.twitter.logging.Logger

import com.twitter.scalding.\_

import com.twitter.scalding.typed.TypedPipe

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

import com.twitter.scalding\_internal.dalv2.DALWrite.\_

import com.twitter.scalding\_internal.dalv2.remote\_access.{ExplicitLocation, ProcAtla}

import com.twitter.scalding\_internal.job.analytics\_batch.{

AnalyticsBatchExecution,

AnalyticsBatchExecutionArgs,

BatchDescription,

BatchFirstTime,

BatchIncrement,

TwitterScheduledExecutionApp

}

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

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

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

import com.twitter.simclusters\_v2.scalding.common.Util

import com.twitter.simclusters\_v2.thriftscala.{ClustersUserIsKnownFor, UserToKnownForClusterScores}

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

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

import java.util.TimeZone

object KnownForSources {

implicit val tz: TimeZone = DateOps.UTC

implicit val parser: DateParser = DateParser.default

def readDALDataset(

d: KeyValDALDataset[KeyVal[Long, ClustersUserIsKnownFor]],

noOlderThan: Duration,

modelVersionToKeep: String

): TypedPipe[(Long, Array[(Int, Float)])] = {

fromKeyVal(

DAL

.readMostRecentSnapshotNoOlderThan(d, noOlderThan)

.withRemoteReadPolicy(ExplicitLocation(ProcAtla))

.toTypedPipe,

modelVersionToKeep

)

}

def fromKeyVal(

in: TypedPipe[KeyVal[Long, ClustersUserIsKnownFor]],

modelVersionToKeep: String

): TypedPipe[(Long, Array[(Int, Float)])] = {

in.collect {

case KeyVal(userId, knownForClusters)

if knownForClusters.knownForModelVersion == modelVersionToKeep =>

(

userId,

knownForClusters.clusterIdToScores.toArray

.map {

case (clusterId, scores) =>

(clusterId, scores.knownForScore.getOrElse(0.0).toFloat)

}

.sortBy(-\_.\_2))

}

}

def toKeyVal(

in: TypedPipe[(Long, Array[(Int, Float)])],

modelVersion: String

): TypedPipe[KeyVal[Long, ClustersUserIsKnownFor]] = {

in.map {

case (userId, clustersArray) =>

val mappedClusters = clustersArray.map {

case (clusterId, score) =>

(clusterId, UserToKnownForClusterScores(Some(score)))

}.toMap

KeyVal(userId, ClustersUserIsKnownFor(modelVersion, mappedClusters))

}

}

val knownFor\_20M\_Dec11\_145K: TypedPipe[(Long, Array[(Int, Float)])] = readDALDataset(

SimclustersV2KnownFor20M145KDec11ScalaDataset,

Days(30),

ModelVersions.Model20M145KDec11

)

val knownFor\_20M\_145K\_updated: TypedPipe[(Long, Array[(Int, Float)])] = readDALDataset(

SimclustersV2KnownFor20M145KUpdatedScalaDataset,

Days(30),

ModelVersions.Model20M145KUpdated

)

val clusterToKnownFor\_20M\_Dec11\_145K: TypedPipe[(Int, List[(Long, Float)])] =

transpose(

knownFor\_20M\_Dec11\_145K

)

val clusterToKnownFor\_20M\_145K\_updated: TypedPipe[(Int, List[(Long, Float)])] =

transpose(

knownFor\_20M\_145K\_updated

)

private val log = Logger()

def readKnownFor(textFile: String): TypedPipe[(Long, Array[(Int, Float)])] = {

TypedPipe

.from(TextLine(textFile))

.flatMap { str =>

if (!str.startsWith("#")) {

try {

val tokens = str.trim.split("\\s+")

val res = Array.newBuilder[(Int, Float)]

val userId = tokens(0).toLong

for (i <- 1 until tokens.length) {

val Array(cIdStr, scoreStr) = tokens(i).split(":")

val clusterId = cIdStr.toInt

val score = scoreStr.toFloat

val newEntry = (clusterId, score)

res += newEntry

}

val result = res.result

if (result.nonEmpty) {

Some((userId, res.result()))

} else None

} catch {

case ex: Throwable =>

log.warning(

s"Error while loading knownFor from $textFile for line <$str>: " +

ex.getMessage

)

None

}

} else None

}

}

def stringifyKnownFor(

input: TypedPipe[(Long, Array[(Int, Float)])]

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

input.mapValues { arr =>

arr.map { case (clusterId, score) => "%d:%.2g".format(clusterId, score) }.mkString("\t")

}

}

def writeKnownForTypedTsv(

input: TypedPipe[(Long, Array[(Int, Float)])],

outputDir: String

): Execution[Unit] = {

stringifyKnownFor(input).writeExecution(TypedTsv(outputDir))

}

def makeKnownForTypedTsv(

input: TypedPipe[(Long, Array[(Int, Float)])],

outputDir: String

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

Execution.getMode.flatMap { mode =>

try {

val dest = TextLine(outputDir)

dest.validateTaps(mode)

Execution.from(KnownForSources.readKnownFor(outputDir))

} catch {

case ivs: InvalidSourceException =>

writeKnownForTypedTsv(input, outputDir).map { \_ => input }

}

}

}

def transpose(

userToCluster: TypedPipe[(Long, Array[(Int, Float)])]

): TypedPipe[(Int, List[(Long, Float)])] = {

userToCluster

.flatMap {

case (userId, clusterWeightPairs) =>

clusterWeightPairs.map {

case (clusterId, weight) =>

(clusterId, List(userId -> weight))

}

}

.sumByKey

.toTypedPipe

}

}

/\*\*

capesospy-v2 update --build\_locally --start\_cron known\_for\_to\_mh \

src/scala/com/twitter/simclusters\_v2/capesos\_config/atla\_proc.yaml

\*/

object KnownForToMHBatch extends TwitterScheduledExecutionApp {

import KnownForSources.\_

/\*\*

\* A simple update function which updates the source by removing deactivated and suspended users.

\* This will be eventually replaced by a regular cluster updating method.

\*/

def updateKnownForSource(

knownForSource: TypedPipe[(Long, ClustersUserIsKnownFor)],

userSource: TypedPipe[FlatUser]

)(

implicit uniqueID: UniqueID

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

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

val numInvalidUsers = Stat("num\_invalid\_users")

val numKnownForUsersLeft = Stat("num\_known\_for\_users\_left")

val numRemovedKnownForUsers = Stat("num\_removed\_known\_for\_users")

val validUsers =

userSource.flatMap {

case flatUser

if !flatUser.deactivated.contains(true) && !flatUser.suspended

.contains(true)

&& flatUser.id.nonEmpty =>

numValidUsers.inc()

flatUser.id

case \_ =>

numInvalidUsers.inc()

None

}

knownForSource.leftJoin(validUsers.asKeys).flatMap {

case (userId, (clustersWithScore, Some(\_))) =>

numKnownForUsersLeft.inc()

Some((userId, clustersWithScore))

case \_ =>

numRemovedKnownForUsers.inc()

None

}

}

// this should happen before InterestedInFromKnownForBatch

private val firstTime: String = "2019-03-22"

private val batchIncrement: Duration = Days(7)

private val outputPath: String = InternalDataPaths.RawKnownForDec11Path

private val execArgs = AnalyticsBatchExecutionArgs(

batchDesc = BatchDescription(this.getClass.getName.replace("$", "")),

firstTime = BatchFirstTime(RichDate(firstTime)),

lastTime = None,

batchIncrement = BatchIncrement(batchIncrement)

)

override def scheduledJob: Execution[Unit] =

AnalyticsBatchExecution(execArgs) { implicit dateRange =>

Execution.withId { implicit uniqueId =>

val numKnownForUsers = Stat("num\_known\_for\_users")

val userSource =

DAL

.readMostRecentSnapshotNoOlderThan(UsersourceFlatScalaDataset, Days(7))

.toTypedPipe

val knownForData = DAL

.readMostRecentSnapshotNoOlderThan(

SimclustersV2RawKnownFor20M145KDec11ScalaDataset,

Days(30))

.toTypedPipe

.map {

case KeyVal(userId, knownForClusters) =>

numKnownForUsers.inc()

(userId, knownForClusters)

}

val result = updateKnownForSource(knownForData, userSource).map {

case (userId, knownForClusters) =>

KeyVal(userId, knownForClusters)

}

Util.printCounters(

result.writeDALVersionedKeyValExecution(

dataset = SimclustersV2RawKnownFor20M145KDec11ScalaDataset,

pathLayout = D.Suffix(outputPath)

)

)

}

}

}