package com.twitter.simclusters\_v2.scalding

import com.twitter.algebird.DecayedValue

import com.twitter.algebird.DecayedValueMonoid

import com.twitter.algebird.Monoid

import com.twitter.algebird.Semigroup

import com.twitter.conversions.DurationOps.\_

import com.twitter.logging.Logger

import com.twitter.scalding.\_

import com.twitter.scalding.typed.UnsortedGrouped

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

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

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

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

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

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

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

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

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

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

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

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

import com.twitter.timelineservice.thriftscala.ContextualizedFavoriteEvent

import com.twitter.timelineservice.thriftscala.FavoriteEventUnion

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

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

import com.twitter.util.Time

import twadoop\_config.configuration.log\_categories.group.timeline.TimelineServiceFavoritesScalaDataset

sealed trait FavState

object Fav extends FavState

object UnFavWithoutPriorFav extends FavState

object UnFavWithPriorFav extends FavState

case class TimestampedFavState(favOrUnfav: FavState, timestampMillis: Long)

object TimestampedFavStateSemigroup extends Semigroup[TimestampedFavState] {

override def plus(left: TimestampedFavState, right: TimestampedFavState): TimestampedFavState = {

/\*\*

\* Assigning to first, second ensures commutative property

\*/

val (first, second) = if (left.timestampMillis < right.timestampMillis) {

(left, right)

} else {

(right, left)

}

(first.favOrUnfav, second.favOrUnfav) match {

case (\_, UnFavWithPriorFav) => second

case (UnFavWithPriorFav, UnFavWithoutPriorFav) =>

TimestampedFavState(UnFavWithPriorFav, second.timestampMillis)

case (Fav, UnFavWithoutPriorFav) =>

TimestampedFavState(UnFavWithPriorFav, second.timestampMillis)

case (UnFavWithoutPriorFav, UnFavWithoutPriorFav) => second

case (\_, Fav) => second

}

}

}

object UserUserFavGraph {

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

// setting the prune threshold in the monoid below to 0.0, since we want to do our own pruning

// outside the monoid, primarily to be able to count how many scores are pruned.

implicit val dvMonoid: Monoid[DecayedValue] = DecayedValueMonoid(0.0)

implicit val lfvSemigroup: Semigroup[TimestampedFavState] = TimestampedFavStateSemigroup

def getSummedFavGraph(

previousGraphOpt: Option[TypedPipe[EdgeWithDecayedWeights]],

newFavsDateRange: DateRange,

halfLivesInDays: List[Int],

minScoreToKeep: Double

)(

implicit uniqueID: UniqueID

): TypedPipe[EdgeWithDecayedWeights] = {

val newFavs = DAL.read(TimelineServiceFavoritesScalaDataset, newFavsDateRange).toTypedPipe

val endTime = Time.fromMilliseconds(newFavsDateRange.end.timestamp)

val userSource =

DAL.readMostRecentSnapshotNoOlderThan(UsersourceFlatScalaDataset, Days(7)).toTypedPipe

getSummedFavGraphWithValidUsers(

previousGraphOpt,

newFavs,

halfLivesInDays,

endTime,

minScoreToKeep,

userSource

)

}

def getSummedFavGraphWithValidUsers(

previousGraphOpt: Option[TypedPipe[EdgeWithDecayedWeights]],

newFavs: TypedPipe[ContextualizedFavoriteEvent],

halfLivesInDays: List[Int],

endTime: Time,

minScoreToKeep: Double,

userSource: TypedPipe[FlatUser]

)(

implicit uniqueID: UniqueID

): TypedPipe[EdgeWithDecayedWeights] = {

val fullGraph = getSummedFavGraph(

previousGraphOpt,

newFavs,

halfLivesInDays,

endTime,

minScoreToKeep

)

removeDeactivedOrSuspendedUsers(fullGraph, userSource)

}

def processRawFavEvents(

favsOrUnfavs: TypedPipe[ContextualizedFavoriteEvent]

)(

implicit uniqueID: UniqueID

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

val numFavsBeforeUniq = Stat("num\_favs\_before\_uniq")

val numUnFavsBeforeUniq = Stat("num\_unfavs\_before\_uniq")

val numFinalFavs = Stat("num\_final\_favs")

val numUnFavsWithPriorFavs = Stat("num\_unfavs\_with\_prior\_favs")

val numUnFavsWithoutPriorFavs = Stat("num\_unfavs\_without\_prior\_favs")

favsOrUnfavs

.flatMap { cfe: ContextualizedFavoriteEvent =>

cfe.event match {

case FavoriteEventUnion.Favorite(fav) =>

numFavsBeforeUniq.inc()

Some(

(

(fav.userId, fav.tweetId, fav.tweetUserId),

TimestampedFavState(Fav, fav.eventTimeMs)))

case FavoriteEventUnion.Unfavorite(unfav) =>

numUnFavsBeforeUniq.inc()

Some(

(

(unfav.userId, unfav.tweetId, unfav.tweetUserId),

TimestampedFavState(UnFavWithoutPriorFav, unfav.eventTimeMs)))

case \_ => None

}

}

.sumByKey

.toTypedPipe

.flatMap {

case fav @ (\_, TimestampedFavState(Fav, \_)) =>

numFinalFavs.inc()

Some(fav)

case unfav @ (\_, TimestampedFavState(UnFavWithoutPriorFav, \_)) =>

numUnFavsWithoutPriorFavs.inc()

Some(unfav)

case (\_, TimestampedFavState(UnFavWithPriorFav, \_)) =>

numUnFavsWithPriorFavs.inc()

None

}

}

private def getGraphFromNewFavsOnly(

newFavs: TypedPipe[ContextualizedFavoriteEvent],

halfLivesInDays: List[Int],

endTime: Time

)(

implicit uniqueID: UniqueID

): UnsortedGrouped[(UserId, UserId), Map[Int, DecayedValue]] = {

val numEventsNewerThanEndTime = Stat("num\_events\_newer\_than\_endtime")

processRawFavEvents(newFavs).map {

case ((userId, \_, authorId), TimestampedFavState(favOrUnfav, timestampMillis)) =>

val halfLifeInDaysToScores = halfLivesInDays.map { halfLifeInDays =>

val givenTime = Time.fromMilliseconds(timestampMillis)

if (givenTime > endTime) {

// technically this should never happen, and even if it did happen,

// we shouldn't have to care, but I'm noticing that the weights aren't being computed

// correctly for events that spilled over the edge

numEventsNewerThanEndTime.inc()

}

val timeInSeconds = math.min(givenTime.inSeconds, endTime.inSeconds)

val value = favOrUnfav match {

case Fav => 1.0

case UnFavWithoutPriorFav => -1.0

case UnFavWithPriorFav => 0.0

}

val decayedValue = DecayedValue.build(value, timeInSeconds, halfLifeInDays.days.inSeconds)

halfLifeInDays -> decayedValue

}

((userId, authorId), halfLifeInDaysToScores.toMap)

}.sumByKey

}

def getSummedFavGraph(

previousGraphOpt: Option[TypedPipe[EdgeWithDecayedWeights]],

newFavs: TypedPipe[ContextualizedFavoriteEvent],

halfLivesInDays: List[Int],

endTime: Time,

minScoreToKeep: Double

)(

implicit uniqueID: UniqueID

): TypedPipe[EdgeWithDecayedWeights] = {

val prunedScoresCounter = Stat("num\_pruned\_scores")

val negativeScoresCounter = Stat("num\_negative\_scores")

val prunedEdgesCounter = Stat("num\_pruned\_edges")

val keptEdgesCounter = Stat("num\_kept\_edges")

val keptScoresCounter = Stat("num\_kept\_scores")

val numCommonEdges = Stat("num\_common\_edges")

val numNewEdges = Stat("num\_new\_edges")

val numOldEdges = Stat("num\_old\_edges")

val unprunedOuterJoinedGraph = previousGraphOpt match {

case Some(previousGraph) =>

previousGraph

.map {

case EdgeWithDecayedWeights(srcId, destId, decayedSums) =>

val ts = decayedSums.lastUpdatedTimestamp.toDouble / 1000

val map = decayedSums.halfLifeInDaysToDecayedSums.map {

case (halfLifeInDays, value) =>

halfLifeInDays -> DecayedValue.build(value, ts, halfLifeInDays.days.inSeconds)

}.toMap

((srcId, destId), map)

}

.outerJoin(getGraphFromNewFavsOnly(newFavs, halfLivesInDays, endTime))

.toTypedPipe

case None =>

getGraphFromNewFavsOnly(newFavs, halfLivesInDays, endTime).toTypedPipe

.map {

case ((srcId, destId), scoreMap) =>

((srcId, destId), (None, Some(scoreMap)))

}

}

unprunedOuterJoinedGraph

.flatMap {

case ((srcId, destId), (previousScoreMapOpt, newScoreMapOpt)) =>

val latestTimeDecayedValues = halfLivesInDays.map { hlInDays =>

hlInDays -> DecayedValue.build(0, endTime.inSeconds, hlInDays.days.inSeconds)

}.toMap

val updatedDecayedValues =

Monoid.sum(

List(previousScoreMapOpt, newScoreMapOpt, Some(latestTimeDecayedValues)).flatten)

(previousScoreMapOpt, newScoreMapOpt) match {

case (Some(pm), None) => numOldEdges.inc()

case (None, Some(nm)) => numNewEdges.inc()

case (Some(pm), Some(nm)) => numCommonEdges.inc()

}

val prunedMap = updatedDecayedValues.flatMap {

case (hlInDays, decayedValue) =>

if (decayedValue.value < minScoreToKeep) {

if (decayedValue.value < 0) {

negativeScoresCounter.inc()

}

prunedScoresCounter.inc()

None

} else {

keptScoresCounter.inc()

Some((hlInDays, decayedValue.value))

}

}

if (prunedMap.nonEmpty) {

keptEdgesCounter.inc()

Some(EdgeWithDecayedWeights(srcId, destId, DecayedSums(endTime.inMillis, prunedMap)))

} else {

prunedEdgesCounter.inc()

None

}

}

}

def removeDeactivedOrSuspendedUsers(

full: TypedPipe[EdgeWithDecayedWeights],

userSource: TypedPipe[FlatUser]

)(

implicit uniqueID: UniqueID

): TypedPipe[EdgeWithDecayedWeights] = {

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

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

val numEdgesBeforeUsersourceJoin = Stat("num\_edges\_before\_join\_with\_usersource")

val numEdgesWithValidSource = Stat("num\_edges\_with\_valid\_source")

val numEdgesWithValidSourceAndDest = Stat("num\_edges\_with\_valid\_source\_and\_dest")

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

}.forceToDisk // avoid reading in the whole of userSource for both of the joins below

val toJoin = full.map { edge =>

numEdgesBeforeUsersourceJoin.inc()

(edge.sourceId, edge)

}

toJoin

.join(validUsers.asKeys)

.map {

case (\_, (edge, \_)) =>

numEdgesWithValidSource.inc()

(edge.destinationId, edge)

}

.join(validUsers.asKeys)

.map {

case (\_, (edge, \_)) =>

numEdgesWithValidSourceAndDest.inc()

edge

}

}

}

/\*\*

\* ./bazel bundle src/scala/com/twitter/simclusters\_v2/scalding:fav\_graph\_adhoc && \

\* oscar hdfs --user frigate --host hadoopnest1.atla.twitter.com --bundle fav\_graph\_adhoc \

\* --tool com.twitter.simclusters\_v2.scalding.UserUserFavGraphAdhoc --screen --screen-detached \

\* --tee logs/userUserFavGraphAdhoc\_20170101 -- --date 2017-01-01 --halfLivesInDays 14 50 100 \

\* --outputDir /user/frigate/your\_ldap/userUserFavGraphAdhoc\_20170101\_hl14\_50\_100

\*

\* ./bazel bundle src/scala/com/twitter/simclusters\_v2/scalding:fav\_graph\_adhoc && \

\* oscar hdfs --user frigate --host hadoopnest1.atla.twitter.com --bundle fav\_graph\_adhoc \

\* --tool com.twitter.simclusters\_v2.scalding.UserUserFavGraphAdhoc --screen --screen-detached \

\* --tee logs/userUserFavGraphAdhoc\_20170102\_addPrevious20170101 -- --date 2017-01-02 \

\* --previousGraphDir /user/frigate/your\_ldap/userUserFavGraphAdhoc\_20170101\_hl14\_50\_100 \

\* --halfLivesInDays 14 50 100 \

\* --outputDir /user/frigate/your\_ldap/userUserFavGraphAdhoc\_20170102\_addPrevious20170101\_hl14\_50\_100

\*/

object UserUserFavGraphAdhoc 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 previousGraphOpt = args.optional("previousGraphDir").map { dir =>

TypedPipe.from(EdgeWithDecayedWtsFixedPathSource(dir))

}

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

val halfLives = args.list("halfLivesInDays").map(\_.toInt)

val minScoreToKeep = args.double("minScoreToKeep", 1e-5)

val outputDir = args("outputDir")

Util.printCounters(

UserUserFavGraph

.getSummedFavGraph(previousGraphOpt, favsDateRange, halfLives, minScoreToKeep)

.writeExecution(EdgeWithDecayedWtsFixedPathSource(outputDir))

)

}

}

}

/\*\*

\* $ capesospy-v2 update --start\_cron fav\_graph src/scala/com/twitter/simclusters\_v2/capesos\_config/atla\_proc.yaml

\*/

object UserUserFavGraphBatch extends TwitterScheduledExecutionApp {

private val firstTime: String = "2017-01-01"

implicit val tz = DateOps.UTC

implicit val parser = DateParser.default

private val batchIncrement: Duration = Days(2)

private val firstStartDate = DateRange.parse(firstTime).start

val outputPath: String = "/user/cassowary/processed/user\_user\_fav\_graph"

val log = Logger()

private val execArgs = AnalyticsBatchExecutionArgs(

batchDesc = BatchDescription(this.getClass.getName),

firstTime = BatchFirstTime(RichDate(firstTime)),

lastTime = None,

batchIncrement = BatchIncrement(batchIncrement)

)

override def scheduledJob: Execution[Unit] = AnalyticsBatchExecution(execArgs) { dateRange =>

Execution.withId { implicit uniqueId =>

Execution.withArgs { args =>

val previousGraph = if (dateRange.start.timestamp == firstStartDate.timestamp) {

log.info("Looks like this is the first time, setting previousGraph to None")

None

} else {

Some(

DAL

.readMostRecentSnapshot(UserUserFavGraphScalaDataset, dateRange - batchIncrement)

.toTypedPipe

)

}

val halfLives = args.list("halfLivesInDays").map(\_.toInt)

val minScoreToKeep = args.double("minScoreToKeep", 1e-5)

Util.printCounters(

UserUserFavGraph

.getSummedFavGraph(previousGraph, dateRange, halfLives, minScoreToKeep)

.writeDALSnapshotExecution(

UserUserFavGraphScalaDataset,

D.Daily,

D.Suffix(outputPath),

D.EBLzo(),

dateRange.end)

)

}

}

}

}

object DumpFavGraphAdhoc extends TwitterExecutionApp {

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

def job: Execution[Unit] =

Execution.getConfigMode.flatMap {

case (config, mode) =>

Execution.withId { implicit uniqueId =>

val favGraph = DAL

.readMostRecentSnapshotNoOlderThan(UserUserFavGraphScalaDataset, Days(10))

.withRemoteReadPolicy(ExplicitLocation(ProcAtla))

.toTypedPipe

.collect {

case edge if edge.weights.halfLifeInDaysToDecayedSums.contains(100) =>

(edge.sourceId, edge.destinationId, edge.weights.halfLifeInDaysToDecayedSums(100))

}

Execution

.sequence(

Seq(

Util.printSummaryOfNumericColumn(

favGraph.map(\_.\_3),

Some("Weight")

),

Util.printSummaryOfNumericColumn(

favGraph.map(c => math.log10(10.0 + c.\_3)),

Some("Weight\_Log\_P10")

),

Util.printSummaryOfNumericColumn(

favGraph.map(c => math.log10(1.0 + c.\_3)),

Some("Weight\_Log\_P1")

),

Util.printSummaryOfCategoricalColumn(favGraph.map(\_.\_1), Some("SourceId")),

Util.printSummaryOfCategoricalColumn(favGraph.map(\_.\_2), Some("DestId"))

)

).flatMap { summarySeq =>

println(summarySeq.mkString("\n"))

Execution.unit

}

}

}

}