package com.twitter.graph.batch.job.tweepcred

import com.twitter.data.proto.Flock

import com.twitter.scalding.\_

import com.twitter.pluck.source.\_

import com.twitter.pluck.source.combined\_user\_source.MostRecentCombinedUserSnapshotSource

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

import com.twitter.service.interactions.InteractionGraph

import graphstore.common.FlockFollowsJavaDataset

import java.util.TimeZone

/\*\*

\* Prepare the graph data for page rank calculation. Also generate the initial

\* pagerank as the starting point. Afterwards, start WeightedPageRank job.

\*

\* Either read a tsv file for testing or read the following to build the graph

\* flock edges Flock.Edge

\* real graph input for weights InteractionGraph.Edge

\*

\* Options:

\* --pwd: working directory, will generate the following files there

\* numnodes: total number of nodes

\* nodes: nodes file <'src\_id, 'dst\_ids, 'weights, 'mass\_prior>

\* pagerank: the page rank file

\* --user\_mass: user mass tsv file, generated by twadoop user\_mass job

\* Optional arguments:

\* --input: use the given tsv file instead of flock and real graph

\* --weighted: do weighted pagerank, default false

\* --flock\_edges\_only: restrict graph to flock edges, default true

\* --input\_pagerank: continue pagerank from this

\*

\* Plus the following options for WeightedPageRank and ExtractTweepcred:

\* --output\_pagerank: where to put pagerank file

\* --output\_tweepcred: where to put tweepcred file

\* Optional:

\* --maxiterations: how many iterations to run. Default is 20

\* --jumpprob: probability of a random jump, default is 0.1

\* --threshold: total difference before finishing early, default 0.001

\* --post\_adjust: whether to do post adjust, default true

\*/

class PreparePageRankData(args: Args) extends Job(args) {

implicit val timeZone: TimeZone = DateOps.UTC

val PWD = args("pwd")

val WEIGHTED = args.getOrElse("weighted", "false").toBoolean

val FLOCK\_EDGES\_ONLY = args.getOrElse("flock\_edges\_only", "true").toBoolean

val ROW\_TYPE\_1 = 1

val ROW\_TYPE\_2 = 2

// graph data and user mass

val userMass = getUserMass

val nodesWithPrior = getGraphData(userMass)

val numNodes = nodesWithPrior.groupAll { \_.size }

numNodes.write(Tsv(PWD + "/numnodes"))

dumpNodes(nodesWithPrior, PWD + "/nodes");

// initial pagerank to start computation

generateInitialPagerank(nodesWithPrior)

// continue with the calculation

override def next = {

Some(new WeightedPageRank(args))

}

/\*\*

\* read flock edges

\*/

def getFlockEdges = {

DAL

.readMostRecentSnapshotNoOlderThan(FlockFollowsJavaDataset, Days(7))

.toTypedSource

.flatMapTo('src\_id, 'dst\_id) { edge: Flock.Edge =>

if (edge.getStateId() == Flock.State.Positive.getNumber()) {

Some((edge.getSourceId(), edge.getDestinationId()))

} else {

None

}

}

}

/\*\*

\* read real graph edges with weights

\*/

def getRealGraphEdges = {

RealGraphEdgeSource()

.flatMapTo('src\_id, 'dst\_id, 'weight) { edge: InteractionGraph.Edge =>

if (edge.getSourceId() != edge.getDestinationId()) {

val srcId = edge.getSourceId()

val dstId = edge.getDestinationId()

val weight = edge.getWeight().toFloat

Some((srcId, dstId, weight))

} else {

None

}

}

}

/\*\*

\* combine real graph and flock. If flock\_edges\_only is true, only take the

\* flock edges; otherwise edges are either from flock or from real graph.

\* edges weights default to be 1, overwritten by weights from real graph

\*/

def getFlockRealGraphEdges = {

val flock = getFlockEdges

if (WEIGHTED) {

val flockWithWeight = flock

.map(() -> ('weight, 'rowtype)) { (u: Unit) =>

(1.0f, ROW\_TYPE\_1)

}

val realGraph = getRealGraphEdges

.map(() -> 'rowtype) { (u: Unit) =>

(ROW\_TYPE\_2)

}

val combined = (flockWithWeight ++ realGraph)

.groupBy('src\_id, 'dst\_id) {

\_.min('rowtype)

.max('weight) // take whichever is bigger

}

if (FLOCK\_EDGES\_ONLY) {

combined.filter('rowtype) { (rowtype: Int) =>

rowtype == ROW\_TYPE\_1

}

} else {

combined

}

} else {

flock.map(() -> ('weight)) { (u: Unit) =>

1.0f

}

}.project('src\_id, 'dst\_id, 'weight)

}

def getCsvEdges(fileName: String) = {

Tsv(fileName).read

.mapTo((0, 1, 2) -> ('src\_id, 'dst\_id, 'weight)) { input: (Long, Long, Float) =>

input

}

}

/\*

\* Compute user mass based on combined user

\*/

def getUserMass =

TypedPipe

.from(MostRecentCombinedUserSnapshotSource)

.flatMap { user =>

UserMass.getUserMass(user)

}

.map { userMassInfo =>

(userMassInfo.userId, userMassInfo.mass)

}

.toPipe[(Long, Double)]('src\_id\_input, 'mass\_prior)

.normalize('mass\_prior)

/\*\*

\* Read either flock/real\_graph or a given tsv file

\* group by the source id, and output node data structure

\* merge with the user\_mass.

\* return <'src\_id, 'dst\_ids, 'weights, 'mass\_prior>

\*

\* make sure src\_id is the same set as in user\_mass, and dst\_ids

\* are subset of user\_mass. eg flock has edges like 1->2,

\* where both users 1 and 2 do not exist anymore

\*/

def getGraphData(userMass: RichPipe) = {

val edges: RichPipe = args.optional("input") match {

case None => getFlockRealGraphEdges

case Some(input) => getCsvEdges(input)

}

// remove edges where dst\_id is not in userMass

val filterByDst = userMass

.joinWithLarger('src\_id\_input -> 'dst\_id, edges)

.discard('src\_id\_input, 'mass\_prior)

// aggreate by the source id

val nodes = filterByDst

.groupBy('src\_id) {

\_.mapReduceMap(('dst\_id, 'weight) -> ('dst\_ids, 'weights)) /\* map1 \*/ { a: (Long, Float) =>

(Vector(a.\_1), if (WEIGHTED) Vector(a.\_2) else Vector())

} /\* reduce \*/ { (a: (Vector[Long], Vector[Float]), b: (Vector[Long], Vector[Float])) =>

{

(a.\_1 ++ b.\_1, a.\_2 ++ b.\_2)

}

} /\* map2 \*/ { a: (Vector[Long], Vector[Float]) =>

a

}

}

.mapTo(

('src\_id, 'dst\_ids, 'weights) -> ('src\_id, 'dst\_ids, 'weights, 'mass\_prior, 'rowtype)) {

input: (Long, Vector[Long], Vector[Float]) =>

{

(input.\_1, input.\_2.toArray, input.\_3.toArray, 0.0, ROW\_TYPE\_1)

}

}

// get to the same schema

val userMassNodes = userMass

.mapTo(('src\_id\_input, 'mass\_prior) -> ('src\_id, 'dst\_ids, 'weights, 'mass\_prior, 'rowtype)) {

input: (Long, Double) =>

{

(input.\_1, Array[Long](), Array[Float](), input.\_2, ROW\_TYPE\_2)

}

}

// make src\_id the same set as in userMass

(nodes ++ userMassNodes)

.groupBy('src\_id) {

\_.sortBy('rowtype)

.head('dst\_ids, 'weights)

.last('mass\_prior, 'rowtype)

}

.filter('rowtype) { input: Int =>

input == ROW\_TYPE\_2

}

}

/\*\*

\* generate the graph data output

\*/

def dumpNodes(nodes: RichPipe, fileName: String) = {

mode match {

case Hdfs(\_, conf) => nodes.write(SequenceFile(fileName))

case \_ =>

nodes

.mapTo((0, 1, 2, 3) -> (0, 1, 2, 3)) { input: (Long, Array[Long], Array[Float], Double) =>

(input.\_1, input.\_2.mkString(","), input.\_3.mkString(","), input.\_4)

}

.write(Tsv(fileName))

}

}

/\*

\* output prior mass or copy the given mass file (merge, normalize)

\* to be used as the starting point

\*/

def generateInitialPagerank(nodes: RichPipe) = {

val prior = nodes

.project('src\_id, 'mass\_prior)

val combined = args.optional("input\_pagerank") match {

case None => prior

case Some(fileName) => {

val massInput = Tsv(fileName).read

.mapTo((0, 1) -> ('src\_id, 'mass\_prior, 'rowtype)) { input: (Long, Double) =>

(input.\_1, input.\_2, ROW\_TYPE\_2)

}

val priorRow = prior

.map(() -> ('rowtype)) { (u: Unit) =>

ROW\_TYPE\_1

}

(priorRow ++ massInput)

.groupBy('src\_id) {

\_.sortBy('rowtype)

.last('mass\_prior)

.head('rowtype)

}

// throw away extra nodes from input file

.filter('rowtype) { (rowtype: Int) =>

rowtype == ROW\_TYPE\_1

}

.discard('rowtype)

.normalize('mass\_prior)

}

}

combined.write(Tsv(PWD + "/pagerank\_0"))

}

}