package com.twitter.follow\_recommendations.common.candidate\_sources.salsa

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.strato.generated.client.onboarding.userrecs.SalsaFirstDegreeOnUserClientColumn

import com.twitter.strato.generated.client.onboarding.userrecs.SalsaSecondDegreeOnUserClientColumn

import com.twitter.follow\_recommendations.common.models.AccountProof

import com.twitter.follow\_recommendations.common.models.CandidateUser

import com.twitter.follow\_recommendations.common.models.FollowProof

import com.twitter.follow\_recommendations.common.models.Reason

import com.twitter.stitch.Stitch

import com.twitter.wtf.candidate.thriftscala.Candidate

import javax.inject.Inject

import javax.inject.Singleton

case class SalsaExpandedCandidate(

candidateId: Long,

numberOfConnections: Int,

totalScore: Double,

connectingUsers: Seq[Long]) {

def toCandidateUser: CandidateUser =

CandidateUser(

id = candidateId,

score = Some(totalScore),

reason = Some(Reason(

Some(AccountProof(followProof = Some(FollowProof(connectingUsers, connectingUsers.size))))))

)

}

case class SimilarUserCandidate(candidateId: Long, score: Double, similarToCandidate: Long)

/\*\*

\* Salsa expander uses pre-computed lists of candidates for each input user id and returns the highest scored candidates in the pre-computed lists as the expansion for the corresponding input id.

\*/

@Singleton

class SalsaExpander @Inject() (

statsReceiver: StatsReceiver,

firstDegreeClient: SalsaFirstDegreeOnUserClientColumn,

secondDegreeClient: SalsaSecondDegreeOnUserClientColumn,

) {

val stats = statsReceiver.scope("salsa\_expander")

private def similarUsers(

input: Seq[Long],

neighbors: Seq[Option[Seq[Candidate]]]

): Seq[SalsaExpandedCandidate] = {

input

.zip(neighbors).flatMap {

case (recId, Some(neighbors)) =>

neighbors.map(neighbor => SimilarUserCandidate(neighbor.userId, neighbor.score, recId))

case \_ => Nil

}.groupBy(\_.candidateId).map {

case (key, neighbors) =>

val scores = neighbors.map(\_.score)

val connectingUsers = neighbors

.sortBy(-\_.score)

.take(SalsaExpander.MaxConnectingUsersToOutputPerExpandedCandidate)

.map(\_.similarToCandidate)

SalsaExpandedCandidate(key, scores.size, scores.sum, connectingUsers)

}

.filter(

\_.numberOfConnections >= math

.min(SalsaExpander.MinConnectingUsersThreshold, input.size)

)

.toSeq

}

def apply(

firstDegreeInput: Seq[Long],

secondDegreeInput: Seq[Long],

maxNumOfCandidatesToReturn: Int

): Stitch[Seq[CandidateUser]] = {

val firstDegreeNeighborsStitch =

Stitch

.collect(firstDegreeInput.map(firstDegreeClient.fetcher

.fetch(\_).map(\_.v.map(\_.candidates.take(SalsaExpander.MaxDirectNeighbors))))).onSuccess {

firstDegreeNeighbors =>

stats.stat("first\_degree\_neighbors").add(firstDegreeNeighbors.flatten.size)

}

val secondDegreeNeighborsStitch =

Stitch

.collect(

secondDegreeInput.map(

secondDegreeClient.fetcher

.fetch(\_).map(

\_.v.map(\_.candidates.take(SalsaExpander.MaxIndirectNeighbors))))).onSuccess {

secondDegreeNeighbors =>

stats.stat("second\_degree\_neighbors").add(secondDegreeNeighbors.flatten.size)

}

val neighborStitches =

Stitch.join(firstDegreeNeighborsStitch, secondDegreeNeighborsStitch).map {

case (first, second) => first ++ second

}

val similarUsersToInput = neighborStitches.map { neighbors =>

similarUsers(firstDegreeInput ++ secondDegreeInput, neighbors)

}

similarUsersToInput.map {

// Rank the candidate cot users by the combined weights from the connecting users. This is the default original implementation. It is unlikely to have weight ties and thus a second ranking function is not necessary.

\_.sortBy(-\_.totalScore)

.take(maxNumOfCandidatesToReturn)

.map(\_.toCandidateUser)

}

}

}

object SalsaExpander {

val MaxDirectNeighbors = 2000

val MaxIndirectNeighbors = 2000

val MinConnectingUsersThreshold = 2

val MaxConnectingUsersToOutputPerExpandedCandidate = 3

}