package com.twitter.cr\_mixer.candidate\_generation

import com.twitter.contentrecommender.thriftscala.TweetInfo

import com.twitter.cr\_mixer.config.TimeoutConfig

import com.twitter.cr\_mixer.model.FrsTweetCandidateGeneratorQuery

import com.twitter.cr\_mixer.model.ModuleNames

import com.twitter.cr\_mixer.model.TweetWithAuthor

import com.twitter.cr\_mixer.param.FrsParams

import com.twitter.cr\_mixer.similarity\_engine.EarlybirdSimilarityEngineRouter

import com.twitter.cr\_mixer.source\_signal.FrsStore

import com.twitter.cr\_mixer.source\_signal.FrsStore.FrsQueryResult

import com.twitter.cr\_mixer.thriftscala.FrsTweet

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.finagle.util.DefaultTimer

import com.twitter.frigate.common.util.StatsUtil

import com.twitter.hermit.constants.AlgorithmFeedbackTokens

import com.twitter.hermit.constants.AlgorithmFeedbackTokens.AlgorithmToFeedbackTokenMap

import com.twitter.hermit.model.Algorithm

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

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

import com.twitter.storehaus.ReadableStore

import com.twitter.timelines.configapi.Params

import com.twitter.util.Future

import javax.inject.Inject

import javax.inject.Named

import javax.inject.Singleton

/\*\*

\* TweetCandidateGenerator based on FRS seed users. For now this candidate generator fetches seed

\* users from FRS, and retrieves the seed users' past tweets from Earlybird with Earlybird light

\* ranking models.

\*/

@Singleton

class FrsTweetCandidateGenerator @Inject() (

@Named(ModuleNames.FrsStore) frsStore: ReadableStore[FrsStore.Query, Seq[FrsQueryResult]],

frsBasedSimilarityEngine: EarlybirdSimilarityEngineRouter,

tweetInfoStore: ReadableStore[TweetId, TweetInfo],

timeoutConfig: TimeoutConfig,

globalStats: StatsReceiver) {

import FrsTweetCandidateGenerator.\_

private val timer = DefaultTimer

private val stats: StatsReceiver = globalStats.scope(this.getClass.getCanonicalName)

private val fetchSeedsStats = stats.scope("fetchSeeds")

private val fetchCandidatesStats = stats.scope("fetchCandidates")

private val filterCandidatesStats = stats.scope("filterCandidates")

private val hydrateCandidatesStats = stats.scope("hydrateCandidates")

private val getCandidatesStats = stats.scope("getCandidates")

/\*\*

\* The function retrieves the candidate for the given user as follows:

\* 1. Seed user fetch from FRS.

\* 2. Candidate fetch from Earlybird.

\* 3. Filtering.

\* 4. Candidate hydration.

\* 5. Truncation.

\*/

def get(

frsTweetCandidateGeneratorQuery: FrsTweetCandidateGeneratorQuery

): Future[Seq[FrsTweet]] = {

val userId = frsTweetCandidateGeneratorQuery.userId

val product = frsTweetCandidateGeneratorQuery.product

val allStats = stats.scope("all")

val perProductStats = stats.scope("perProduct", product.name)

StatsUtil.trackItemsStats(allStats) {

StatsUtil.trackItemsStats(perProductStats) {

val result = for {

seedAuthorWithScores <- StatsUtil.trackOptionItemMapStats(fetchSeedsStats) {

fetchSeeds(

userId,

frsTweetCandidateGeneratorQuery.impressedUserList,

frsTweetCandidateGeneratorQuery.languageCodeOpt,

frsTweetCandidateGeneratorQuery.countryCodeOpt,

frsTweetCandidateGeneratorQuery.params,

)

}

tweetCandidates <- StatsUtil.trackOptionItemsStats(fetchCandidatesStats) {

fetchCandidates(

userId,

seedAuthorWithScores.map(\_.keys.toSeq).getOrElse(Seq.empty),

frsTweetCandidateGeneratorQuery.impressedTweetList,

seedAuthorWithScores.map(\_.mapValues(\_.score)).getOrElse(Map.empty),

frsTweetCandidateGeneratorQuery.params

)

}

filteredTweetCandidates <- StatsUtil.trackOptionItemsStats(filterCandidatesStats) {

filterCandidates(

tweetCandidates,

frsTweetCandidateGeneratorQuery.params

)

}

hydratedTweetCandidates <- StatsUtil.trackOptionItemsStats(hydrateCandidatesStats) {

hydrateCandidates(

seedAuthorWithScores,

filteredTweetCandidates

)

}

} yield {

hydratedTweetCandidates

.map(\_.take(frsTweetCandidateGeneratorQuery.maxNumResults)).getOrElse(Seq.empty)

}

result.raiseWithin(timeoutConfig.frsBasedTweetEndpointTimeout)(timer)

}

}

}

/\*\*

\* Fetch recommended seed users from FRS

\*/

private def fetchSeeds(

userId: UserId,

userDenyList: Set[UserId],

languageCodeOpt: Option[String],

countryCodeOpt: Option[String],

params: Params

): Future[Option[Map[UserId, FrsQueryResult]]] = {

frsStore

.get(

FrsStore.Query(

userId,

params(FrsParams.FrsBasedCandidateGenerationMaxSeedsNumParam),

params(FrsParams.FrsBasedCandidateGenerationDisplayLocationParam).displayLocation,

userDenyList.toSeq,

languageCodeOpt,

countryCodeOpt

)).map {

\_.map { seedAuthors =>

seedAuthors.map(user => user.userId -> user).toMap

}

}

}

/\*\*

\* Fetch tweet candidates from Earlybird

\*/

private def fetchCandidates(

searcherUserId: UserId,

seedAuthors: Seq[UserId],

impressedTweetList: Set[TweetId],

frsUserToScores: Map[UserId, Double],

params: Params

): Future[Option[Seq[TweetWithAuthor]]] = {

if (seedAuthors.nonEmpty) {

// call earlybird

val query = EarlybirdSimilarityEngineRouter.queryFromParams(

Some(searcherUserId),

seedAuthors,

impressedTweetList,

frsUserToScoresForScoreAdjustment = Some(frsUserToScores),

params

)

frsBasedSimilarityEngine.get(query)

} else Future.None

}

/\*\*

\* Filter candidates that do not pass visibility filter policy

\*/

private def filterCandidates(

candidates: Option[Seq[TweetWithAuthor]],

params: Params

): Future[Option[Seq[TweetWithAuthor]]] = {

val tweetIds = candidates.map(\_.map(\_.tweetId).toSet).getOrElse(Set.empty)

if (params(FrsParams.FrsBasedCandidateGenerationEnableVisibilityFilteringParam))

Future

.collect(tweetInfoStore.multiGet(tweetIds)).map { tweetInfos =>

candidates.map {

// If tweetInfo does not exist, we will filter out this tweet candidate.

\_.filter(candidate => tweetInfos.getOrElse(candidate.tweetId, None).isDefined)

}

}

else {

Future.value(candidates)

}

}

/\*\*

\* Hydrate the candidates with the FRS candidate sources and scores

\*/

private def hydrateCandidates(

frsAuthorWithScores: Option[Map[UserId, FrsQueryResult]],

candidates: Option[Seq[TweetWithAuthor]]

): Future[Option[Seq[FrsTweet]]] = {

Future.value {

candidates.map {

\_.map { tweetWithAuthor =>

val frsQueryResult = frsAuthorWithScores.flatMap(\_.get(tweetWithAuthor.authorId))

FrsTweet(

tweetId = tweetWithAuthor.tweetId,

authorId = tweetWithAuthor.authorId,

frsPrimarySource = frsQueryResult.flatMap(\_.primarySource),

frsAuthorScore = frsQueryResult.map(\_.score),

frsCandidateSourceScores = frsQueryResult.flatMap { result =>

result.sourceWithScores.map {

\_.collect {

// see TokenStrToAlgorithmMap @ https://sourcegraph.twitter.biz/git.twitter.biz/source/-/blob/hermit/hermit-core/src/main/scala/com/twitter/hermit/constants/AlgorithmFeedbackTokens.scala

// see Algorithm @ https://sourcegraph.twitter.biz/git.twitter.biz/source/-/blob/hermit/hermit-core/src/main/scala/com/twitter/hermit/model/Algorithm.scala

case (candidateSourceAlgoStr, score)

if AlgorithmFeedbackTokens.TokenStrToAlgorithmMap.contains(

candidateSourceAlgoStr) =>

AlgorithmToFeedbackTokenMap.getOrElse(

AlgorithmFeedbackTokens.TokenStrToAlgorithmMap

.getOrElse(candidateSourceAlgoStr, DefaultAlgo),

DefaultAlgoToken) -> score

}

}

}

)

}

}

}

}

}

object FrsTweetCandidateGenerator {

val DefaultAlgo: Algorithm.Value = Algorithm.Other

// 9999 is the token for Algorithm.Other

val DefaultAlgoToken: Int = AlgorithmToFeedbackTokenMap.getOrElse(DefaultAlgo, 9999)

}