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

import com.twitter.cr\_mixer.blender.SwitchBlender

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

import com.twitter.cr\_mixer.filter.PostRankFilterRunner

import com.twitter.cr\_mixer.filter.PreRankFilterRunner

import com.twitter.cr\_mixer.logging.CrMixerScribeLogger

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

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

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

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

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

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

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

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

import com.twitter.cr\_mixer.ranker.SwitchRanker

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

import com.twitter.cr\_mixer.source\_signal.UssStore.EnabledNegativeSourceTypes

import com.twitter.finagle.stats.StatsReceiver

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

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

import com.twitter.util.Future

import com.twitter.util.JavaTimer

import com.twitter.util.Timer

import javax.inject.Inject

import javax.inject.Singleton

/\*\*

\* For now it performs the main steps as follows:

\* 1. Source signal (via USS, FRS) fetch

\* 2. Candidate generation

\* 3. Filtering

\* 4. Interleave blender

\* 5. Ranker

\* 6. Post-ranker filter

\* 7. Truncation

\*/

@Singleton

class CrCandidateGenerator @Inject() (

sourceInfoRouter: SourceInfoRouter,

candidateSourceRouter: CandidateSourcesRouter,

switchBlender: SwitchBlender,

preRankFilterRunner: PreRankFilterRunner,

postRankFilterRunner: PostRankFilterRunner,

switchRanker: SwitchRanker,

crMixerScribeLogger: CrMixerScribeLogger,

timeoutConfig: TimeoutConfig,

globalStats: StatsReceiver) {

private val timer: Timer = new JavaTimer(true)

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

private val fetchSourcesStats = stats.scope("fetchSources")

private val fetchPositiveSourcesStats = stats.scope("fetchPositiveSources")

private val fetchNegativeSourcesStats = stats.scope("fetchNegativeSources")

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

private val fetchCandidatesAfterFilterStats = stats.scope("fetchCandidatesAfterFilter")

private val preRankFilterStats = stats.scope("preRankFilter")

private val interleaveStats = stats.scope("interleave")

private val rankStats = stats.scope("rank")

private val postRankFilterStats = stats.scope("postRankFilter")

private val blueVerifiedTweetStats = stats.scope("blueVerifiedTweetStats")

private val blueVerifiedTweetStatsPerSimilarityEngine =

stats.scope("blueVerifiedTweetStatsPerSimilarityEngine")

def get(query: CrCandidateGeneratorQuery): Future[Seq[RankedCandidate]] = {

val allStats = stats.scope("all")

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

val perProductBlueVerifiedStats =

blueVerifiedTweetStats.scope("perProduct", query.product.toString)

StatsUtil.trackItemsStats(allStats) {

trackResultStats(perProductStats) {

StatsUtil.trackItemsStats(perProductStats) {

val result = for {

(sourceSignals, sourceGraphsMap) <- StatsUtil.trackBlockStats(fetchSourcesStats) {

fetchSources(query)

}

initialCandidates <- StatsUtil.trackBlockStats(fetchCandidatesAfterFilterStats) {

// find the positive and negative signals

val (positiveSignals, negativeSignals) = sourceSignals.partition { signal =>

!EnabledNegativeSourceTypes.contains(signal.sourceType)

}

fetchPositiveSourcesStats.stat("size").add(positiveSignals.size)

fetchNegativeSourcesStats.stat("size").add(negativeSignals.size)

// find the positive signals to keep, removing block and muted users

val filteredSourceInfo =

if (negativeSignals.nonEmpty && query.params(

RecentNegativeSignalParams.EnableSourceParam)) {

filterSourceInfo(positiveSignals, negativeSignals)

} else {

positiveSignals

}

// fetch candidates from the positive signals

StatsUtil.trackBlockStats(fetchCandidatesStats) {

fetchCandidates(query, filteredSourceInfo, sourceGraphsMap)

}

}

filteredCandidates <- StatsUtil.trackBlockStats(preRankFilterStats) {

preRankFilter(query, initialCandidates)

}

interleavedCandidates <- StatsUtil.trackItemsStats(interleaveStats) {

interleave(query, filteredCandidates)

}

rankedCandidates <- StatsUtil.trackItemsStats(rankStats) {

val candidatesToRank =

interleavedCandidates.take(query.params(RankerParams.MaxCandidatesToRank))

rank(query, candidatesToRank)

}

postRankFilterCandidates <- StatsUtil.trackItemsStats(postRankFilterStats) {

postRankFilter(query, rankedCandidates)

}

} yield {

trackTopKStats(

800,

postRankFilterCandidates,

isQueryK = false,

perProductBlueVerifiedStats)

trackTopKStats(

400,

postRankFilterCandidates,

isQueryK = false,

perProductBlueVerifiedStats)

trackTopKStats(

query.maxNumResults,

postRankFilterCandidates,

isQueryK = true,

perProductBlueVerifiedStats)

val (blueVerifiedTweets, remainingTweets) =

postRankFilterCandidates.partition(

\_.tweetInfo.hasBlueVerifiedAnnotation.contains(true))

val topKBlueVerified = blueVerifiedTweets.take(query.maxNumResults)

val topKRemaining = remainingTweets.take(query.maxNumResults - topKBlueVerified.size)

trackBlueVerifiedTweetStats(topKBlueVerified, perProductBlueVerifiedStats)

if (topKBlueVerified.nonEmpty && query.params(RankerParams.EnableBlueVerifiedTopK)) {

topKBlueVerified ++ topKRemaining

} else {

postRankFilterCandidates

}

}

result.raiseWithin(timeoutConfig.serviceTimeout)(timer)

}

}

}

}

private def fetchSources(

query: CrCandidateGeneratorQuery

): Future[(Set[SourceInfo], Map[String, Option[GraphSourceInfo]])] = {

crMixerScribeLogger.scribeSignalSources(

query,

sourceInfoRouter

.get(query.userId, query.product, query.userState, query.params))

}

private def filterSourceInfo(

positiveSignals: Set[SourceInfo],

negativeSignals: Set[SourceInfo]

): Set[SourceInfo] = {

val filterUsers: Set[Long] = negativeSignals.flatMap {

case SourceInfo(\_, InternalId.UserId(userId), \_) => Some(userId)

case \_ => None

}

positiveSignals.filter {

case SourceInfo(\_, InternalId.UserId(userId), \_) => !filterUsers.contains(userId)

case \_ => true

}

}

def fetchCandidates(

query: CrCandidateGeneratorQuery,

sourceSignals: Set[SourceInfo],

sourceGraphs: Map[String, Option[GraphSourceInfo]]

): Future[Seq[Seq[InitialCandidate]]] = {

val initialCandidates = candidateSourceRouter

.fetchCandidates(

query.userId,

sourceSignals,

sourceGraphs,

query.params

)

initialCandidates.map(\_.flatten.map { candidate =>

if (candidate.tweetInfo.hasBlueVerifiedAnnotation.contains(true)) {

blueVerifiedTweetStatsPerSimilarityEngine

.scope(query.product.toString).scope(

candidate.candidateGenerationInfo.contributingSimilarityEngines.head.similarityEngineType.toString).counter(

candidate.tweetInfo.authorId.toString).incr()

}

})

crMixerScribeLogger.scribeInitialCandidates(

query,

initialCandidates

)

}

private def preRankFilter(

query: CrCandidateGeneratorQuery,

candidates: Seq[Seq[InitialCandidate]]

): Future[Seq[Seq[InitialCandidate]]] = {

crMixerScribeLogger.scribePreRankFilterCandidates(

query,

preRankFilterRunner

.runSequentialFilters(query, candidates))

}

private def postRankFilter(

query: CrCandidateGeneratorQuery,

candidates: Seq[RankedCandidate]

): Future[Seq[RankedCandidate]] = {

postRankFilterRunner.run(query, candidates)

}

private def interleave(

query: CrCandidateGeneratorQuery,

candidates: Seq[Seq[InitialCandidate]]

): Future[Seq[BlendedCandidate]] = {

crMixerScribeLogger.scribeInterleaveCandidates(

query,

switchBlender

.blend(query.params, query.userState, candidates))

}

private def rank(

query: CrCandidateGeneratorQuery,

candidates: Seq[BlendedCandidate],

): Future[Seq[RankedCandidate]] = {

crMixerScribeLogger.scribeRankedCandidates(

query,

switchRanker.rank(query, candidates)

)

}

private def trackResultStats(

stats: StatsReceiver

)(

fn: => Future[Seq[RankedCandidate]]

): Future[Seq[RankedCandidate]] = {

fn.onSuccess { candidates =>

trackReasonChosenSourceTypeStats(candidates, stats)

trackReasonChosenSimilarityEngineStats(candidates, stats)

trackPotentialReasonsSourceTypeStats(candidates, stats)

trackPotentialReasonsSimilarityEngineStats(candidates, stats)

}

}

private def trackReasonChosenSourceTypeStats(

candidates: Seq[RankedCandidate],

stats: StatsReceiver

): Unit = {

candidates

.groupBy(\_.reasonChosen.sourceInfoOpt.map(\_.sourceType))

.foreach {

case (sourceTypeOpt, rankedCands) =>

val sourceType = sourceTypeOpt.map(\_.toString).getOrElse("RequesterId") // default

stats.stat("reasonChosen", "sourceType", sourceType, "size").add(rankedCands.size)

}

}

private def trackReasonChosenSimilarityEngineStats(

candidates: Seq[RankedCandidate],

stats: StatsReceiver

): Unit = {

candidates

.groupBy(\_.reasonChosen.similarityEngineInfo.similarityEngineType)

.foreach {

case (seInfoType, rankedCands) =>

stats

.stat("reasonChosen", "similarityEngine", seInfoType.toString, "size").add(

rankedCands.size)

}

}

private def trackPotentialReasonsSourceTypeStats(

candidates: Seq[RankedCandidate],

stats: StatsReceiver

): Unit = {

candidates

.flatMap(\_.potentialReasons.map(\_.sourceInfoOpt.map(\_.sourceType)))

.groupBy(source => source)

.foreach {

case (sourceInfoOpt, seq) =>

val sourceType = sourceInfoOpt.map(\_.toString).getOrElse("RequesterId") // default

stats.stat("potentialReasons", "sourceType", sourceType, "size").add(seq.size)

}

}

private def trackPotentialReasonsSimilarityEngineStats(

candidates: Seq[RankedCandidate],

stats: StatsReceiver

): Unit = {

candidates

.flatMap(\_.potentialReasons.map(\_.similarityEngineInfo.similarityEngineType))

.groupBy(se => se)

.foreach {

case (seType, seq) =>

stats.stat("potentialReasons", "similarityEngine", seType.toString, "size").add(seq.size)

}

}

private def trackBlueVerifiedTweetStats(

candidates: Seq[RankedCandidate],

statsReceiver: StatsReceiver

): Unit = {

candidates.foreach { candidate =>

if (candidate.tweetInfo.hasBlueVerifiedAnnotation.contains(true)) {

statsReceiver.counter(candidate.tweetInfo.authorId.toString).incr()

statsReceiver

.scope(candidate.tweetInfo.authorId.toString).counter(candidate.tweetId.toString).incr()

}

}

}

private def trackTopKStats(

k: Int,

tweetCandidates: Seq[RankedCandidate],

isQueryK: Boolean,

statsReceiver: StatsReceiver

): Unit = {

val (topK, beyondK) = tweetCandidates.splitAt(k)

val blueVerifiedIds = tweetCandidates.collect {

case candidate if candidate.tweetInfo.hasBlueVerifiedAnnotation.contains(true) =>

candidate.tweetInfo.authorId

}.toSet

blueVerifiedIds.foreach { blueVerifiedId =>

val numTweetsTopK = topK.count(\_.tweetInfo.authorId == blueVerifiedId)

val numTweetsBeyondK = beyondK.count(\_.tweetInfo.authorId == blueVerifiedId)

if (isQueryK) {

statsReceiver.scope(blueVerifiedId.toString).stat(s"topK").add(numTweetsTopK)

statsReceiver

.scope(blueVerifiedId.toString).stat(s"beyondK").add(numTweetsBeyondK)

} else {

statsReceiver.scope(blueVerifiedId.toString).stat(s"top$k").add(numTweetsTopK)

statsReceiver

.scope(blueVerifiedId.toString).stat(s"beyond$k").add(numTweetsBeyondK)

}

}

}

}