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

import com.twitter.contentrecommender.thriftscala.TweetInfo

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

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

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

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

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

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

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

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

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

import com.twitter.finagle.stats.StatsReceiver

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

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

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

import com.twitter.storehaus.ReadableStore

import com.twitter.timelines.configapi

import com.twitter.util.Future

import javax.inject.Inject

import javax.inject.Named

import javax.inject.Singleton

@Singleton

class RelatedTweetCandidateGenerator @Inject() (

@Named(ModuleNames.TweetBasedUnifiedSimilarityEngine) tweetBasedUnifiedSimilarityEngine: StandardSimilarityEngine[

TweetBasedUnifiedSimilarityEngine.Query,

TweetWithCandidateGenerationInfo

],

@Named(ModuleNames.ProducerBasedUnifiedSimilarityEngine) producerBasedUnifiedSimilarityEngine: StandardSimilarityEngine[

ProducerBasedUnifiedSimilarityEngine.Query,

TweetWithCandidateGenerationInfo

],

preRankFilterRunner: PreRankFilterRunner,

relatedTweetScribeLogger: RelatedTweetScribeLogger,

tweetInfoStore: ReadableStore[TweetId, TweetInfo],

globalStats: StatsReceiver) {

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

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

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

def get(

query: RelatedTweetCandidateGeneratorQuery

): Future[Seq[InitialCandidate]] = {

val allStats = stats.scope("all")

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

StatsUtil.trackItemsStats(allStats) {

StatsUtil.trackItemsStats(perProductStats) {

for {

initialCandidates <- StatsUtil.trackBlockStats(fetchCandidatesStats) {

fetchCandidates(query)

}

filteredCandidates <- StatsUtil.trackBlockStats(preRankFilterStats) {

preRankFilter(query, initialCandidates)

}

} yield {

filteredCandidates.headOption

.getOrElse(

throw new UnsupportedOperationException(

"RelatedTweetCandidateGenerator results invalid")

).take(query.maxNumResults)

}

}

}

}

def fetchCandidates(

query: RelatedTweetCandidateGeneratorQuery

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

relatedTweetScribeLogger.scribeInitialCandidates(

query,

query.internalId match {

case InternalId.TweetId(\_) =>

getCandidatesFromSimilarityEngine(

query,

TweetBasedUnifiedSimilarityEngine.fromParamsForRelatedTweet,

tweetBasedUnifiedSimilarityEngine.getCandidates)

case InternalId.UserId(\_) =>

getCandidatesFromSimilarityEngine(

query,

ProducerBasedUnifiedSimilarityEngine.fromParamsForRelatedTweet,

producerBasedUnifiedSimilarityEngine.getCandidates)

case \_ =>

throw new UnsupportedOperationException(

"RelatedTweetCandidateGenerator gets invalid InternalId")

}

)

}

/\*\*\*

\* fetch Candidates from TweetBased/ProducerBased Unified Similarity Engine,

\* and apply VF filter based on TweetInfoStore

\* To align with the downstream processing (filter, rank), we tend to return a Seq[Seq[InitialCandidate]]

\* instead of a Seq[Candidate] even though we only have a Seq in it.

\*/

private def getCandidatesFromSimilarityEngine[QueryType](

query: RelatedTweetCandidateGeneratorQuery,

fromParamsForRelatedTweet: (InternalId, configapi.Params) => QueryType,

getFunc: QueryType => Future[Option[Seq[TweetWithCandidateGenerationInfo]]]

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

/\*\*\*

\* We wrap the query to be a Seq of queries for the Sim Engine to ensure evolvability of candidate generation

\* and as a result, it will return Seq[Seq[InitialCandidate]]

\*/

val engineQueries =

Seq(fromParamsForRelatedTweet(query.internalId, query.params))

Future

.collect {

engineQueries.map { query =>

for {

candidates <- getFunc(query)

prefilterCandidates <- convertToInitialCandidates(

candidates.toSeq.flatten

)

} yield prefilterCandidates

}

}

}

private def preRankFilter(

query: RelatedTweetCandidateGeneratorQuery,

candidates: Seq[Seq[InitialCandidate]]

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

relatedTweetScribeLogger.scribePreRankFilterCandidates(

query,

preRankFilterRunner

.runSequentialFilters(query, candidates))

}

private[candidate\_generation] def convertToInitialCandidates(

candidates: Seq[TweetWithCandidateGenerationInfo],

): Future[Seq[InitialCandidate]] = {

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

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

/\*\*\*

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

\* This tweetInfo filter also acts as the VF filter

\*/

candidates.collect {

case candidate if tweetInfos.getOrElse(candidate.tweetId, None).isDefined =>

val tweetInfo = tweetInfos(candidate.tweetId)

.getOrElse(throw new IllegalStateException("Check previous line's condition"))

InitialCandidate(

tweetId = candidate.tweetId,

tweetInfo = tweetInfo,

candidate.candidateGenerationInfo

)

}

}

}

}