package com.twitter.frigate.pushservice.refresh\_handler

import com.twitter.finagle.stats.Counter

import com.twitter.finagle.stats.Stat

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.base.Stats.track

import com.twitter.frigate.common.base.Stats.trackSeq

import com.twitter.frigate.common.base.\_

import com.twitter.frigate.common.logger.MRLogger

import com.twitter.frigate.pushservice.model.PushTypes.PushCandidate

import com.twitter.frigate.pushservice.model.PushTypes.RawCandidate

import com.twitter.frigate.pushservice.model.PushTypes.Target

import com.twitter.frigate.pushservice.adaptor.\_

import com.twitter.frigate.pushservice.params.PushFeatureSwitchParams

import com.twitter.frigate.pushservice.rank.RFPHLightRanker

import com.twitter.frigate.pushservice.rank.RFPHRanker

import com.twitter.frigate.pushservice.scriber.MrRequestScribeHandler

import com.twitter.frigate.pushservice.take.candidate\_validator.RFPHCandidateValidator

import com.twitter.frigate.pushservice.target.PushTargetUserBuilder

import com.twitter.frigate.pushservice.target.RFPHTargetPredicates

import com.twitter.frigate.pushservice.util.RFPHTakeStepUtil

import com.twitter.frigate.pushservice.util.AdhocStatsUtil

import com.twitter.frigate.pushservice.thriftscala.PushContext

import com.twitter.frigate.pushservice.thriftscala.RefreshRequest

import com.twitter.frigate.pushservice.thriftscala.RefreshResponse

import com.twitter.frigate.thriftscala.CommonRecommendationType

import com.twitter.hermit.predicate.Predicate

import com.twitter.timelines.configapi.FeatureValue

import com.twitter.util.\_

case class ResultWithDebugInfo(result: Result, predicateResults: Seq[PredicateWithResult])

class RefreshForPushHandler(

val pushTargetUserBuilder: PushTargetUserBuilder,

val candSourceGenerator: PushCandidateSourceGenerator,

rfphRanker: RFPHRanker,

candidateHydrator: PushCandidateHydrator,

candidateValidator: RFPHCandidateValidator,

rfphTakeStepUtil: RFPHTakeStepUtil,

rfphRestrictStep: RFPHRestrictStep,

val rfphNotifier: RefreshForPushNotifier,

rfphStatsRecorder: RFPHStatsRecorder,

mrRequestScriberNode: String,

rfphFeatureHydrator: RFPHFeatureHydrator,

rfphPrerankFilter: RFPHPrerankFilter,

rfphLightRanker: RFPHLightRanker

)(

globalStats: StatsReceiver)

extends FetchRankFlowWithHydratedCandidates[Target, RawCandidate, PushCandidate] {

val log = MRLogger("RefreshForPushHandler")

implicit val statsReceiver: StatsReceiver =

globalStats.scope("RefreshForPushHandler")

private val maxCandidatesToBatchInTakeStat: Stat =

statsReceiver.stat("max\_cands\_to\_batch\_in\_take")

private val rfphRequestCounter = statsReceiver.counter("requests")

private val buildTargetStats = statsReceiver.scope("build\_target")

private val processStats = statsReceiver.scope("process")

private val notifyStats = statsReceiver.scope("notify")

private val lightRankingStats: StatsReceiver = statsReceiver.scope("light\_ranking")

private val reRankingStats: StatsReceiver = statsReceiver.scope("rerank")

private val featureHydrationLatency: StatsReceiver =

statsReceiver.scope("featureHydrationLatency")

private val candidateHydrationStats: StatsReceiver = statsReceiver.scope("candidate\_hydration")

lazy val candSourceEligibleCounter: Counter =

candidateStats.counter("cand\_source\_eligible")

lazy val candSourceNotEligibleCounter: Counter =

candidateStats.counter("cand\_source\_not\_eligible")

//pre-ranking stats

val allCandidatesFilteredPreRank = filterStats.counter("all\_candidates\_filtered")

// total invalid candidates

val totalStats: StatsReceiver = statsReceiver.scope("total")

val totalInvalidCandidatesStat: Stat = totalStats.stat("candidates\_invalid")

val mrRequestScribeBuiltStats: Counter = statsReceiver.counter("mr\_request\_scribe\_built")

val mrRequestCandidateScribeStats = statsReceiver.scope("mr\_request\_scribe\_candidates")

val mrRequestTargetScribeStats = statsReceiver.scope("mr\_request\_scribe\_target")

val mrRequestScribeHandler =

new MrRequestScribeHandler(mrRequestScriberNode, statsReceiver.scope("mr\_request\_scribe"))

val adhocStatsUtil = new AdhocStatsUtil(statsReceiver.scope("adhoc\_stats"))

private def numRecsPerTypeStat(crt: CommonRecommendationType) =

fetchStats.scope(crt.toString).stat("dist")

// static list of target predicates

private val targetPredicates = RFPHTargetPredicates(targetStats.scope("predicates"))

def buildTarget(

userId: Long,

inputPushContext: Option[PushContext],

forcedFeatureValues: Option[Map[String, FeatureValue]] = None

): Future[Target] =

pushTargetUserBuilder.buildTarget(userId, inputPushContext, forcedFeatureValues)

override def targetPredicates(target: Target): List[Predicate[Target]] = targetPredicates

override def isTargetValid(target: Target): Future[Result] = {

val resultFut = if (target.skipFilters) {

Future.value(trackTargetPredStats(None))

} else {

predicateSeq(target).track(Seq(target)).map { resultArr =>

trackTargetPredStats(resultArr(0))

}

}

track(targetStats)(resultFut)

}

override def candidateSources(

target: Target

): Future[Seq[CandidateSource[Target, RawCandidate]]] = {

Future

.collect(candSourceGenerator.sources.map { cs =>

cs.isCandidateSourceAvailable(target).map { isEligible =>

if (isEligible) {

candSourceEligibleCounter.incr()

Some(cs)

} else {

candSourceNotEligibleCounter.incr()

None

}

}

}).map(\_.flatten)

}

override def updateCandidateCounter(

candidateResults: Seq[CandidateResult[PushCandidate, Result]]

): Unit = {

candidateResults.foreach {

case candidateResult if candidateResult.result == OK =>

okCandidateCounter.incr()

case candidateResult if candidateResult.result.isInstanceOf[Invalid] =>

invalidCandidateCounter.incr()

case \_ =>

}

}

override def hydrateCandidates(

candidates: Seq[CandidateDetails[RawCandidate]]

): Future[Seq[CandidateDetails[PushCandidate]]] = candidateHydrator(candidates)

override def filter(

target: Target,

hydratedCandidates: Seq[CandidateDetails[PushCandidate]]

): Future[

(Seq[CandidateDetails[PushCandidate]], Seq[CandidateResult[PushCandidate, Result]])

] = rfphPrerankFilter.filter(target, hydratedCandidates)

def lightRankAndTake(

target: Target,

candidates: Seq[CandidateDetails[PushCandidate]]

): Future[Seq[CandidateDetails[PushCandidate]]] = {

rfphLightRanker.rank(target, candidates)

}

override def rank(

target: Target,

candidatesDetails: Seq[CandidateDetails[PushCandidate]]

): Future[Seq[CandidateDetails[PushCandidate]]] = {

val featureHydratedCandidatesFut = trackSeq(featureHydrationLatency)(

rfphFeatureHydrator

.candidateFeatureHydration(candidatesDetails, target.mrRequestContextForFeatureStore)

)

featureHydratedCandidatesFut.flatMap { featureHydratedCandidates =>

rfphStatsRecorder.rankDistributionStats(featureHydratedCandidates, numRecsPerTypeStat)

rfphRanker.initialRank(target, candidatesDetails)

}

}

def reRank(

target: Target,

rankedCandidates: Seq[CandidateDetails[PushCandidate]]

): Future[Seq[CandidateDetails[PushCandidate]]] = {

rfphRanker.reRank(target, rankedCandidates)

}

override def validCandidates(

target: Target,

candidates: Seq[PushCandidate]

): Future[Seq[Result]] = {

Future.collect(candidates.map { candidate =>

rfphTakeStepUtil.isCandidateValid(candidate, candidateValidator).map(res => res.result)

})

}

override def desiredCandidateCount(target: Target): Int = target.desiredCandidateCount

override def batchForCandidatesCheck(target: Target): Int = {

val fsParam = PushFeatureSwitchParams.NumberOfMaxCandidatesToBatchInRFPHTakeStep

val maxToBatch = target.params(fsParam)

maxCandidatesToBatchInTakeStat.add(maxToBatch)

maxToBatch

}

override def process(

target: Target,

externalCandidates: Seq[RawCandidate] = Nil

): Future[Response[PushCandidate, Result]] = {

isTargetValid(target).flatMap {

case OK =>

for {

candidatesFromSources <- trackSeq(fetchStats)(fetchCandidates(target))

externalCandidateDetails = externalCandidates.map(

CandidateDetails(\_, "refresh\_for\_push\_handler\_external\_candidate"))

allCandidates = candidatesFromSources ++ externalCandidateDetails

hydratedCandidatesWithCopy <-

trackSeq(candidateHydrationStats)(hydrateCandidates(allCandidates))

\_ = adhocStatsUtil.getCandidateSourceStats(hydratedCandidatesWithCopy)

(candidates, preRankingFilteredCandidates) <-

track(filterStats)(filter(target, hydratedCandidatesWithCopy))

\_ = adhocStatsUtil.getPreRankingFilterStats(preRankingFilteredCandidates)

lightRankerFilteredCandidates <-

trackSeq(lightRankingStats)(lightRankAndTake(target, candidates))

\_ = adhocStatsUtil.getLightRankingStats(lightRankerFilteredCandidates)

rankedCandidates <- trackSeq(rankingStats)(rank(target, lightRankerFilteredCandidates))

\_ = adhocStatsUtil.getRankingStats(rankedCandidates)

rerankedCandidates <- trackSeq(reRankingStats)(reRank(target, rankedCandidates))

\_ = adhocStatsUtil.getReRankingStats(rerankedCandidates)

(restrictedCandidates, restrictFilteredCandidates) =

rfphRestrictStep.restrict(target, rerankedCandidates)

allTakeCandidateResults <- track(takeStats)(

take(target, restrictedCandidates, desiredCandidateCount(target))

)

\_ = adhocStatsUtil.getTakeCandidateResultStats(allTakeCandidateResults)

\_ <- track(mrRequestCandidateScribeStats)(

mrRequestScribeHandler.scribeForCandidateFiltering(

target,

hydratedCandidatesWithCopy,

preRankingFilteredCandidates,

rankedCandidates,

rerankedCandidates,

restrictFilteredCandidates,

allTakeCandidateResults

))

} yield {

/\*\*

\* Take processes post restrict step candidates and returns both:

\* 1. valid + invalid candidates

\* 2. Candidates that are not processed (more than desired) + restricted candidates

\* We need #2 only for importance sampling

\*/

val takeCandidateResults =

allTakeCandidateResults.filterNot { candResult =>

candResult.result == MoreThanDesiredCandidates

}

val totalInvalidCandidates = {

preRankingFilteredCandidates.size + //pre-ranking filtered candidates

(rerankedCandidates.length - restrictedCandidates.length) + //candidates reject in restrict step

takeCandidateResults.count(\_.result != OK) //candidates reject in take step

}

takeInvalidCandidateDist.add(

takeCandidateResults

.count(\_.result != OK)

) // take step invalid candidates

totalInvalidCandidatesStat.add(totalInvalidCandidates)

val allCandidateResults = takeCandidateResults ++ preRankingFilteredCandidates

Response(OK, allCandidateResults)

}

case result: Result =>

for (\_ <- track(mrRequestTargetScribeStats)(

mrRequestScribeHandler.scribeForTargetFiltering(target, result))) yield {

mrRequestScribeBuiltStats.incr()

Response(result, Nil)

}

}

}

def refreshAndSend(request: RefreshRequest): Future[RefreshResponse] = {

rfphRequestCounter.incr()

for {

target <- track(buildTargetStats)(

pushTargetUserBuilder

.buildTarget(request.userId, request.context))

response <- track(processStats)(process(target, externalCandidates = Seq.empty))

refreshResponse <- track(notifyStats)(rfphNotifier.checkResponseAndNotify(response, target))

} yield {

refreshResponse

}

}

}