package com.twitter.frigate.pushservice.predicate.magic\_fanout

import com.twitter.audience\_rewards.thriftscala.HasSuperFollowingRelationshipRequest

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.base.MagicFanoutCandidate

import com.twitter.frigate.common.base.MagicFanoutCreatorEventCandidate

import com.twitter.frigate.common.base.MagicFanoutProductLaunchCandidate

import com.twitter.frigate.common.history.RecItems

import com.twitter.frigate.common.predicate.FatiguePredicate.build

import com.twitter.frigate.common.predicate.FatiguePredicate.productLaunchTypeRecTypesOnlyFilter

import com.twitter.frigate.common.predicate.FatiguePredicate.recOnlyFilter

import com.twitter.frigate.common.store.interests.InterestsLookupRequestWithContext

import com.twitter.frigate.common.store.interests.SemanticCoreEntityId

import com.twitter.frigate.common.util.IbisAppPushDeviceSettingsUtil

import com.twitter.frigate.magic\_events.thriftscala.CreatorFanoutType

import com.twitter.frigate.magic\_events.thriftscala.ProductType

import com.twitter.frigate.magic\_events.thriftscala.TargetID

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

import com.twitter.frigate.pushservice.model.MagicFanoutEventHydratedCandidate

import com.twitter.frigate.pushservice.model.MagicFanoutEventPushCandidate

import com.twitter.frigate.pushservice.model.MagicFanoutNewsEventPushCandidate

import com.twitter.frigate.pushservice.config.Config

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

import com.twitter.frigate.pushservice.predicate.FatiguePredicate

import com.twitter.frigate.pushservice.predicate.PredicatesForCandidate

import com.twitter.frigate.thriftscala.CommonRecommendationType

import com.twitter.frigate.thriftscala.NotificationDisplayLocation

import com.twitter.gizmoduck.thriftscala.User

import com.twitter.hermit.predicate.NamedPredicate

import com.twitter.hermit.predicate.Predicate

import com.twitter.interests.thriftscala.UserInterests

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

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

import com.twitter.storehaus.ReadableStore

import com.twitter.timelines.configapi.Param

import com.twitter.util.Duration

import com.twitter.util.Future

object MagicFanoutPredicatesForCandidate {

/\*\*

\* Check if Semantic Core reasons satisfy rank threshold ( for heavy users a non broad entity should satisfy the threshold)

\*/

def magicFanoutErgInterestRankThresholdPredicate(

implicit stats: StatsReceiver

): NamedPredicate[MagicFanoutEventHydratedCandidate] = {

val name = "magicfanout\_interest\_erg\_rank\_threshold"

val scopedStatsReceiver = stats.scope(s"predicate\_$name")

Predicate

.fromAsync { candidate: MagicFanoutEventHydratedCandidate =>

candidate.target.isHeavyUserState.map { isHeavyUser =>

lazy val rankThreshold =

if (isHeavyUser) {

candidate.target.params(PushFeatureSwitchParams.MagicFanoutRankErgThresholdHeavy)

} else {

candidate.target.params(PushFeatureSwitchParams.MagicFanoutRankErgThresholdNonHeavy)

}

MagicFanoutPredicatesUtil

.checkIfValidErgScEntityReasonExists(

candidate.effectiveMagicEventsReasons,

rankThreshold

)

}

}

.withStats(scopedStatsReceiver)

.withName(name)

}

def newsNotificationFatigue(

)(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate] = {

val name = "news\_notification\_fatigue"

val scopedStatsReceiver = stats.scope(s"predicate\_$name")

Predicate

.fromAsync { candidate: PushCandidate =>

FatiguePredicate

.recTypeSetOnly(

notificationDisplayLocation = NotificationDisplayLocation.PushToMobileDevice,

recTypes = Set(CommonRecommendationType.MagicFanoutNewsEvent),

maxInInterval =

candidate.target.params(PushFeatureSwitchParams.MFMaxNumberOfPushesInInterval),

interval = candidate.target.params(PushFeatureSwitchParams.MFPushIntervalInHours),

minInterval = candidate.target.params(PushFeatureSwitchParams.MFMinIntervalFatigue)

)

.apply(Seq(candidate))

.map(\_.headOption.getOrElse(false))

}

.withStats(scopedStatsReceiver)

.withName(name)

}

/\*\*

\* Check if reason contains any optouted semantic core entity interests.

\*

\* @param stats

\*

\* @return

\*/

def magicFanoutNoOptoutInterestPredicate(

implicit stats: StatsReceiver

): NamedPredicate[MagicFanoutEventPushCandidate] = {

val name = "magicfanout\_optout\_interest\_predicate"

val scopedStatsReceiver = stats.scope(s"predicate\_$name")

val withOptOutInterestsCounter = stats.counter("with\_optout\_interests")

val withoutOptOutInterestsCounter = stats.counter("without\_optout\_interests")

Predicate

.fromAsync { candidate: MagicFanoutEventPushCandidate =>

candidate.target.optOutSemanticCoreInterests.map {

case (

optOutUserInterests: Seq[SemanticCoreEntityId]

) =>

withOptOutInterestsCounter.incr()

optOutUserInterests

.intersect(candidate.annotatedAndInferredSemanticCoreEntities).isEmpty

case \_ =>

withoutOptOutInterestsCounter.incr()

true

}

}

.withStats(scopedStatsReceiver)

.withName(name)

}

/\*\*

\* Checks if the target has only one device language language,

\* and that language is targeted for that event

\*

\* @param statsReceiver

\*

\* @return

\*/

def inferredUserDeviceLanguagePredicate(

implicit statsReceiver: StatsReceiver

): NamedPredicate[MagicFanoutEventPushCandidate] = {

val name = "inferred\_device\_language"

val scopedStats = statsReceiver.scope(s"predicate\_$name")

Predicate

.fromAsync { candidate: MagicFanoutEventPushCandidate =>

val target = candidate.target

target.deviceInfo.map {

\_.flatMap { deviceInfo =>

val languages = deviceInfo.deviceLanguages.getOrElse(Seq.empty[String])

val distinctDeviceLanguages =

IbisAppPushDeviceSettingsUtil.distinctDeviceLanguages(languages)

candidate.newsForYouMetadata.map { newsForYouMetadata =>

val eventLocales = newsForYouMetadata.locales.getOrElse(Seq.empty)

val eventLanguages = eventLocales.flatMap(\_.language).map(\_.toLowerCase).distinct

eventLanguages.intersect(distinctDeviceLanguages).nonEmpty

}

}.getOrElse(false)

}

}

.withStats(scopedStats)

.withName(name)

}

/\*\*

\* Bypass predicate if high priority push

\*/

def highPriorityNewsEventExceptedPredicate(

predicate: NamedPredicate[MagicFanoutNewsEventPushCandidate]

)(

implicit config: Config

): NamedPredicate[MagicFanoutNewsEventPushCandidate] = {

PredicatesForCandidate.exceptedPredicate(

name = "high\_priority\_excepted\_" + predicate.name,

fn = MagicFanoutPredicatesUtil.checkIfHighPriorityNewsEventForCandidate,

predicate

)(config.statsReceiver)

}

/\*\*

\* Bypass predicate if high priority push

\*/

def highPriorityEventExceptedPredicate(

predicate: NamedPredicate[MagicFanoutEventPushCandidate]

)(

implicit config: Config

): NamedPredicate[MagicFanoutEventPushCandidate] = {

PredicatesForCandidate.exceptedPredicate(

name = "high\_priority\_excepted\_" + predicate.name,

fn = MagicFanoutPredicatesUtil.checkIfHighPriorityEventForCandidate,

predicate

)(config.statsReceiver)

}

def magicFanoutSimClusterTargetingPredicate(

implicit stats: StatsReceiver

): NamedPredicate[MagicFanoutEventPushCandidate] = {

val name = "simcluster\_targeting"

val scopedStats = stats.scope(s"predicate\_$name")

val userStateCounters = scopedStats.scope("user\_state")

Predicate

.fromAsync { candidate: MagicFanoutEventPushCandidate =>

candidate.target.isHeavyUserState.map { isHeavyUser =>

val simClusterEmbeddings = candidate.newsForYouMetadata.flatMap(

\_.eventContextScribe.flatMap(\_.simClustersEmbeddings))

val TopKSimClustersCount = 50

val eventSimClusterVectorOpt: Option[MagicFanoutPredicatesUtil.SimClusterScores] =

MagicFanoutPredicatesUtil.getEventSimClusterVector(

simClusterEmbeddings.map(\_.toMap),

(ModelVersion.Model20m145kUpdated, EmbeddingType.FollowBasedTweet),

TopKSimClustersCount

)

val userSimClusterVectorOpt: Option[MagicFanoutPredicatesUtil.SimClusterScores] =

MagicFanoutPredicatesUtil.getUserSimClusterVector(candidate.effectiveMagicEventsReasons)

(eventSimClusterVectorOpt, userSimClusterVectorOpt) match {

case (

Some(eventSimClusterVector: MagicFanoutPredicatesUtil.SimClusterScores),

Some(userSimClusterVector)) =>

val score = eventSimClusterVector

.normedDotProduct(userSimClusterVector, eventSimClusterVector)

val threshold = if (isHeavyUser) {

candidate.target.params(

PushFeatureSwitchParams.MagicFanoutSimClusterDotProductHeavyUserThreshold)

} else {

candidate.target.params(

PushFeatureSwitchParams.MagicFanoutSimClusterDotProductNonHeavyUserThreshold)

}

val isPassed = score >= threshold

userStateCounters.scope(isHeavyUser.toString).counter(s"$isPassed").incr()

isPassed

case (None, Some(userSimClusterVector)) =>

candidate.commonRecType == CommonRecommendationType.MagicFanoutSportsEvent

case \_ => false

}

}

}

.withStats(scopedStats)

.withName(name)

}

def geoTargetingHoldback(

)(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate with MagicFanoutCandidate] = {

Predicate

.from[PushCandidate with MagicFanoutCandidate] { candidate =>

if (MagicFanoutPredicatesUtil.reasonsContainGeoTarget(

candidate.candidateMagicEventsReasons)) {

candidate.target.params(PushFeatureSwitchParams.EnableMfGeoTargeting)

} else true

}

.withStats(stats.scope("geo\_targeting\_holdback"))

.withName("geo\_targeting\_holdback")

}

def geoOptOutPredicate(

userStore: ReadableStore[Long, User]

)(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate with MagicFanoutCandidate] = {

Predicate

.fromAsync[PushCandidate with MagicFanoutCandidate] { candidate =>

if (MagicFanoutPredicatesUtil.reasonsContainGeoTarget(

candidate.candidateMagicEventsReasons)) {

userStore.get(candidate.target.targetId).map { userOpt =>

val isGeoAllowed = userOpt

.flatMap(\_.account)

.exists(\_.allowLocationHistoryPersonalization)

isGeoAllowed

}

} else {

Future.True

}

}

.withStats(stats.scope("geo\_opt\_out\_predicate"))

.withName("geo\_opt\_out\_predicate")

}

/\*\*

\* Check if Semantic Core reasons contains valid utt reason & reason is within top k topics followed by user

\*/

def magicFanoutTopicFollowsTargetingPredicate(

implicit stats: StatsReceiver,

interestsLookupStore: ReadableStore[InterestsLookupRequestWithContext, UserInterests]

): NamedPredicate[MagicFanoutEventHydratedCandidate] = {

val name = "magicfanout\_topic\_follows\_targeting"

val scopedStatsReceiver = stats.scope(s"predicate\_$name")

Predicate

.fromAsync[PushCandidate with MagicFanoutEventHydratedCandidate] { candidate =>

candidate.followedTopicLocalizedEntities.map(\_.nonEmpty)

}

.withStats(scopedStatsReceiver)

.withName(name)

}

/\*\* Requires the magicfanout candidate to have a UserID reason which ranks below the follow

\* rank threshold. If no UserID target exists the candidate is dropped. \*/

def followRankThreshold(

threshold: Param[Int]

)(

implicit statsReceiver: StatsReceiver

): NamedPredicate[PushCandidate with MagicFanoutCandidate] = {

val name = "follow\_rank\_threshold"

Predicate

.from[PushCandidate with MagicFanoutCandidate] { c =>

c.candidateMagicEventsReasons.exists { fanoutReason =>

fanoutReason.reason match {

case TargetID.UserID(\_) =>

fanoutReason.rank.exists { rank =>

rank <= c.target.params(threshold)

}

case \_ => false

}

}

}

.withStats(statsReceiver.scope(name))

.withName(name)

}

def userGeneratedEventsPredicate(

implicit statsReceiver: StatsReceiver

): NamedPredicate[PushCandidate with MagicFanoutEventHydratedCandidate] = {

val name = "user\_generated\_moments"

val stats = statsReceiver.scope(name)

Predicate

.from { candidate: PushCandidate with MagicFanoutEventHydratedCandidate =>

val isUgmMoment = candidate.semanticCoreEntityTags.values.flatten.toSet

.contains(MagicFanoutPredicatesUtil.UgmMomentTag)

if (isUgmMoment) {

candidate.target.params(PushFeatureSwitchParams.MagicFanoutNewsUserGeneratedEventsEnable)

} else true

}.withStats(stats)

.withName(name)

}

def escherbirdMagicfanoutEventParam(

)(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate with MagicFanoutEventPushCandidate] = {

val name = "magicfanout\_escherbird\_fs"

val scopedStatsReceiver = stats.scope(s"predicate\_$name")

Predicate

.fromAsync[PushCandidate with MagicFanoutEventPushCandidate] { candidate =>

val candidateFrigateNotif = candidate.frigateNotification.magicFanoutEventNotification

val isEscherbirdEvent = candidateFrigateNotif.exists(\_.isEscherbirdEvent.contains(true))

scopedStatsReceiver.counter(s"with\_escherbird\_flag\_$isEscherbirdEvent").incr()

if (isEscherbirdEvent) {

val listOfEventsSemanticCoreDomainIds =

candidate.target.params(PushFeatureSwitchParams.ListOfEventSemanticCoreDomainIds)

val candScDomainEvent =

if (listOfEventsSemanticCoreDomainIds.nonEmpty) {

candidate.eventSemanticCoreDomainIds

.intersect(listOfEventsSemanticCoreDomainIds).nonEmpty

} else {

false

}

scopedStatsReceiver

.counter(

s"with\_escherbird\_fs\_in\_list\_of\_event\_semantic\_core\_domains\_$candScDomainEvent").incr()

Future.value(candScDomainEvent)

} else {

Future.True

}

}

.withStats(scopedStatsReceiver)

.withName(name)

}

/\*\*

\* Checks if the user has custom targeting enabled.If so, bucket the user in experiment. This custom targeting refers to adding

\* tweet authors as targets in the eventfanout service.

\* @param stats [StatsReceiver]

\* @return NamedPredicate[PushCandidate with MagicFanoutEventPushCandidate]

\*/

def hasCustomTargetingForNewsEventsParam(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate with MagicFanoutEventPushCandidate] = {

val name = "magicfanout\_hascustomtargeting"

val scopedStatsReceiver = stats.scope(s"predicate\_$name")

Predicate

.from[PushCandidate with MagicFanoutEventPushCandidate] { candidate =>

candidate.candidateMagicEventsReasons.exists { fanoutReason =>

fanoutReason.reason match {

case userIdReason: TargetID.UserID =>

if (userIdReason.userID.hasCustomTargeting.contains(true)) {

candidate.target.params(

PushFeatureSwitchParams.MagicFanoutEnableCustomTargetingNewsEvent)

} else true

case \_ => true

}

}

}

.withStats(scopedStatsReceiver)

.withName(name)

}

def magicFanoutProductLaunchFatigue(

)(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate with MagicFanoutProductLaunchCandidate] = {

val name = "magic\_fanout\_product\_launch\_fatigue"

val scopedStatsReceiver = stats.scope(s"predicate\_$name")

Predicate

.fromAsync { candidate: PushCandidate with MagicFanoutProductLaunchCandidate =>

val target = candidate.target

val (interval, maxInInterval, minInterval) = {

candidate.productLaunchType match {

case ProductType.BlueVerified =>

(

target.params(PushFeatureSwitchParams.ProductLaunchPushIntervalInHours),

target.params(PushFeatureSwitchParams.ProductLaunchMaxNumberOfPushesInInterval),

target.params(PushFeatureSwitchParams.ProductLaunchMinIntervalFatigue))

case \_ =>

(Duration.fromDays(1), 0, Duration.Zero)

}

}

build(

interval = interval,

maxInInterval = maxInInterval,

minInterval = minInterval,

filterHistory = productLaunchTypeRecTypesOnlyFilter(

Set(CommonRecommendationType.MagicFanoutProductLaunch),

candidate.productLaunchType.toString),

notificationDisplayLocation = NotificationDisplayLocation.PushToMobileDevice

).flatContraMap { candidate: PushCandidate => candidate.target.history }

.apply(Seq(candidate))

.map(\_.headOption.getOrElse(false))

}

.withStats(scopedStatsReceiver)

.withName(name)

}

def creatorPushTargetIsNotCreator(

)(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate with MagicFanoutCreatorEventCandidate] = {

val name = "magic\_fanout\_creator\_is\_self"

val scopedStatsReceiver = stats.scope(s"predicate\_$name")

Predicate

.from { candidate: PushCandidate with MagicFanoutCreatorEventCandidate =>

candidate.target.targetId != candidate.creatorId

}

.withStats(scopedStatsReceiver)

.withName(name)

}

def duplicateCreatorPredicate(

)(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate with MagicFanoutCreatorEventCandidate] = {

val name = "magic\_fanout\_creator\_duplicate\_creator\_id"

val scopedStatsReceiver = stats.scope(s"predicate\_$name")

Predicate

.fromAsync { cand: PushCandidate with MagicFanoutCreatorEventCandidate =>

cand.target.pushRecItems.map { recItems: RecItems =>

!recItems.creatorIds.contains(cand.creatorId)

}

}

.withStats(scopedStatsReceiver)

.withName(name)

}

def isSuperFollowingCreator(

)(

implicit config: Config,

stats: StatsReceiver

): NamedPredicate[PushCandidate with MagicFanoutCreatorEventCandidate] = {

val name = "magic\_fanout\_is\_already\_superfollowing\_creator"

val scopedStatsReceiver = stats.scope(s"predicate\_$name")

Predicate

.fromAsync { cand: PushCandidate with MagicFanoutCreatorEventCandidate =>

config.hasSuperFollowingRelationshipStore

.get(

HasSuperFollowingRelationshipRequest(

sourceUserId = cand.target.targetId,

targetUserId = cand.creatorId)).map(\_.getOrElse(false))

}

.withStats(scopedStatsReceiver)

.withName(name)

}

def magicFanoutCreatorPushFatiguePredicate(

)(

implicit stats: StatsReceiver

): NamedPredicate[PushCandidate with MagicFanoutCreatorEventCandidate] = {

val name = "magic\_fanout\_creator\_fatigue"

val scopedStatsReceiver = stats.scope(s"predicate\_$name")

Predicate

.fromAsync { candidate: PushCandidate with MagicFanoutCreatorEventCandidate =>

val target = candidate.target

val (interval, maxInInterval, minInterval) = {

candidate.creatorFanoutType match {

case CreatorFanoutType.UserSubscription =>

(

target.params(PushFeatureSwitchParams.CreatorSubscriptionPushIntervalInHours),

target.params(

PushFeatureSwitchParams.CreatorSubscriptionPushMaxNumberOfPushesInInterval),

target.params(PushFeatureSwitchParams.CreatorSubscriptionPushhMinIntervalFatigue))

case CreatorFanoutType.NewCreator =>

(

target.params(PushFeatureSwitchParams.NewCreatorPushIntervalInHours),

target.params(PushFeatureSwitchParams.NewCreatorPushMaxNumberOfPushesInInterval),

target.params(PushFeatureSwitchParams.NewCreatorPushMinIntervalFatigue))

case \_ =>

(Duration.fromDays(1), 0, Duration.Zero)

}

}

build(

interval = interval,

maxInInterval = maxInInterval,

minInterval = minInterval,

filterHistory = recOnlyFilter(candidate.commonRecType),

notificationDisplayLocation = NotificationDisplayLocation.PushToMobileDevice

).flatContraMap { candidate: PushCandidate => candidate.target.history }

.apply(Seq(candidate))

.map(\_.headOption.getOrElse(false))

}

.withStats(scopedStatsReceiver)

.withName(name)

}

}