package com.twitter.timelineranker.recap\_author

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.servo.util.FutureArrow

import com.twitter.storehaus.Store

import com.twitter.timelineranker.common.\_

import com.twitter.timelineranker.core.CandidateEnvelope

import com.twitter.timelineranker.core.HydratedCandidatesAndFeaturesEnvelope

import com.twitter.timelineranker.model.RecapQuery.DependencyProvider

import com.twitter.timelineranker.model.\_

import com.twitter.timelineranker.monitoring.UsersSearchResultMonitoringTransform

import com.twitter.timelineranker.parameters.monitoring.MonitoringParams

import com.twitter.timelineranker.parameters.recap.RecapParams

import com.twitter.timelineranker.parameters.recap\_author.RecapAuthorParams

import com.twitter.timelineranker.recap.model.ContentFeatures

import com.twitter.timelineranker.util.CopyContentFeaturesIntoHydratedTweetsTransform

import com.twitter.timelineranker.util.CopyContentFeaturesIntoThriftTweetFeaturesTransform

import com.twitter.timelineranker.util.TweetFilters

import com.twitter.timelineranker.visibility.FollowGraphDataProvider

import com.twitter.timelines.clients.gizmoduck.GizmoduckClient

import com.twitter.timelines.clients.manhattan.UserMetadataClient

import com.twitter.timelines.clients.relevance\_search.SearchClient

import com.twitter.timelines.clients.tweetypie.TweetyPieClient

import com.twitter.timelines.model.TweetId

import com.twitter.timelines.util.FailOpenHandler

import com.twitter.timelines.util.stats.RequestStatsReceiver

import com.twitter.timelines.visibility.VisibilityEnforcer

import com.twitter.util.Future

/\*\*

\* This source controls what tweets are fetched from earlybird given a

\* list of authors to fetch tweets from. The controls available are:

\* 1. The ''filters'' val, which is also overridden

\* by the query options in TweetKindOptions (see Recap.scala, the

\* parent class, for details on how this override works). For example, one

\* can choose to retrieve replies, retweets and/or extended replies

\* by changing the options passed in, which get added to ''filters''.

\* 2. The visiblityEnforcer passed in, which controls what visibility rules

\* are applied to the tweets returned from earlybird (e.g. mutes, blocks).

\*/

class RecapAuthorSource(

gizmoduckClient: GizmoduckClient,

searchClient: SearchClient,

tweetyPieClient: TweetyPieClient,

userMetadataClient: UserMetadataClient,

followGraphDataProvider: FollowGraphDataProvider,

contentFeaturesCache: Store[TweetId, ContentFeatures],

visibilityEnforcer: VisibilityEnforcer,

statsReceiver: StatsReceiver) {

private[this] val baseScope = statsReceiver.scope("recapAuthor")

private[this] val requestStats = RequestStatsReceiver(baseScope)

private[this] val failOpenScope = baseScope.scope("failOpen")

private[this] val userProfileHandler = new FailOpenHandler(failOpenScope, "userProfileInfo")

private[this] val userLanguagesHandler = new FailOpenHandler(failOpenScope, "userLanguages")

/\*

\* Similar to RecapSource, we filter out tweets directed at non-followed users that

\* are not "replies" i.e. those that begin with the @-handle.

\* For tweets to non-followed users that are replies, these are "extended replies"

\* and are handled separately by the dynamic filters (see Recap.scala for details).

\* Reply and retweet filtering is also handled by dynamic filters, overriden by

\* TweetKindOptions passed in with the query (again, details in Recap.scala)

\* We however do not filter out tweets from non-followed users, unlike RecapSource,

\* because one of the main use cases of this endpoint is to retrieve tweets from out

\* of network authors.

\*/

val filters: TweetFilters.ValueSet = TweetFilters.ValueSet(

TweetFilters.DuplicateTweets,

TweetFilters.DuplicateRetweets,

TweetFilters.DirectedAtNotFollowedUsers,

TweetFilters.NonReplyDirectedAtNotFollowedUsers

)

private[this] val visibilityEnforcingTransform = new VisibilityEnforcingTransform(

visibilityEnforcer

)

private[this] val hydratedTweetsFilter = new HydratedTweetsFilterTransform(

outerFilters = filters,

innerFilters = TweetFilters.None,

useFollowGraphData = true,

useSourceTweets = false,

statsReceiver = baseScope,

numRetweetsAllowed = HydratedTweetsFilterTransform.NumDuplicateRetweetsAllowed

)

private[this] val dynamicHydratedTweetsFilter = new TweetKindOptionHydratedTweetsFilterTransform(

useFollowGraphData = false,

useSourceTweets = false,

statsReceiver = baseScope

)

private[this] val maxFollowedUsersProvider =

DependencyProvider.value(RecapParams.MaxFollowedUsers.default)

private[this] val followGraphDataTransform =

new FollowGraphDataTransform(followGraphDataProvider, maxFollowedUsersProvider)

private[this] val maxSearchResultCountProvider = DependencyProvider { query =>

query.maxCount.getOrElse(query.params(RecapParams.DefaultMaxTweetCount))

}

private[this] val relevanceOptionsMaxHitsToProcessProvider =

DependencyProvider.from(RecapParams.RelevanceOptionsMaxHitsToProcessParam)

private[this] val retrieveSearchResultsTransform = new RecapAuthorSearchResultsTransform(

searchClient = searchClient,

maxCountProvider = maxSearchResultCountProvider,

relevanceOptionsMaxHitsToProcessProvider = relevanceOptionsMaxHitsToProcessProvider,

enableSettingTweetTypesWithTweetKindOptionProvider =

DependencyProvider.from(RecapParams.EnableSettingTweetTypesWithTweetKindOption),

statsReceiver = baseScope,

logSearchDebugInfo = false)

private[this] val debugAuthorsMonitoringProvider =

DependencyProvider.from(MonitoringParams.DebugAuthorsAllowListParam)

private[this] val preTruncateSearchResultsTransform =

new UsersSearchResultMonitoringTransform(

name = "RecapSearchResultsTruncationTransform",

new RecapSearchResultsTruncationTransform(

extraSortBeforeTruncationGate = DependencyProvider.True,

maxCountProvider = maxSearchResultCountProvider,

statsReceiver = baseScope.scope("afterSearchResultsTransform")

),

baseScope.scope("afterSearchResultsTransform"),

debugAuthorsMonitoringProvider

)

private[this] val searchResultsTransform = retrieveSearchResultsTransform

.andThen(preTruncateSearchResultsTransform)

private[this] val userProfileInfoTransform =

new UserProfileInfoTransform(userProfileHandler, gizmoduckClient)

private[this] val languagesTransform =

new UserLanguagesTransform(userLanguagesHandler, userMetadataClient)

private[this] val contentFeaturesHydrationTransform =

new ContentFeaturesHydrationTransformBuilder(

tweetyPieClient = tweetyPieClient,

contentFeaturesCache = contentFeaturesCache,

enableContentFeaturesGate =

RecapQuery.paramGate(RecapAuthorParams.EnableContentFeaturesHydrationParam),

enableTokensInContentFeaturesGate =

RecapQuery.paramGate(RecapAuthorParams.EnableTokensInContentFeaturesHydrationParam),

enableTweetTextInContentFeaturesGate =

RecapQuery.paramGate(RecapAuthorParams.EnableTweetTextInContentFeaturesHydrationParam),

enableConversationControlContentFeaturesGate = RecapQuery.paramGate(

RecapAuthorParams.EnableConversationControlInContentFeaturesHydrationParam),

enableTweetMediaHydrationGate =

RecapQuery.paramGate(RecapAuthorParams.EnableTweetMediaHydrationParam),

hydrateInReplyToTweets = false,

statsReceiver = baseScope

).build()

private[this] def hydratesContentFeatures(

hydratedEnvelope: HydratedCandidatesAndFeaturesEnvelope

): Boolean =

hydratedEnvelope.candidateEnvelope.query.hydratesContentFeatures.getOrElse(true)

private[this] val contentFeaturesTransformer = FutureArrow.choose(

predicate = hydratesContentFeatures,

ifTrue = contentFeaturesHydrationTransform

.andThen(CopyContentFeaturesIntoThriftTweetFeaturesTransform)

.andThen(CopyContentFeaturesIntoHydratedTweetsTransform),

ifFalse = FutureArrow[

HydratedCandidatesAndFeaturesEnvelope,

HydratedCandidatesAndFeaturesEnvelope

](Future.value) // empty transformer

)

private[this] val candidateGenerationTransform = new CandidateGenerationTransform(baseScope)

val hydrationAndFilteringPipeline: FutureArrow[RecapQuery, CandidateEnvelope] =

CreateCandidateEnvelopeTransform // Create empty CandidateEnvelope

.andThen(followGraphDataTransform) // Fetch follow graph data

.andThen(searchResultsTransform) // Fetch search results

.andThen(SearchResultDedupAndSortingTransform)

.andThen(CandidateTweetHydrationTransform) // candidate hydration

.andThen(visibilityEnforcingTransform) // filter hydrated tweets to visible ones

.andThen(hydratedTweetsFilter) // filter hydrated tweets based on predefined filter

.andThen(dynamicHydratedTweetsFilter) // filter hydrated tweets based on query TweetKindOption

.andThen(

TrimToMatchHydratedTweetsTransform

) // trim search result set to match filtered hydrated tweets (this needs to be accurate for feature hydration)

// runs the main pipeline in parallel with fetching user profile info and user languages

val featureHydrationDataTransform: FeatureHydrationDataTransform =

new FeatureHydrationDataTransform(

hydrationAndFilteringPipeline,

languagesTransform,

userProfileInfoTransform

)

// Copy transforms must go after the search features transform, as the search transform

// overwrites the ThriftTweetFeatures.

val featureHydrationPipeline: FutureArrow[RecapQuery, CandidateTweetsResult] =

featureHydrationDataTransform

.andThen(

InNetworkTweetsSearchFeaturesHydrationTransform

) // RecapAuthorSource uses InNetworkTweetsSearchFeaturesHydrationTransform because PYLE uses the in-network model and features.

.andThen(contentFeaturesTransformer)

.andThen(candidateGenerationTransform)

def get(query: RecapQuery): Future[CandidateTweetsResult] = {

requestStats.addEventStats {

featureHydrationPipeline(query)

}

}

def get(queries: Seq[RecapQuery]): Future[Seq[CandidateTweetsResult]] = {

Future.collect(queries.map(get))

}

}