package com.twitter.timelineranker.source

import com.google.common.annotations.VisibleForTesting

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.logging.Logger

import com.twitter.search.earlybird.thriftscala.ThriftSearchResult

import com.twitter.timelineranker.core.FollowGraphData

import com.twitter.timelineranker.model.\_

import com.twitter.timelineranker.parameters.revchron.ReverseChronTimelineQueryContext

import com.twitter.timelineranker.util.TweetFiltersBasedOnSearchMetadata

import com.twitter.timelineranker.util.TweetsPostFilterBasedOnSearchMetadata

import com.twitter.timelineranker.util.SearchResultWithVisibilityActors

import com.twitter.timelineranker.visibility.FollowGraphDataProvider

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

import com.twitter.timelines.model.TweetId

import com.twitter.timelines.model.UserId

import com.twitter.timelines.util.stats.RequestStats

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

import com.twitter.timelines.visibility.VisibilityEnforcer

import com.twitter.timelineservice.model.TimelineId

import com.twitter.timelineservice.model.core.TimelineKind

import com.twitter.util.Future

object ReverseChronHomeTimelineSource {

// Post search filters applied to tweets using metadata included in search results.

val FiltersBasedOnSearchMetadata: TweetFiltersBasedOnSearchMetadata.ValueSet =

TweetFiltersBasedOnSearchMetadata.ValueSet(

TweetFiltersBasedOnSearchMetadata.DuplicateRetweets,

TweetFiltersBasedOnSearchMetadata.DuplicateTweets

)

object GetTweetsResult {

val Empty: GetTweetsResult = GetTweetsResult(0, 0L, Nil)

val EmptyFuture: Future[GetTweetsResult] = Future.value(Empty)

}

case class GetTweetsResult(

// numSearchResults is the result count before filtering so may not match tweets.size

numSearchResults: Int,

minTweetIdFromSearch: TweetId,

tweets: Seq[Tweet])

}

/\*\*

\* Timeline source that enables materializing reverse chron timelines

\* using search infrastructure.

\*/

class ReverseChronHomeTimelineSource(

searchClient: SearchClient,

followGraphDataProvider: FollowGraphDataProvider,

visibilityEnforcer: VisibilityEnforcer,

statsReceiver: StatsReceiver)

extends RequestStats {

import ReverseChronHomeTimelineSource.\_

private[this] val logger = Logger.get("ReverseChronHomeTimelineSource")

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

private[this] val stats = RequestStatsReceiver(scope)

private[this] val emptyTimelineReturnedCounter =

scope.counter("emptyTimelineReturnedDueToMaxFollows")

private[this] val maxCountStat = scope.stat("maxCount")

private[this] val numTweetsStat = scope.stat("numTweets")

private[this] val requestedAdditionalTweetsAfterFilter =

scope.counter("requestedAdditionalTweetsAfterFilter")

private[this] val emptyTimelines = scope.counter("emptyTimelines")

private[this] val emptyTimelinesWithSignificantFollowing =

scope.counter("emptyTimelinesWithSignificantFollowing")

// Threshold to use to determine if a user has a significant followings list size

private[this] val SignificantFollowingThreshold = 20

def get(contexts: Seq[ReverseChronTimelineQueryContext]): Seq[Future[Timeline]] = {

contexts.map(get)

}

def get(context: ReverseChronTimelineQueryContext): Future[Timeline] = {

stats.addEventStats {

val query: ReverseChronTimelineQuery = context.query

// We only support Tweet ID range at present.

val tweetIdRange =

query.range.map(TweetIdRange.fromTimelineRange).getOrElse(TweetIdRange.default)

val userId = query.userId

val timelineId = TimelineId(userId, TimelineKind.home)

val maxFollowingCount = context.maxFollowedUsers()

followGraphDataProvider

.get(

userId,

maxFollowingCount

)

.flatMap { followGraphData =>

// We return an empty timeline if a given user follows more than the limit

// on the number of users. This is because, such a user's timeline will quickly

// fill up displacing materialized tweets wasting the materialation work.

// This behavior can be disabled via featureswitches to support non-materialization

// use cases when we should always return a timeline.

if (followGraphData.filteredFollowedUserIds.isEmpty ||

(followGraphData.followedUserIds.size >= maxFollowingCount && context

.returnEmptyWhenOverMaxFollows())) {

if (followGraphData.followedUserIds.size >= maxFollowingCount) {

emptyTimelineReturnedCounter.incr()

}

Future.value(Timeline.empty(timelineId))

} else {

val maxCount = getMaxCount(context)

val numEntriesToRequest = (maxCount \* context.maxCountMultiplier()).toInt

maxCountStat.add(numEntriesToRequest)

val allUserIds = followGraphData.followedUserIds :+ userId

getTweets(

userId,

allUserIds,

followGraphData,

numEntriesToRequest,

tweetIdRange,

context

).map { tweets =>

if (tweets.isEmpty) {

emptyTimelines.incr()

if (followGraphData.followedUserIds.size >= SignificantFollowingThreshold) {

emptyTimelinesWithSignificantFollowing.incr()

logger.debug(

"Search returned empty home timeline for user %s (follow count %s), query: %s",

userId,

followGraphData.followedUserIds.size,

query)

}

}

// If we had requested more entries than maxCount (due to multiplier being > 1.0)

// then we need to trim it back to maxCount.

val truncatedTweets = tweets.take(maxCount)

numTweetsStat.add(truncatedTweets.size)

Timeline(

timelineId,

truncatedTweets.map(tweet => TimelineEntryEnvelope(tweet))

)

}

}

}

}

}

/\*\*

\* Gets tweets from search and performs post-filtering.

\*

\* If we do not end up with sufficient tweets after post-filtering,

\* we issue a second call to search to get more tweets if:

\* -- such behavior is enabled by setting backfillFilteredEntries to true.

\* -- the original call to search returned requested number of tweets.

\* -- after post-filtering, the percentage of filtered out tweets

\* exceeds the value of tweetsFilteringLossageThresholdPercent.

\*/

private def getTweets(

userId: UserId,

allUserIds: Seq[UserId],

followGraphData: FollowGraphData,

numEntriesToRequest: Int,

tweetIdRange: TweetIdRange,

context: ReverseChronTimelineQueryContext

): Future[Seq[Tweet]] = {

getTweetsHelper(

userId,

allUserIds,

followGraphData,

numEntriesToRequest,

tweetIdRange,

context.directedAtNarrowcastingViaSearch(),

context.postFilteringBasedOnSearchMetadataEnabled(),

context.getTweetsFromArchiveIndex()

).flatMap { result =>

val numAdditionalTweetsToRequest = getNumAdditionalTweetsToRequest(

numEntriesToRequest,

result.numSearchResults,

result.numSearchResults - result.tweets.size,

context

)

if (numAdditionalTweetsToRequest > 0) {

requestedAdditionalTweetsAfterFilter.incr()

val updatedRange = tweetIdRange.copy(toId = Some(result.minTweetIdFromSearch))

getTweetsHelper(

userId,

allUserIds,

followGraphData,

numAdditionalTweetsToRequest,

updatedRange,

context.directedAtNarrowcastingViaSearch(),

context.postFilteringBasedOnSearchMetadataEnabled(),

context.getTweetsFromArchiveIndex()

).map { result2 => result.tweets ++ result2.tweets }

} else {

Future.value(result.tweets)

}

}

}

private[source] def getNumAdditionalTweetsToRequest(

numTweetsRequested: Int,

numTweetsFoundBySearch: Int,

numTweetsFilteredOut: Int,

context: ReverseChronTimelineQueryContext

): Int = {

require(numTweetsFoundBySearch <= numTweetsRequested)

if (!context.backfillFilteredEntries() || (numTweetsFoundBySearch < numTweetsRequested)) {

// If multiple calls are not enabled or if search did not find enough tweets,

// there is no point in making another call to get more.

0

} else {

val numTweetsFilteredOutPercent = numTweetsFilteredOut \* 100.0 / numTweetsFoundBySearch

if (numTweetsFilteredOutPercent > context.tweetsFilteringLossageThresholdPercent()) {

// We assume that the next call will also have lossage percentage similar to the first call.

// Therefore, we proactively request proportionately more tweets so that we do not

// end up needing a third call.

// In any case, regardless of what we get in the second call, we do not make any subsequent calls.

val adjustedFilteredOutPercent =

math.min(numTweetsFilteredOutPercent, context.tweetsFilteringLossageLimitPercent())

val numTweetsToRequestMultiplier = 100 / (100 - adjustedFilteredOutPercent)

val numTweetsToRequest = (numTweetsFilteredOut \* numTweetsToRequestMultiplier).toInt

numTweetsToRequest

} else {

// Did not have sufficient lossage to warrant an extra call.

0

}

}

}

private def getClientId(subClientId: String): String = {

// Hacky: Extract the environment from the existing clientId set by TimelineRepositoryBuilder

val env = searchClient.clientId.split('.').last

s"timelineranker.$subClientId.$env"

}

private def getTweetsHelper(

userId: UserId,

allUserIds: Seq[UserId],

followGraphData: FollowGraphData,

maxCount: Int,

tweetIdRange: TweetIdRange,

withDirectedAtNarrowcasting: Boolean,

postFilteringBasedOnSearchMetadataEnabled: Boolean,

getTweetsFromArchiveIndex: Boolean

): Future[GetTweetsResult] = {

val beforeTweetIdExclusive = tweetIdRange.toId

val afterTweetIdExclusive = tweetIdRange.fromId

val searchClientId: Option[String] = if (!getTweetsFromArchiveIndex) {

// Set a custom clientId which has different QPS quota and access.

// Used for notify we are fetching from realtime only.

// see: SEARCH-42651

Some(getClientId("home\_materialization\_realtime\_only"))

} else {

// Let the searchClient derive its clientId for the regular case of fetching from archive

None

}

searchClient

.getUsersTweetsReverseChron(

userId = userId,

followedUserIds = allUserIds.toSet,

retweetsMutedUserIds = followGraphData.retweetsMutedUserIds,

maxCount = maxCount,

beforeTweetIdExclusive = beforeTweetIdExclusive,

afterTweetIdExclusive = afterTweetIdExclusive,

withDirectedAtNarrowcasting = withDirectedAtNarrowcasting,

postFilteringBasedOnSearchMetadataEnabled = postFilteringBasedOnSearchMetadataEnabled,

getTweetsFromArchiveIndex = getTweetsFromArchiveIndex,

searchClientId = searchClientId

)

.flatMap { searchResults =>

if (searchResults.nonEmpty) {

val minTweetId = searchResults.last.id

val filteredTweetsFuture = filterTweets(

userId,

followGraphData.inNetworkUserIds,

searchResults,

FiltersBasedOnSearchMetadata,

postFilteringBasedOnSearchMetadataEnabled = postFilteringBasedOnSearchMetadataEnabled,

visibilityEnforcer

)

filteredTweetsFuture.map(tweets =>

GetTweetsResult(searchResults.size, minTweetId, tweets))

} else {

GetTweetsResult.EmptyFuture

}

}

}

def filterTweets(

userId: UserId,

inNetworkUserIds: Seq[UserId],

searchResults: Seq[ThriftSearchResult],

filtersBasedOnSearchMetadata: TweetFiltersBasedOnSearchMetadata.ValueSet,

postFilteringBasedOnSearchMetadataEnabled: Boolean = true,

visibilityEnforcer: VisibilityEnforcer

): Future[Seq[Tweet]] = {

val filteredTweets = if (postFilteringBasedOnSearchMetadataEnabled) {

val tweetsPostFilterBasedOnSearchMetadata =

new TweetsPostFilterBasedOnSearchMetadata(filtersBasedOnSearchMetadata, logger, scope)

tweetsPostFilterBasedOnSearchMetadata.apply(userId, inNetworkUserIds, searchResults)

} else {

searchResults

}

visibilityEnforcer

.apply(Some(userId), filteredTweets.map(SearchResultWithVisibilityActors(\_, scope)))

.map(\_.map { searchResult =>

new Tweet(

id = searchResult.tweetId,

userId = Some(searchResult.userId),

sourceTweetId = searchResult.sourceTweetId,

sourceUserId = searchResult.sourceUserId)

})

}

@VisibleForTesting

private[source] def getMaxCount(context: ReverseChronTimelineQueryContext): Int = {

val maxCountFromQuery = ReverseChronTimelineQueryContext.MaxCount(context.query.maxCount)

val maxCountFromContext = context.maxCount()

math.min(maxCountFromQuery, maxCountFromContext)

}

}