package com.twitter.usersignalservice.signals

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.store.strato.StratoFetchableStore

import com.twitter.frigate.data\_pipeline.candidate\_generation.thriftscala.ClientEngagementEvent

import com.twitter.frigate.data\_pipeline.candidate\_generation.thriftscala.LatestEvents

import com.twitter.frigate.data\_pipeline.candidate\_generation.thriftscala.LatestNegativeEngagementEvents

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

import com.twitter.storehaus.ReadableStore

import com.twitter.strato.client.Client

import com.twitter.twistly.common.TweetId

import com.twitter.twistly.common.UserId

import com.twitter.usersignalservice.base.BaseSignalFetcher

import com.twitter.usersignalservice.base.Query

import com.twitter.usersignalservice.thriftscala.Signal

import com.twitter.usersignalservice.thriftscala.SignalType

import com.twitter.util.Future

import com.twitter.util.Timer

import javax.inject.Inject

import javax.inject.Singleton

@Singleton

case class NotificationOpenAndClickFetcher @Inject() (

stratoClient: Client,

timer: Timer,

stats: StatsReceiver)

extends BaseSignalFetcher {

import NotificationOpenAndClickFetcher.\_

override type RawSignalType = ClientEngagementEvent

override val name: String = this.getClass.getCanonicalName

override val statsReceiver: StatsReceiver = stats.scope(this.name)

private val latestEventsStore: ReadableStore[UserId, LatestEvents] = {

StratoFetchableStore

.withUnitView[UserId, LatestEvents](stratoClient, latestEventStoreColumn)

}

private val notificationNegativeEngagementStore: ReadableStore[UserId, Seq[

NotificationNegativeEngagement

]] = {

StratoFetchableStore

.withUnitView[UserId, LatestNegativeEngagementEvents](

stratoClient,

labeledPushRecsNegativeEngagementsColumn).mapValues(fromLatestNegativeEngagementEvents)

}

override def getRawSignals(

userId: UserId

): Future[Option[Seq[RawSignalType]]] = {

val notificationNegativeEngagementEventsFut =

notificationNegativeEngagementStore.get(userId)

val latestEventsFut = latestEventsStore.get(userId)

Future

.join(latestEventsFut, notificationNegativeEngagementEventsFut).map {

case (latestEventsOpt, latestNegativeEngagementEventsOpt) =>

latestEventsOpt.map { latestEvents =>

// Negative Engagement Events Filter

filterNegativeEngagementEvents(

latestEvents.engagementEvents,

latestNegativeEngagementEventsOpt.getOrElse(Seq.empty),

statsReceiver.scope("filterNegativeEngagementEvents"))

}

}

}

override def process(

query: Query,

rawSignals: Future[Option[Seq[RawSignalType]]]

): Future[Option[Seq[Signal]]] = {

rawSignals.map {

\_.map {

\_.take(query.maxResults.getOrElse(Int.MaxValue)).map { clientEngagementEvent =>

Signal(

SignalType.NotificationOpenAndClickV1,

timestamp = clientEngagementEvent.timestampMillis,

targetInternalId = Some(InternalId.TweetId(clientEngagementEvent.tweetId))

)

}

}

}

}

}

object NotificationOpenAndClickFetcher {

private val latestEventStoreColumn = "frigate/magicrecs/labeledPushRecsAggregated.User"

private val labeledPushRecsNegativeEngagementsColumn =

"frigate/magicrecs/labeledPushRecsNegativeEngagements.User"

case class NotificationNegativeEngagement(

tweetId: TweetId,

timestampMillis: Long,

isNtabDisliked: Boolean,

isReportTweetClicked: Boolean,

isReportTweetDone: Boolean,

isReportUserClicked: Boolean,

isReportUserDone: Boolean)

def fromLatestNegativeEngagementEvents(

latestNegativeEngagementEvents: LatestNegativeEngagementEvents

): Seq[NotificationNegativeEngagement] = {

latestNegativeEngagementEvents.negativeEngagementEvents.map { event =>

NotificationNegativeEngagement(

event.tweetId,

event.timestampMillis,

event.isNtabDisliked.getOrElse(false),

event.isReportTweetClicked.getOrElse(false),

event.isReportTweetDone.getOrElse(false),

event.isReportUserClicked.getOrElse(false),

event.isReportUserDone.getOrElse(false)

)

}

}

private def filterNegativeEngagementEvents(

engagementEvents: Seq[ClientEngagementEvent],

negativeEvents: Seq[NotificationNegativeEngagement],

statsReceiver: StatsReceiver

): Seq[ClientEngagementEvent] = {

if (negativeEvents.nonEmpty) {

statsReceiver.counter("filterNegativeEngagementEvents").incr()

statsReceiver.stat("eventSizeBeforeFilter").add(engagementEvents.size)

val negativeEngagementIdSet =

negativeEvents.collect {

case event

if event.isNtabDisliked || event.isReportTweetClicked || event.isReportTweetDone || event.isReportUserClicked || event.isReportUserDone =>

event.tweetId

}.toSet

// negative event size

statsReceiver.stat("negativeEventsSize").add(negativeEngagementIdSet.size)

// filter out negative engagement sources

val result = engagementEvents.filterNot { event =>

negativeEngagementIdSet.contains(event.tweetId)

}

statsReceiver.stat("eventSizeAfterFilter").add(result.size)

result

} else engagementEvents

}

}