package com.twitter.frigate.pushservice.store

import com.twitter.conversions.DurationOps.\_

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.candidate.TargetDecider

import com.twitter.frigate.common.history.History

import com.twitter.frigate.common.history.HistoryStoreKeyContext

import com.twitter.frigate.common.history.PushServiceHistoryStore

import com.twitter.frigate.data\_pipeline.thriftscala.\_

import com.twitter.frigate.thriftscala.FrigateNotification

import com.twitter.hermit.store.labeled\_push\_recs.LabeledPushRecsJoinedWithNotificationHistoryStore

import com.twitter.logging.Logger

import com.twitter.storehaus.ReadableStore

import com.twitter.util.Future

import com.twitter.util.Time

case class LabeledPushRecsVerifyingStoreKey(

historyStoreKey: HistoryStoreKeyContext,

useHydratedDataset: Boolean,

verifyHydratedDatasetResults: Boolean) {

def userId: Long = historyStoreKey.targetUserId

}

case class LabeledPushRecsVerifyingStoreResponse(

userHistory: UserHistoryValue,

unequalNotificationsUnhydratedToHydrated: Option[

Map[(Time, FrigateNotification), FrigateNotification]

],

missingFromHydrated: Option[Map[Time, FrigateNotification]])

case class LabeledPushRecsVerifyingStore(

labeledPushRecsStore: ReadableStore[UserHistoryKey, UserHistoryValue],

historyStore: PushServiceHistoryStore

)(

implicit stats: StatsReceiver)

extends ReadableStore[LabeledPushRecsVerifyingStoreKey, LabeledPushRecsVerifyingStoreResponse] {

private def getByJoiningWithRealHistory(

key: HistoryStoreKeyContext

): Future[Option[UserHistoryValue]] = {

val historyFut = historyStore.get(key, Some(365.days))

val toJoinWithRealHistoryFut = labeledPushRecsStore.get(UserHistoryKey.UserId(key.targetUserId))

Future.join(historyFut, toJoinWithRealHistoryFut).map {

case (\_, None) => None

case (History(realtimeHistoryMap), Some(uhValue)) =>

Some(

LabeledPushRecsJoinedWithNotificationHistoryStore

.joinLabeledPushRecsSentWithNotificationHistory(uhValue, realtimeHistoryMap, stats)

)

}

}

private def processUserHistoryValue(uhValue: UserHistoryValue): Map[Time, FrigateNotification] = {

uhValue.events

.getOrElse(Nil)

.collect {

case Event(

EventType.LabeledPushRecSend,

Some(tsMillis),

Some(EventUnion.LabeledPushRecSendEvent(lprs: LabeledPushRecSendEvent))

) if lprs.pushRecSendEvent.frigateNotification.isDefined =>

Time.fromMilliseconds(tsMillis) -> lprs.pushRecSendEvent.frigateNotification.get

}

.toMap

}

override def get(

key: LabeledPushRecsVerifyingStoreKey

): Future[Option[LabeledPushRecsVerifyingStoreResponse]] = {

val uhKey = UserHistoryKey.UserId(key.userId)

if (!key.useHydratedDataset) {

getByJoiningWithRealHistory(key.historyStoreKey).map { uhValueOpt =>

uhValueOpt.map { uhValue => LabeledPushRecsVerifyingStoreResponse(uhValue, None, None) }

}

} else {

labeledPushRecsStore.get(uhKey).flatMap { hydratedValueOpt: Option[UserHistoryValue] =>

if (!key.verifyHydratedDatasetResults) {

Future.value(hydratedValueOpt.map { uhValue =>

LabeledPushRecsVerifyingStoreResponse(uhValue, None, None)

})

} else {

getByJoiningWithRealHistory(key.historyStoreKey).map {

joinedWithRealHistoryOpt: Option[UserHistoryValue] =>

val joinedWithRealHistoryMap =

joinedWithRealHistoryOpt.map(processUserHistoryValue).getOrElse(Map.empty)

val hydratedMap = hydratedValueOpt.map(processUserHistoryValue).getOrElse(Map.empty)

val unequal = joinedWithRealHistoryMap.flatMap {

case (time, frigateNotif) =>

hydratedMap.get(time).collect {

case n if n != frigateNotif => ((time, frigateNotif), n)

}

}

val missing = joinedWithRealHistoryMap.filter {

case (time, frigateNotif) => !hydratedMap.contains(time)

}

hydratedValueOpt.map { hydratedValue =>

LabeledPushRecsVerifyingStoreResponse(hydratedValue, Some(unequal), Some(missing))

}

}

}

}

}

}

}

case class LabeledPushRecsStoreKey(target: TargetDecider, historyStoreKey: HistoryStoreKeyContext) {

def userId: Long = historyStoreKey.targetUserId

}

case class LabeledPushRecsDecideredStore(

verifyingStore: ReadableStore[

LabeledPushRecsVerifyingStoreKey,

LabeledPushRecsVerifyingStoreResponse

],

useHydratedLabeledSendsDatasetDeciderKey: String,

verifyHydratedLabeledSendsForHistoryDeciderKey: String

)(

implicit globalStats: StatsReceiver)

extends ReadableStore[LabeledPushRecsStoreKey, UserHistoryValue] {

private val log = Logger()

private val stats = globalStats.scope("LabeledPushRecsDecideredStore")

private val numComparisons = stats.counter("num\_comparisons")

private val numMissingStat = stats.stat("num\_missing")

private val numUnequalStat = stats.stat("num\_unequal")

override def get(key: LabeledPushRecsStoreKey): Future[Option[UserHistoryValue]] = {

val useHydrated = key.target.isDeciderEnabled(

useHydratedLabeledSendsDatasetDeciderKey,

stats,

useRandomRecipient = true

)

val verifyHydrated = if (useHydrated) {

key.target.isDeciderEnabled(

verifyHydratedLabeledSendsForHistoryDeciderKey,

stats,

useRandomRecipient = true

)

} else false

val newKey = LabeledPushRecsVerifyingStoreKey(key.historyStoreKey, useHydrated, verifyHydrated)

verifyingStore.get(newKey).map {

case None => None

case Some(LabeledPushRecsVerifyingStoreResponse(uhValue, unequalOpt, missingOpt)) =>

(unequalOpt, missingOpt) match {

case (Some(unequal), Some(missing)) =>

numComparisons.incr()

numMissingStat.add(missing.size)

numUnequalStat.add(unequal.size)

case \_ => //no-op

}

Some(uhValue)

}

}

}