package com.twitter.home\_mixer.product.scored\_tweets.feature\_hydrator.real\_time\_aggregates

import com.twitter.home\_mixer.product.scored\_tweets.feature\_hydrator.real\_time\_aggregates.BaseRealtimeAggregateHydrator.\_

import com.twitter.home\_mixer.util.DataRecordUtil

import com.twitter.home\_mixer.util.ObservedKeyValueResultHandler

import com.twitter.ml.api.DataRecord

import com.twitter.ml.api.DataRecordMerger

import com.twitter.ml.api.FeatureContext

import com.twitter.ml.api.constant.SharedFeatures

import com.twitter.ml.api.util.SRichDataRecord

import com.twitter.ml.api.{Feature => MLApiFeature}

import com.twitter.servo.cache.ReadCache

import com.twitter.servo.keyvalue.KeyValueResult

import com.twitter.timelines.data\_processing.ml\_util.aggregation\_framework.AggregateGroup

import com.twitter.util.Future

import com.twitter.util.Time

import com.twitter.util.Try

import java.lang.{Double => JDouble}

import scala.collection.JavaConverters.\_

trait BaseRealtimeAggregateHydrator[K] extends ObservedKeyValueResultHandler {

val client: ReadCache[K, DataRecord]

val aggregateGroups: Seq[AggregateGroup]

val aggregateGroupToPrefix: Map[AggregateGroup, String] = Map.empty

private lazy val typedAggregateGroupsList = aggregateGroups.map(\_.buildTypedAggregateGroups())

private lazy val featureContexts: Seq[FeatureContext] = typedAggregateGroupsList.map {

typedAggregateGroups =>

new FeatureContext(

(SharedFeatures.TIMESTAMP +: typedAggregateGroups.flatMap(\_.allOutputFeatures)).asJava

)

}

private lazy val aggregateFeaturesRenameMap: Map[MLApiFeature[\_], MLApiFeature[\_]] = {

val prefixes: Seq[Option[String]] = aggregateGroups.map(aggregateGroupToPrefix.get)

typedAggregateGroupsList

.zip(prefixes).map {

case (typedAggregateGroups, prefix) =>

if (prefix.nonEmpty)

typedAggregateGroups

.map {

\_.outputFeaturesToRenamedOutputFeatures(prefix.get)

}.reduce(\_ ++ \_)

else

Map.empty[MLApiFeature[\_], MLApiFeature[\_]]

}.reduce(\_ ++ \_)

}

private lazy val renamedFeatureContexts: Seq[FeatureContext] =

typedAggregateGroupsList.map { typedAggregateGroups =>

val renamedAllOutputFeatures = typedAggregateGroups.flatMap(\_.allOutputFeatures).map {

feature => aggregateFeaturesRenameMap.getOrElse(feature, feature)

}

new FeatureContext(renamedAllOutputFeatures.asJava)

}

private lazy val decays: Seq[TimeDecay] = typedAggregateGroupsList.map { typedAggregateGroups =>

RealTimeAggregateTimeDecay(

typedAggregateGroups.flatMap(\_.continuousFeatureIdsToHalfLives).toMap)

.apply(\_, \_)

}

private val drMerger = new DataRecordMerger

private def postTransformer(dataRecord: Try[Option[DataRecord]]): Try[DataRecord] = {

dataRecord.map {

case Some(dr) =>

val newDr = new DataRecord()

featureContexts.zip(renamedFeatureContexts).zip(decays).foreach {

case ((featureContext, renamedFeatureContext), decay) =>

val decayedDr = applyDecay(dr, featureContext, decay)

val renamedDr = DataRecordUtil.applyRename(

dataRecord = decayedDr,

featureContext,

renamedFeatureContext,

aggregateFeaturesRenameMap)

drMerger.merge(newDr, renamedDr)

}

newDr

case \_ => new DataRecord

}

}

def fetchAndConstructDataRecords(possiblyKeys: Seq[Option[K]]): Future[Seq[Try[DataRecord]]] = {

val keys = possiblyKeys.flatten

val response: Future[KeyValueResult[K, DataRecord]] =

if (keys.isEmpty) Future.value(KeyValueResult.empty)

else {

val batchResponses = keys

.grouped(RequestBatchSize)

.map(keyGroup => client.get(keyGroup))

.toSeq

Future.collect(batchResponses).map(\_.reduce(\_ ++ \_))

}

response.map { result =>

possiblyKeys.map { possiblyKey =>

val value = observedGet(key = possiblyKey, keyValueResult = result)

postTransformer(value)

}

}

}

}

object BaseRealtimeAggregateHydrator {

private val RequestBatchSize = 5

type TimeDecay = scala.Function2[com.twitter.ml.api.DataRecord, scala.Long, scala.Unit]

private def applyDecay(

dataRecord: DataRecord,

featureContext: FeatureContext,

decay: TimeDecay

): DataRecord = {

def time: Long = Time.now.inMillis

val richFullDr = new SRichDataRecord(dataRecord, featureContext)

val richNewDr = new SRichDataRecord(new DataRecord, featureContext)

val featureIterator = featureContext.iterator()

featureIterator.forEachRemaining { feature =>

if (richFullDr.hasFeature(feature)) {

val typedFeature = feature.asInstanceOf[MLApiFeature[JDouble]]

richNewDr.setFeatureValue(typedFeature, richFullDr.getFeatureValue(typedFeature))

}

}

val resultDr = richNewDr.getRecord

decay(resultDr, time)

resultDr

}

}