package com.twitter.timelines.prediction.common.aggregates.real\_time

import com.twitter.finagle.stats.Counter

import com.twitter.finagle.stats.StatsReceiver

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

import com.twitter.ml.api.DataRecord

import com.twitter.ml.api.DataRecordMerger

import com.twitter.ml.api.Feature

import com.twitter.ml.api.RichDataRecord

import com.twitter.ml.featurestore.catalog.entities.core.Author

import com.twitter.ml.featurestore.catalog.entities.core.Tweet

import com.twitter.ml.featurestore.catalog.entities.core.User

import com.twitter.ml.featurestore.lib.online.FeatureStoreClient

import com.twitter.summingbird.Producer

import com.twitter.summingbird.storm.Storm

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

import com.twitter.timelines.prediction.features.common.TimelinesSharedFeatures

import java.lang.{Long => JLong}

import com.twitter.unified\_user\_actions.thriftscala.ActionType

import com.twitter.unified\_user\_actions.thriftscala.UnifiedUserAction

private[real\_time] object StormAggregateSourceUtils {

type UserId = Long

type AuthorId = Long

type TweetId = Long

/\*\*

\* Attaches a [[FeatureStoreClient]] to the underyling [[Producer]]. The FeatureStoreClient

\* hydrates additional user features.

\*

\* @param underlyingProducer converts a stream of [[com.twitter.clientapp.thriftscala.LogEvent]]

\* to a stream of [[DataRecord]].

\*/

def wrapByFeatureStoreClient(

underlyingProducer: Producer[Storm, Event[DataRecord]],

jobConfig: RealTimeAggregatesJobConfig,

scopedStatsReceiver: StatsReceiver

): Producer[Storm, Event[DataRecord]] = {

lazy val keyDataRecordCounter = scopedStatsReceiver.counter("keyDataRecord")

lazy val keyFeatureCounter = scopedStatsReceiver.counter("keyFeature")

lazy val leftDataRecordCounter = scopedStatsReceiver.counter("leftDataRecord")

lazy val rightDataRecordCounter = scopedStatsReceiver.counter("rightDataRecord")

lazy val mergeNumFeaturesCounter = scopedStatsReceiver.counter("mergeNumFeatures")

lazy val authorKeyDataRecordCounter = scopedStatsReceiver.counter("authorKeyDataRecord")

lazy val authorKeyFeatureCounter = scopedStatsReceiver.counter("authorKeyFeature")

lazy val authorLeftDataRecordCounter = scopedStatsReceiver.counter("authorLeftDataRecord")

lazy val authorRightDataRecordCounter = scopedStatsReceiver.counter("authorRightDataRecord")

lazy val authorMergeNumFeaturesCounter = scopedStatsReceiver.counter("authorMergeNumFeatures")

lazy val tweetKeyDataRecordCounter =

scopedStatsReceiver.counter("tweetKeyDataRecord")

lazy val tweetKeyFeatureCounter = scopedStatsReceiver.counter("tweetKeyFeature")

lazy val tweetLeftDataRecordCounter =

scopedStatsReceiver.counter("tweetLeftDataRecord")

lazy val tweetRightDataRecordCounter =

scopedStatsReceiver.counter("tweetRightDataRecord")

lazy val tweetMergeNumFeaturesCounter =

scopedStatsReceiver.counter("tweetMergeNumFeatures")

@transient lazy val featureStoreClient: FeatureStoreClient =

FeatureStoreUtils.mkFeatureStoreClient(

serviceIdentifier = jobConfig.serviceIdentifier,

statsReceiver = scopedStatsReceiver

)

lazy val joinUserFeaturesDataRecordProducer =

if (jobConfig.keyedByUserEnabled) {

lazy val keyedByUserFeaturesStormService: Storm#Service[Set[UserId], DataRecord] =

Storm.service(

new UserFeaturesReadableStore(

featureStoreClient = featureStoreClient,

userEntity = User,

userFeaturesAdapter = UserFeaturesAdapter

)

)

leftJoinDataRecordProducer(

keyFeature = SharedFeatures.USER\_ID,

leftDataRecordProducer = underlyingProducer,

rightStormService = keyedByUserFeaturesStormService,

keyDataRecordCounter = keyDataRecordCounter,

keyFeatureCounter = keyFeatureCounter,

leftDataRecordCounter = leftDataRecordCounter,

rightDataRecordCounter = rightDataRecordCounter,

mergeNumFeaturesCounter = mergeNumFeaturesCounter

)

} else {

underlyingProducer

}

lazy val joinAuthorFeaturesDataRecordProducer =

if (jobConfig.keyedByAuthorEnabled) {

lazy val keyedByAuthorFeaturesStormService: Storm#Service[Set[AuthorId], DataRecord] =

Storm.service(

new UserFeaturesReadableStore(

featureStoreClient = featureStoreClient,

userEntity = Author,

userFeaturesAdapter = AuthorFeaturesAdapter

)

)

leftJoinDataRecordProducer(

keyFeature = TimelinesSharedFeatures.SOURCE\_AUTHOR\_ID,

leftDataRecordProducer = joinUserFeaturesDataRecordProducer,

rightStormService = keyedByAuthorFeaturesStormService,

keyDataRecordCounter = authorKeyDataRecordCounter,

keyFeatureCounter = authorKeyFeatureCounter,

leftDataRecordCounter = authorLeftDataRecordCounter,

rightDataRecordCounter = authorRightDataRecordCounter,

mergeNumFeaturesCounter = authorMergeNumFeaturesCounter

)

} else {

joinUserFeaturesDataRecordProducer

}

lazy val joinTweetFeaturesDataRecordProducer = {

if (jobConfig.keyedByTweetEnabled) {

lazy val keyedByTweetFeaturesStormService: Storm#Service[Set[TweetId], DataRecord] =

Storm.service(

new TweetFeaturesReadableStore(

featureStoreClient = featureStoreClient,

tweetEntity = Tweet,

tweetFeaturesAdapter = TweetFeaturesAdapter

)

)

leftJoinDataRecordProducer(

keyFeature = TimelinesSharedFeatures.SOURCE\_TWEET\_ID,

leftDataRecordProducer = joinAuthorFeaturesDataRecordProducer,

rightStormService = keyedByTweetFeaturesStormService,

keyDataRecordCounter = tweetKeyDataRecordCounter,

keyFeatureCounter = tweetKeyFeatureCounter,

leftDataRecordCounter = tweetLeftDataRecordCounter,

rightDataRecordCounter = tweetRightDataRecordCounter,

mergeNumFeaturesCounter = tweetMergeNumFeaturesCounter

)

} else {

joinAuthorFeaturesDataRecordProducer

}

}

joinTweetFeaturesDataRecordProducer

}

private[this] lazy val DataRecordMerger = new DataRecordMerger

/\*\*

\* Make join key from the client event data record and return both.

\* @param keyFeature Feature to extract join key value: USER\_ID, SOURCE\_TWEET\_ID, etc.

\* @param record DataRecord containing client engagement and basic tweet-side features

\* @return The return type is a tuple of this key and original data record which will be used

\* in the subsequent leftJoin operation.

\*/

private[this] def mkKey(

keyFeature: Feature[JLong],

record: DataRecord,

keyDataRecordCounter: Counter,

keyFeatureCounter: Counter

): Set[Long] = {

keyDataRecordCounter.incr()

val richRecord = new RichDataRecord(record)

if (richRecord.hasFeature(keyFeature)) {

keyFeatureCounter.incr()

val key: Long = richRecord.getFeatureValue(keyFeature).toLong

Set(key)

} else {

Set.empty[Long]

}

}

/\*\*

\* After the leftJoin, merge the client event data record and the joined data record

\* into a single data record used for further aggregation.

\*/

private[this] def mergeDataRecord(

leftRecord: Event[DataRecord],

rightRecordOpt: Option[DataRecord],

leftDataRecordCounter: Counter,

rightDataRecordCounter: Counter,

mergeNumFeaturesCounter: Counter

): Event[DataRecord] = {

leftDataRecordCounter.incr()

rightRecordOpt.foreach { rightRecord =>

rightDataRecordCounter.incr()

DataRecordMerger.merge(leftRecord.event, rightRecord)

mergeNumFeaturesCounter.incr(new RichDataRecord(leftRecord.event).numFeatures())

}

leftRecord

}

private[this] def leftJoinDataRecordProducer(

keyFeature: Feature[JLong],

leftDataRecordProducer: Producer[Storm, Event[DataRecord]],

rightStormService: Storm#Service[Set[Long], DataRecord],

keyDataRecordCounter: => Counter,

keyFeatureCounter: => Counter,

leftDataRecordCounter: => Counter,

rightDataRecordCounter: => Counter,

mergeNumFeaturesCounter: => Counter

): Producer[Storm, Event[DataRecord]] = {

val keyedLeftDataRecordProducer: Producer[Storm, (Set[Long], Event[DataRecord])] =

leftDataRecordProducer.map {

case dataRecord: HomeEvent[DataRecord] =>

val key = mkKey(

keyFeature = keyFeature,

record = dataRecord.event,

keyDataRecordCounter = keyDataRecordCounter,

keyFeatureCounter = keyFeatureCounter

)

(key, dataRecord)

case dataRecord: ProfileEvent[DataRecord] =>

val key = Set.empty[Long]

(key, dataRecord)

case dataRecord: SearchEvent[DataRecord] =>

val key = Set.empty[Long]

(key, dataRecord)

case dataRecord: UuaEvent[DataRecord] =>

val key = Set.empty[Long]

(key, dataRecord)

}

keyedLeftDataRecordProducer

.leftJoin(rightStormService)

.map {

case (\_, (leftRecord, rightRecordOpt)) =>

mergeDataRecord(

leftRecord = leftRecord,

rightRecordOpt = rightRecordOpt,

leftDataRecordCounter = leftDataRecordCounter,

rightDataRecordCounter = rightDataRecordCounter,

mergeNumFeaturesCounter = mergeNumFeaturesCounter

)

}

}

/\*\*

\* Filter Unified User Actions events to include only actions that has home timeline visit prior to landing on the page

\*/

def isUuaBCEEventsFromHome(event: UnifiedUserAction): Boolean = {

def breadcrumbViewsContain(view: String): Boolean =

event.eventMetadata.breadcrumbViews.map(\_.contains(view)).getOrElse(false)

(event.actionType) match {

case ActionType.ClientTweetV2Impression if breadcrumbViewsContain("home") =>

true

case ActionType.ClientTweetVideoFullscreenV2Impression

if (breadcrumbViewsContain("home") & breadcrumbViewsContain("video")) =>

true

case ActionType.ClientProfileV2Impression if breadcrumbViewsContain("home") =>

true

case \_ => false

}

}

}