package com.twitter.timelines.data\_processing.ml\_util.aggregation\_framework.scalding

import com.twitter.ml.api.\_

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

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

import com.twitter.scalding.Stat

import com.twitter.scalding.typed.TypedPipe

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

import com.twitter.timelines.data\_processing.ml\_util.sampling.SamplingUtils

trait AggregateFeaturesMergerBase {

import Utils.\_

def samplingRateOpt: Option[Double]

def numReducers: Int = 2000

def numReducersMerge: Int = 20000

def aggregationConfig: AggregationConfig

def storeRegister: StoreRegister

def storeMerger: StoreMerger

def getAggregatePipe(storeName: String): DataSetPipe

def applyMaxSizeByTypeOpt(aggregateType: AggregateType.Value): Option[Int] = Option.empty[Int]

def usersActiveSourcePipe: TypedPipe[Long]

def numRecords: Stat

def numFilteredRecords: Stat

/\*

\* This method should only be called with a storeName that corresponds

\* to a user aggregate store.

\*/

def extractUserFeaturesMap(storeName: String): TypedPipe[(Long, KeyedRecord)] = {

val aggregateKey = storeRegister.storeNameToTypeMap(storeName)

samplingRateOpt

.map(rate => SamplingUtils.userBasedSample(getAggregatePipe(storeName), rate))

.getOrElse(getAggregatePipe(storeName)) // must return store with only user aggregates

.records

.map { r: DataRecord =>

val record = SRichDataRecord(r)

val userId = record.getFeatureValue(USER\_ID).longValue

record.clearFeature(USER\_ID)

(userId, KeyedRecord(aggregateKey, r))

}

}

/\*

\* When the secondaryKey being used is a String, then the shouldHash function should be set to true.

\* Refactor such that the shouldHash parameter is removed and the behavior

\* is defaulted to true.

\*

\* This method should only be called with a storeName that contains records with the

\* desired secondaryKey. We provide secondaryKeyFilterPipeOpt against which secondary

\* keys can be filtered to help prune the final merged MH dataset.

\*/

def extractSecondaryTuples[T](

storeName: String,

secondaryKey: Feature[T],

shouldHash: Boolean = false,

maxSizeOpt: Option[Int] = None,

secondaryKeyFilterPipeOpt: Option[TypedPipe[Long]] = None

): TypedPipe[(Long, KeyedRecordMap)] = {

val aggregateKey = storeRegister.storeNameToTypeMap(storeName)

val extractedRecordsBySecondaryKey =

samplingRateOpt

.map(rate => SamplingUtils.userBasedSample(getAggregatePipe(storeName), rate))

.getOrElse(getAggregatePipe(storeName))

.records

.map { r: DataRecord =>

val record = SRichDataRecord(r)

val userId = keyFromLong(r, USER\_ID)

val secondaryId = extractSecondary(r, secondaryKey, shouldHash)

record.clearFeature(USER\_ID)

record.clearFeature(secondaryKey)

numRecords.inc()

(userId, secondaryId -> r)

}

val grouped =

(secondaryKeyFilterPipeOpt match {

case Some(secondaryKeyFilterPipe: TypedPipe[Long]) =>

extractedRecordsBySecondaryKey

.map {

// In this step, we swap `userId` with `secondaryId` to join on the `secondaryId`

// It is important to swap them back after the join, otherwise the job will fail.

case (userId, (secondaryId, r)) =>

(secondaryId, (userId, r))

}

.join(secondaryKeyFilterPipe.groupBy(identity))

.map {

case (secondaryId, ((userId, r), \_)) =>

numFilteredRecords.inc()

(userId, secondaryId -> r)

}

case \_ => extractedRecordsBySecondaryKey

}).group

.withReducers(numReducers)

maxSizeOpt match {

case Some(maxSize) =>

grouped

.take(maxSize)

.mapValueStream(recordsIter => Iterator(KeyedRecordMap(aggregateKey, recordsIter.toMap)))

.toTypedPipe

case None =>

grouped

.mapValueStream(recordsIter => Iterator(KeyedRecordMap(aggregateKey, recordsIter.toMap)))

.toTypedPipe

}

}

def userPipes: Seq[TypedPipe[(Long, KeyedRecord)]] =

storeRegister.allStores.flatMap { storeConfig =>

val StoreConfig(storeNames, aggregateType, \_) = storeConfig

require(storeMerger.isValidToMerge(storeNames))

if (aggregateType == AggregateType.User) {

storeNames.map(extractUserFeaturesMap)

} else None

}.toSeq

private def getSecondaryKeyFilterPipeOpt(

aggregateType: AggregateType.Value

): Option[TypedPipe[Long]] = {

if (aggregateType == AggregateType.UserAuthor) {

Some(usersActiveSourcePipe)

} else None

}

def userSecondaryKeyPipes: Seq[TypedPipe[(Long, KeyedRecordMap)]] = {

storeRegister.allStores.flatMap { storeConfig =>

val StoreConfig(storeNames, aggregateType, shouldHash) = storeConfig

require(storeMerger.isValidToMerge(storeNames))

if (aggregateType != AggregateType.User) {

storeNames.flatMap { storeName =>

storeConfig.secondaryKeyFeatureOpt

.map { secondaryFeature =>

extractSecondaryTuples(

storeName,

secondaryFeature,

shouldHash,

applyMaxSizeByTypeOpt(aggregateType),

getSecondaryKeyFilterPipeOpt(aggregateType)

)

}

}

} else None

}.toSeq

}

def joinedAggregates: TypedPipe[(Long, MergedRecordsDescriptor)] = {

(userPipes ++ userSecondaryKeyPipes)

.reduce(\_ ++ \_)

.group

.withReducers(numReducersMerge)

.mapGroup {

case (uid, keyedRecordsAndMaps) =>

/\*

\* For every user, partition their records by aggregate type.

\* AggregateType.User should only contain KeyedRecord whereas

\* other aggregate types (with secondary keys) contain KeyedRecordMap.

\*/

val (userRecords, userSecondaryKeyRecords) = keyedRecordsAndMaps.toList

.map { record =>

record match {

case record: KeyedRecord => (record.aggregateType, record)

case record: KeyedRecordMap => (record.aggregateType, record)

}

}

.groupBy(\_.\_1)

.mapValues(\_.map(\_.\_2))

.partition(\_.\_1 == AggregateType.User)

val userAggregateRecordMap: Map[AggregateType.Value, Option[KeyedRecord]] =

userRecords

.asInstanceOf[Map[AggregateType.Value, List[KeyedRecord]]]

.map {

case (aggregateType, keyedRecords) =>

val mergedKeyedRecordOpt = mergeKeyedRecordOpts(keyedRecords.map(Some(\_)): \_\*)

(aggregateType, mergedKeyedRecordOpt)

}

val userSecondaryKeyAggregateRecordOpt: Map[AggregateType.Value, Option[KeyedRecordMap]] =

userSecondaryKeyRecords

.asInstanceOf[Map[AggregateType.Value, List[KeyedRecordMap]]]

.map {

case (aggregateType, keyedRecordMaps) =>

val keyedRecordMapOpt =

keyedRecordMaps.foldLeft(Option.empty[KeyedRecordMap]) {

(mergedRecOpt, nextRec) =>

applyMaxSizeByTypeOpt(aggregateType)

.map { maxSize =>

mergeKeyedRecordMapOpts(mergedRecOpt, Some(nextRec), maxSize)

}.getOrElse {

mergeKeyedRecordMapOpts(mergedRecOpt, Some(nextRec))

}

}

(aggregateType, keyedRecordMapOpt)

}

Iterator(

MergedRecordsDescriptor(

userId = uid,

keyedRecords = userAggregateRecordMap,

keyedRecordMaps = userSecondaryKeyAggregateRecordOpt

)

)

}.toTypedPipe

}

}