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

import com.twitter.ml.api.\_

import com.twitter.ml.api.Feature

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

import com.twitter.scalding.typed.TypedPipe

import com.twitter.scalding.typed.UnsortedGrouped

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

import java.util.{Set => JSet}

import scala.collection.JavaConverters.\_

object SparseBinaryAggregateJoin {

import TypedAggregateGroup.\_

def makeKey(record: DataRecord, joinKeyList: List[Feature[\_]]): String = {

joinKeyList.map {

case sparseKey: Feature.SparseBinary =>

SRichDataRecord(record).getFeatureValue(sparseFeature(sparseKey))

case nonSparseKey: Feature[\_] =>

SRichDataRecord(record).getFeatureValue(nonSparseKey)

}.toString

}

/\*\*

\* @param record Data record to get all possible sparse aggregate keys from

\* @param List of join key features (some can be sparse and some non-sparse)

\* @return A list of string keys to use for joining

\*/

def makeKeyPermutations(record: DataRecord, joinKeyList: List[Feature[\_]]): List[String] = {

val allIdValues = joinKeyList.flatMap {

case sparseKey: Feature.SparseBinary => {

val id = sparseKey.getDenseFeatureId

val valuesOpt = Option(SRichDataRecord(record).getFeatureValue(sparseKey))

.map(\_.asInstanceOf[JSet[String]].asScala.toSet)

valuesOpt.map { (id, \_) }

}

case nonSparseKey: Feature[\_] => {

val id = nonSparseKey.getDenseFeatureId

Option(SRichDataRecord(record).getFeatureValue(nonSparseKey)).map { value =>

(id, Set(value.toString))

}

}

}

sparseBinaryPermutations(allIdValues).toList.map { idValues =>

joinKeyList.map { key => idValues.getOrElse(key.getDenseFeatureId, "") }.toString

}

}

private[this] def mkKeyIndexedAggregates(

joinFeaturesDataSet: DataSetPipe,

joinKeyList: List[Feature[\_]]

): TypedPipe[(String, DataRecord)] =

joinFeaturesDataSet.records

.map { record => (makeKey(record, joinKeyList), record) }

private[this] def mkKeyIndexedInput(

inputDataSet: DataSetPipe,

joinKeyList: List[Feature[\_]]

): TypedPipe[(String, DataRecord)] =

inputDataSet.records

.flatMap { record =>

for {

key <- makeKeyPermutations(record, joinKeyList)

} yield { (key, record) }

}

private[this] def mkKeyIndexedInputWithUniqueId(

inputDataSet: DataSetPipe,

joinKeyList: List[Feature[\_]],

uniqueIdFeatureList: List[Feature[\_]]

): TypedPipe[(String, String)] =

inputDataSet.records

.flatMap { record =>

for {

key <- makeKeyPermutations(record, joinKeyList)

} yield { (key, makeKey(record, uniqueIdFeatureList)) }

}

private[this] def mkRecordIndexedAggregates(

keyIndexedInput: TypedPipe[(String, DataRecord)],

keyIndexedAggregates: TypedPipe[(String, DataRecord)]

): UnsortedGrouped[DataRecord, List[DataRecord]] =

keyIndexedInput

.join(keyIndexedAggregates)

.map { case (\_, (inputRecord, aggregateRecord)) => (inputRecord, aggregateRecord) }

.group

.toList

private[this] def mkRecordIndexedAggregatesWithUniqueId(

keyIndexedInput: TypedPipe[(String, String)],

keyIndexedAggregates: TypedPipe[(String, DataRecord)]

): UnsortedGrouped[String, List[DataRecord]] =

keyIndexedInput

.join(keyIndexedAggregates)

.map { case (\_, (inputId, aggregateRecord)) => (inputId, aggregateRecord) }

.group

.toList

def mkJoinedDataSet(

inputDataSet: DataSetPipe,

joinFeaturesDataSet: DataSetPipe,

recordIndexedAggregates: UnsortedGrouped[DataRecord, List[DataRecord]],

mergePolicy: SparseBinaryMergePolicy

): TypedPipe[DataRecord] =

inputDataSet.records

.map(record => (record, ()))

.leftJoin(recordIndexedAggregates)

.map {

case (inputRecord, (\_, aggregateRecordsOpt)) =>

aggregateRecordsOpt

.map { aggregateRecords =>

mergePolicy.mergeRecord(

inputRecord,

aggregateRecords,

joinFeaturesDataSet.featureContext

)

inputRecord

}

.getOrElse(inputRecord)

}

def mkJoinedDataSetWithUniqueId(

inputDataSet: DataSetPipe,

joinFeaturesDataSet: DataSetPipe,

recordIndexedAggregates: UnsortedGrouped[String, List[DataRecord]],

mergePolicy: SparseBinaryMergePolicy,

uniqueIdFeatureList: List[Feature[\_]]

): TypedPipe[DataRecord] =

inputDataSet.records

.map(record => (makeKey(record, uniqueIdFeatureList), record))

.leftJoin(recordIndexedAggregates)

.map {

case (\_, (inputRecord, aggregateRecordsOpt)) =>

aggregateRecordsOpt

.map { aggregateRecords =>

mergePolicy.mergeRecord(

inputRecord,

aggregateRecords,

joinFeaturesDataSet.featureContext

)

inputRecord

}

.getOrElse(inputRecord)

}

/\*\*

\* If uniqueIdFeatures is non-empty and the join keys include a sparse binary

\* key, the join will use this set of keys as a unique id to reduce

\* memory consumption. You should need this option only for

\* memory-intensive joins to avoid OOM errors.

\*/

def apply(

inputDataSet: DataSetPipe,

joinKeys: Product,

joinFeaturesDataSet: DataSetPipe,

mergePolicy: SparseBinaryMergePolicy = PickFirstRecordPolicy,

uniqueIdFeaturesOpt: Option[Product] = None

): DataSetPipe = {

val joinKeyList = joinKeys.productIterator.toList.asInstanceOf[List[Feature[\_]]]

val sparseBinaryJoinKeySet =

joinKeyList.toSet.filter(\_.getFeatureType() == FeatureType.SPARSE\_BINARY)

val containsSparseBinaryKey = !sparseBinaryJoinKeySet.isEmpty

if (containsSparseBinaryKey) {

val uniqueIdFeatureList = uniqueIdFeaturesOpt

.map(uniqueIdFeatures =>

uniqueIdFeatures.productIterator.toList.asInstanceOf[List[Feature[\_]]])

.getOrElse(List.empty[Feature[\_]])

val keyIndexedAggregates = mkKeyIndexedAggregates(joinFeaturesDataSet, joinKeyList)

val joinedDataSet = if (uniqueIdFeatureList.isEmpty) {

val keyIndexedInput = mkKeyIndexedInput(inputDataSet, joinKeyList)

val recordIndexedAggregates =

mkRecordIndexedAggregates(keyIndexedInput, keyIndexedAggregates)

mkJoinedDataSet(inputDataSet, joinFeaturesDataSet, recordIndexedAggregates, mergePolicy)

} else {

val keyIndexedInput =

mkKeyIndexedInputWithUniqueId(inputDataSet, joinKeyList, uniqueIdFeatureList)

val recordIndexedAggregates =

mkRecordIndexedAggregatesWithUniqueId(keyIndexedInput, keyIndexedAggregates)

mkJoinedDataSetWithUniqueId(

inputDataSet,

joinFeaturesDataSet,

recordIndexedAggregates,

mergePolicy,

uniqueIdFeatureList

)

}

DataSetPipe(

joinedDataSet,

mergePolicy.mergeContext(

inputDataSet.featureContext,

joinFeaturesDataSet.featureContext

)

)

} else {

inputDataSet.joinWithSmaller(joinKeys, joinFeaturesDataSet) { \_.pass }

}

}

}