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

import com.twitter.algebird.Semigroup

import com.twitter.ml.api.DataRecord

import com.twitter.ml.api.DataRecordMerger

import com.twitter.summingbird.Platform

import com.twitter.summingbird.Producer

import com.twitter.summingbird.TailProducer

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

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

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

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

object AggregatesV2Job {

private lazy val merger = new DataRecordMerger

/\*\*

\* Merges all "incremental" records with the same aggregation key

\* into a single record.

\*

\* @param recordsPerKey A set of (AggregationKey, DataRecord) tuples

\* known to share the same AggregationKey

\* @return A single merged datarecord

\*/

def mergeRecords(recordsPerKey: Set[(AggregationKey, DataRecord)]): DataRecord =

recordsPerKey.foldLeft(new DataRecord) {

case (merged: DataRecord, (key: AggregationKey, elem: DataRecord)) => {

merger.merge(merged, elem)

merged

}

}

/\*\*

\* Given a set of aggregates to compute and a datarecord, extract key-value

\* pairs to output to the summingbird store.

\*

\* @param dataRecord input data record

\* @param aggregates set of aggregates to compute

\* @param featureCounters counters to apply to each input data record

\* @return computed aggregates

\*/

def computeAggregates(

dataRecord: DataRecord,

aggregates: Set[TypedAggregateGroup[\_]],

featureCounters: Seq[DataRecordFeatureCounter]

): Map[AggregationKey, DataRecord] = {

val computedAggregates = aggregates

.flatMap(\_.computeAggregateKVPairs(dataRecord))

.groupBy { case (aggregationKey: AggregationKey, \_) => aggregationKey }

.mapValues(mergeRecords)

featureCounters.foreach(counter =>

computedAggregates.map(agg => DataRecordFeatureCounter(counter, agg.\_2)))

computedAggregates

}

/\*\*

\* Util method to apply a filter on containment in an optional set.

\*

\* @param setOptional Optional set of items to check containment in.

\* @param toCheck Item to check if contained in set.

\* @return If the optional set is None, returns true.

\*/

def setFilter[T](setOptional: Option[Set[T]], toCheck: T): Boolean =

setOptional.map(\_.contains(toCheck)).getOrElse(true)

/\*\*

\* Util for filtering a collection of `TypedAggregateGroup`

\*

\* @param aggregates a set of aggregates

\* @param sourceNames Optional filter on which AggregateGroups to process

\* based on the name of the input source.

\* @param storeNames Optional filter on which AggregateGroups to process

\* based on the name of the output store.

\* @return filtered aggregates

\*/

def filterAggregates(

aggregates: Set[TypedAggregateGroup[\_]],

sourceNames: Option[Set[String]],

storeNames: Option[Set[String]]

): Set[TypedAggregateGroup[\_]] =

aggregates

.filter { aggregateGroup =>

val sourceName = aggregateGroup.inputSource.name

val storeName = aggregateGroup.outputStore.name

val containsSource = setFilter(sourceNames, sourceName)

val containsStore = setFilter(storeNames, storeName)

containsSource && containsStore

}

/\*\*

\* The core summingbird job code.

\*

\* For each aggregate in the set passed in, the job

\* processes all datarecords in the input producer

\* stream to generate "incremental" contributions to

\* these aggregates, and emits them grouped by

\* aggregation key so that summingbird can aggregate them.

\*

\* It is important that after applying the sourceNameFilter and storeNameFilter,

\* all the result AggregateGroups share the same startDate, otherwise the job

\* will fail or give invalid results.

\*

\* @param aggregateSet A set of aggregates to compute. All aggregates

\* in this set that pass the sourceNameFilter and storeNameFilter

\* defined below, if any, will be computed.

\* @param aggregateSourceToSummingbird Function that maps from our logical

\* AggregateSource abstraction to the underlying physical summingbird

\* producer of data records to aggregate (e.g. scalding/eventbus source)

\* @param aggregateStoreToSummingbird Function that maps from our logical

\* AggregateStore abstraction to the underlying physical summingbird

\* store to write output aggregate records to (e.g. mahattan for scalding,

\* or memcache for heron)

\* @param featureCounters counters to use with each input DataRecord

\* @return summingbird tail producer

\*/

def generateJobGraph[P <: Platform[P]](

aggregateSet: Set[TypedAggregateGroup[\_]],

aggregateSourceToSummingbird: AggregateSource => Option[Producer[P, DataRecord]],

aggregateStoreToSummingbird: AggregateStore => Option[P#Store[AggregationKey, DataRecord]],

featureCounters: Seq[DataRecordFeatureCounter] = Seq.empty

)(

implicit semigroup: Semigroup[DataRecord]

): TailProducer[P, Any] = {

val tailProducerList: List[TailProducer[P, Any]] = aggregateSet

.groupBy { aggregate => (aggregate.inputSource, aggregate.outputStore) }

.flatMap {

case (

(inputSource: AggregateSource, outputStore: AggregateStore),

aggregatesInThisStore

) => {

val producerOpt = aggregateSourceToSummingbird(inputSource)

val storeOpt = aggregateStoreToSummingbird(outputStore)

(producerOpt, storeOpt) match {

case (Some(producer), Some(store)) =>

Some(

producer

.flatMap(computeAggregates(\_, aggregatesInThisStore, featureCounters))

.name("FLATMAP")

.sumByKey(store)

.name("SUMMER")

)

case \_ => None

}

}

}

.toList

tailProducerList.reduceLeft { (left, right) => left.also(right) }

}

def aggregateNames(aggregateSet: Set[TypedAggregateGroup[\_]]) = {

aggregateSet

.map(typedGroup =>

(

typedGroup.aggregatePrefix,

typedGroup.individualAggregateDescriptors

.flatMap(\_.outputFeatures.map(\_.getFeatureName)).mkString(",")))

}.toMap

}