package com.twitter.simclusters\_v2.summingbird.common

import com.twitter.algebird.DecayedValue

import com.twitter.algebird.Monoid

import com.twitter.algebird.OptionMonoid

import com.twitter.algebird.ScMapMonoid

import com.twitter.algebird\_internal.thriftscala.{DecayedValue => ThriftDecayedValue}

import com.twitter.simclusters\_v2.common.SimClustersEmbedding

import com.twitter.simclusters\_v2.thriftscala.ClustersWithScores

import com.twitter.simclusters\_v2.thriftscala.MultiModelClustersWithScores

import com.twitter.simclusters\_v2.thriftscala.MultiModelTopKTweetsWithScores

import com.twitter.simclusters\_v2.thriftscala.ModelVersion

import com.twitter.simclusters\_v2.thriftscala.MultiModelPersistentSimClustersEmbedding

import com.twitter.simclusters\_v2.thriftscala.PersistentSimClustersEmbedding

import com.twitter.simclusters\_v2.thriftscala.Scores

import com.twitter.simclusters\_v2.thriftscala.SimClustersEmbeddingMetadata

import com.twitter.simclusters\_v2.thriftscala.TopKClustersWithScores

import com.twitter.simclusters\_v2.thriftscala.TopKTweetsWithScores

import com.twitter.simclusters\_v2.thriftscala.{SimClustersEmbedding => ThriftSimClustersEmbedding}

import com.twitter.snowflake.id.SnowflakeId

import scala.collection.mutable

/\*\*

\* Contains various monoids used in the EntityJob

\*/

object Monoids {

class ScoresMonoid(implicit thriftDecayedValueMonoid: ThriftDecayedValueMonoid)

extends Monoid[Scores] {

private val optionalThriftDecayedValueMonoid =

new OptionMonoid[ThriftDecayedValue]()

override val zero: Scores = Scores()

override def plus(x: Scores, y: Scores): Scores = {

Scores(

optionalThriftDecayedValueMonoid.plus(

x.favClusterNormalized8HrHalfLifeScore,

y.favClusterNormalized8HrHalfLifeScore

),

optionalThriftDecayedValueMonoid.plus(

x.followClusterNormalized8HrHalfLifeScore,

y.followClusterNormalized8HrHalfLifeScore

)

)

}

}

class ClustersWithScoresMonoid(implicit scoresMonoid: ScoresMonoid)

extends Monoid[ClustersWithScores] {

private val optionMapMonoid =

new OptionMonoid[collection.Map[Int, Scores]]()(new ScMapMonoid[Int, Scores]())

override val zero: ClustersWithScores = ClustersWithScores()

override def plus(x: ClustersWithScores, y: ClustersWithScores): ClustersWithScores = {

ClustersWithScores(

optionMapMonoid.plus(x.clustersToScore, y.clustersToScore)

)

}

}

class MultiModelClustersWithScoresMonoid(implicit scoresMonoid: ScoresMonoid)

extends Monoid[MultiModelClustersWithScores] {

override val zero: MultiModelClustersWithScores = MultiModelClustersWithScores()

override def plus(

x: MultiModelClustersWithScores,

y: MultiModelClustersWithScores

): MultiModelClustersWithScores = {

// We reuse the logic from the Monoid for the Value here

val clustersWithScoreMonoid = Implicits.clustersWithScoreMonoid

MultiModelClustersWithScores(

MultiModelUtils.mergeTwoMultiModelMaps(

x.multiModelClustersWithScores,

y.multiModelClustersWithScores,

clustersWithScoreMonoid))

}

}

class TopKClustersWithScoresMonoid(

topK: Int,

threshold: Double

)(

implicit thriftDecayedValueMonoid: ThriftDecayedValueMonoid)

extends Monoid[TopKClustersWithScores] {

override val zero: TopKClustersWithScores = TopKClustersWithScores()

override def plus(

x: TopKClustersWithScores,

y: TopKClustersWithScores

): TopKClustersWithScores = {

val mergedFavMap = TopKScoresUtils

.mergeTwoTopKMapWithDecayedValues(

x.topClustersByFavClusterNormalizedScore

.map(\_.mapValues(

\_.favClusterNormalized8HrHalfLifeScore.getOrElse(thriftDecayedValueMonoid.zero))),

y.topClustersByFavClusterNormalizedScore

.map(\_.mapValues(

\_.favClusterNormalized8HrHalfLifeScore.getOrElse(thriftDecayedValueMonoid.zero))),

topK,

threshold

).map(\_.mapValues(decayedValue =>

Scores(favClusterNormalized8HrHalfLifeScore = Some(decayedValue))))

val mergedFollowMap = TopKScoresUtils

.mergeTwoTopKMapWithDecayedValues(

x.topClustersByFollowClusterNormalizedScore

.map(\_.mapValues(

\_.followClusterNormalized8HrHalfLifeScore.getOrElse(thriftDecayedValueMonoid.zero))),

y.topClustersByFollowClusterNormalizedScore

.map(\_.mapValues(

\_.followClusterNormalized8HrHalfLifeScore.getOrElse(thriftDecayedValueMonoid.zero))),

topK,

threshold

).map(\_.mapValues(decayedValue =>

Scores(followClusterNormalized8HrHalfLifeScore = Some(decayedValue))))

TopKClustersWithScores(

mergedFavMap,

mergedFollowMap

)

}

}

class TopKTweetsWithScoresMonoid(

topK: Int,

threshold: Double,

tweetAgeThreshold: Long

)(

implicit thriftDecayedValueMonoid: ThriftDecayedValueMonoid)

extends Monoid[TopKTweetsWithScores] {

override val zero: TopKTweetsWithScores = TopKTweetsWithScores()

override def plus(x: TopKTweetsWithScores, y: TopKTweetsWithScores): TopKTweetsWithScores = {

val oldestTweetId = SnowflakeId.firstIdFor(System.currentTimeMillis() - tweetAgeThreshold)

val mergedFavMap = TopKScoresUtils

.mergeTwoTopKMapWithDecayedValues(

x.topTweetsByFavClusterNormalizedScore

.map(\_.mapValues(

\_.favClusterNormalized8HrHalfLifeScore.getOrElse(thriftDecayedValueMonoid.zero))),

y.topTweetsByFavClusterNormalizedScore

.map(\_.mapValues(

\_.favClusterNormalized8HrHalfLifeScore.getOrElse(thriftDecayedValueMonoid.zero))),

topK,

threshold

).map(\_.filter(\_.\_1 >= oldestTweetId).mapValues(decayedValue =>

Scores(favClusterNormalized8HrHalfLifeScore = Some(decayedValue))))

TopKTweetsWithScores(mergedFavMap, None)

}

}

class MultiModelTopKTweetsWithScoresMonoid(

)(

implicit thriftDecayedValueMonoid: ThriftDecayedValueMonoid)

extends Monoid[MultiModelTopKTweetsWithScores] {

override val zero: MultiModelTopKTweetsWithScores = MultiModelTopKTweetsWithScores()

override def plus(

x: MultiModelTopKTweetsWithScores,

y: MultiModelTopKTweetsWithScores

): MultiModelTopKTweetsWithScores = {

// We reuse the logic from the Monoid for the Value here

val topKTweetsWithScoresMonoid = Implicits.topKTweetsWithScoresMonoid

MultiModelTopKTweetsWithScores(

MultiModelUtils.mergeTwoMultiModelMaps(

x.multiModelTopKTweetsWithScores,

y.multiModelTopKTweetsWithScores,

topKTweetsWithScoresMonoid))

}

}

/\*\*

\* Merge two PersistentSimClustersEmbedding. The latest embedding overwrite the old embedding.

\* The new count equals to the sum of the count.

\*/

class PersistentSimClustersEmbeddingMonoid extends Monoid[PersistentSimClustersEmbedding] {

override val zero: PersistentSimClustersEmbedding = PersistentSimClustersEmbedding(

ThriftSimClustersEmbedding(),

SimClustersEmbeddingMetadata()

)

private val optionLongMonoid = new OptionMonoid[Long]()

override def plus(

x: PersistentSimClustersEmbedding,

y: PersistentSimClustersEmbedding

): PersistentSimClustersEmbedding = {

val latest =

if (x.metadata.updatedAtMs.getOrElse(0L) > y.metadata.updatedAtMs.getOrElse(0L)) x else y

latest.copy(

metadata = latest.metadata.copy(

updatedCount = optionLongMonoid.plus(x.metadata.updatedCount, y.metadata.updatedCount)))

}

}

class MultiModelPersistentSimClustersEmbeddingMonoid

extends Monoid[MultiModelPersistentSimClustersEmbedding] {

override val zero: MultiModelPersistentSimClustersEmbedding =

MultiModelPersistentSimClustersEmbedding(Map[ModelVersion, PersistentSimClustersEmbedding]())

override def plus(

x: MultiModelPersistentSimClustersEmbedding,

y: MultiModelPersistentSimClustersEmbedding

): MultiModelPersistentSimClustersEmbedding = {

val monoid = Implicits.persistentSimClustersEmbeddingMonoid

// PersistentSimClustersEmbeddings is the only required thrift object so we need to wrap it

// in Some

MultiModelUtils.mergeTwoMultiModelMaps(

Some(x.multiModelPersistentSimClustersEmbedding),

Some(y.multiModelPersistentSimClustersEmbedding),

monoid) match {

// clean up the empty embeddings

case Some(res) =>

MultiModelPersistentSimClustersEmbedding(res.flatMap {

// in some cases the list of SimClustersScore is empty, so we want to remove the

// modelVersion from the list of Models for the embedding

case (modelVersion, persistentSimClustersEmbedding) =>

persistentSimClustersEmbedding.embedding.embedding match {

case embedding if embedding.nonEmpty =>

Map(modelVersion -> persistentSimClustersEmbedding)

case \_ =>

None

}

})

case \_ => zero

}

}

}

/\*\*

\* Merge two PersistentSimClustersEmbeddings. The embedding with the longest l2 norm overwrites

\* the other embedding. The new count equals to the sum of the count.

\*/

class PersistentSimClustersEmbeddingLongestL2NormMonoid

extends Monoid[PersistentSimClustersEmbedding] {

override val zero: PersistentSimClustersEmbedding = PersistentSimClustersEmbedding(

ThriftSimClustersEmbedding(),

SimClustersEmbeddingMetadata()

)

override def plus(

x: PersistentSimClustersEmbedding,

y: PersistentSimClustersEmbedding

): PersistentSimClustersEmbedding = {

if (SimClustersEmbedding(x.embedding).l2norm >= SimClustersEmbedding(y.embedding).l2norm) x

else y

}

}

class MultiModelPersistentSimClustersEmbeddingLongestL2NormMonoid

extends Monoid[MultiModelPersistentSimClustersEmbedding] {

override val zero: MultiModelPersistentSimClustersEmbedding =

MultiModelPersistentSimClustersEmbedding(Map[ModelVersion, PersistentSimClustersEmbedding]())

override def plus(

x: MultiModelPersistentSimClustersEmbedding,

y: MultiModelPersistentSimClustersEmbedding

): MultiModelPersistentSimClustersEmbedding = {

val monoid = Implicits.persistentSimClustersEmbeddingLongestL2NormMonoid

MultiModelUtils.mergeTwoMultiModelMaps(

Some(x.multiModelPersistentSimClustersEmbedding),

Some(y.multiModelPersistentSimClustersEmbedding),

monoid) match {

// clean up empty embeddings

case Some(res) =>

MultiModelPersistentSimClustersEmbedding(res.flatMap {

case (modelVersion, persistentSimClustersEmbedding) =>

// in some cases the list of SimClustersScore is empty, so we want to remove the

// modelVersion from the list of Models for the embedding

persistentSimClustersEmbedding.embedding.embedding match {

case embedding if embedding.nonEmpty =>

Map(modelVersion -> persistentSimClustersEmbedding)

case \_ =>

None

}

})

case \_ => zero

}

}

}

object TopKScoresUtils {

/\*\*

\* Function for merging TopK scores with decayed values.

\*

\* This is for use with topk scores where all scores are updated at the same time (i.e. most

\* time-decayed embedding aggregations). Rather than storing individual scores as algebird.DecayedValue

\* and replicating time information for every key, we can store a single timestamp for the entire

\* embedding and replicate the decay logic when processing each score.

\*

\* This should replicate the behaviour of `mergeTwoTopKMapWithDecayedValues`

\*

\* The logic is:

\* - Determine the most recent update and build a DecayedValue for it (decayedValueForLatestTime)

\* - For each (cluster, score), decay the score relative to the time of the most-recently updated embedding

\* - This is a no-op for scores from the most recently-updated embedding, and will scale scores

\* for the older embedding.

\* - Drop any (cluster, score) which are below the `threshold` score

\* - If both input embeddings contribute a score for the same cluster, keep the one with the largest score (after scaling)

\* - Sort (cluster, score) by score and keep the `topK`

\*

\*/

def mergeClusterScoresWithUpdateTimes[Key](

x: Seq[(Key, Double)],

xUpdatedAtMs: Long,

y: Seq[(Key, Double)],

yUpdatedAtMs: Long,

halfLifeMs: Long,

topK: Int,

threshold: Double

): Seq[(Key, Double)] = {

val latestUpdate = math.max(xUpdatedAtMs, yUpdatedAtMs)

val decayedValueForLatestTime = DecayedValue.build(0.0, latestUpdate, halfLifeMs)

val merged = mutable.HashMap[Key, Double]()

x.foreach {

case (key, score) =>

val decayedScore = Implicits.decayedValueMonoid

.plus(

DecayedValue.build(score, xUpdatedAtMs, halfLifeMs),

decayedValueForLatestTime

).value

if (decayedScore > threshold)

merged += key -> decayedScore

}

y.foreach {

case (key, score) =>

val decayedScore = Implicits.decayedValueMonoid

.plus(

DecayedValue.build(score, yUpdatedAtMs, halfLifeMs),

decayedValueForLatestTime

).value

if (decayedScore > threshold)

merged.get(key) match {

case Some(existingValue) =>

if (decayedScore > existingValue)

merged += key -> decayedScore

case None =>

merged += key -> decayedScore

}

}

merged.toSeq

.sortBy(-\_.\_2)

.take(topK)

}

/\*\*

\* Function for merging to TopK map with decayed values.

\*

\* First of all, all the values will be decayed to the latest scaled timestamp to be comparable.

\*

\* If the same key appears at both a and b, the one with larger scaled time (or larger value when

\* their scaled times are same) will be taken. The values smaller than the threshold will be dropped.

\*

\* After merging, if the size is larger than TopK, only scores with topK largest value will be kept.

\*/

def mergeTwoTopKMapWithDecayedValues[T](

a: Option[collection.Map[T, ThriftDecayedValue]],

b: Option[collection.Map[T, ThriftDecayedValue]],

topK: Int,

threshold: Double

)(

implicit thriftDecayedValueMonoid: ThriftDecayedValueMonoid

): Option[collection.Map[T, ThriftDecayedValue]] = {

if (a.isEmpty || a.exists(\_.isEmpty)) {

return b

}

if (b.isEmpty || b.exists(\_.isEmpty)) {

return a

}

val latestScaledTime = (a.get.view ++ b.get.view).map {

case (\_, scores) =>

scores.scaledTime

}.max

val decayedValueWithLatestScaledTime = ThriftDecayedValue(0.0, latestScaledTime)

val merged = mutable.HashMap[T, ThriftDecayedValue]()

a.foreach {

\_.foreach {

case (k, v) =>

// decay the value to latest scaled time

val decayedScores = thriftDecayedValueMonoid

.plus(v, decayedValueWithLatestScaledTime)

// only merge if the value is larger than the threshold

if (decayedScores.value > threshold) {

merged += k -> decayedScores

}

}

}

b.foreach {

\_.foreach {

case (k, v) =>

val decayedScores = thriftDecayedValueMonoid

.plus(v, decayedValueWithLatestScaledTime)

// only merge if the value is larger than the threshold

if (decayedScores.value > threshold) {

if (!merged.contains(k)) {

merged += k -> decayedScores

} else {

// only update if the value is larger than the one already merged

if (decayedScores.value > merged(k).value) {

merged.update(k, decayedScores)

}

}

}

}

}

// add some buffer size (~ 0.2 \* topK) to avoid sorting and taking too frequently

if (merged.size > topK \* 1.2) {

Some(

merged.toSeq

.sortBy { case (\_, scores) => scores.value \* -1 }

.take(topK)

.toMap

)

} else {

Some(merged)

}

}

}

object MultiModelUtils {

/\*\*

\* In order to reduce complexity we use the Monoid for the value to plus two MultiModel maps

\*/

def mergeTwoMultiModelMaps[T](

a: Option[collection.Map[ModelVersion, T]],

b: Option[collection.Map[ModelVersion, T]],

monoid: Monoid[T]

): Option[collection.Map[ModelVersion, T]] = {

(a, b) match {

case (Some(\_), None) => a

case (None, Some(\_)) => b

case (Some(aa), Some(bb)) =>

val res = ModelVersionProfiles.ModelVersionProfiles.foldLeft(Map[ModelVersion, T]()) {

(map, model) =>

map + (model.\_1 -> monoid.plus(

aa.getOrElse(model.\_1, monoid.zero),

bb.getOrElse(model.\_1, monoid.zero)

))

}

Some(res)

case \_ => None

}

}

}

}