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

import com.twitter.bijection.Injection

import com.twitter.bijection.scrooge.CompactScalaCodec

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

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

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

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

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

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

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

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

import com.twitter.storage.client.manhattan.kv.ManhattanKVClientMtlsParams

import com.twitter.storehaus.ReadableStore

import com.twitter.storehaus\_internal.manhattan.ManhattanCluster

import com.twitter.storehaus\_internal.manhattan.Athena

import com.twitter.storehaus\_internal.manhattan.ManhattanRO

import com.twitter.storehaus\_internal.manhattan.ManhattanROConfig

import com.twitter.storehaus\_internal.manhattan.Nash

import com.twitter.storehaus\_internal.util.ApplicationID

import com.twitter.storehaus\_internal.util.DatasetName

import com.twitter.storehaus\_internal.util.HDFSPath

object UserInterestedInReadableStore {

// Clusters whose size is greater than this will not be considered. This is how the using UTEG

// experiment was run (because it could not process such clusters), and we don't have such a

// restriction for the Summingbird/Memcache implementation, but noticing that we aren't scoring

// tweets correctly in the big clusters. The fix for this seems a little involved, so for now

// let's just exclude such clusters.

val MaxClusterSizeForUserInterestedInDataset: Int = 5e6.toInt

val modelVersionToDatasetMap: Map[String, String] = Map(

ModelVersions.Model20M145KDec11 -> "simclusters\_v2\_interested\_in",

ModelVersions.Model20M145KUpdated -> "simclusters\_v2\_interested\_in\_20m\_145k\_updated",

ModelVersions.Model20M145K2020 -> "simclusters\_v2\_interested\_in\_20m\_145k\_2020"

)

// Producer embedding based User InterestedIn.

val modelVersionToDenserDatasetMap: Map[String, String] = Map(

ModelVersions.Model20M145KUpdated -> "simclusters\_v2\_interested\_in\_from\_producer\_embeddings\_model20m145kupdated"

)

val modelVersionToIIAPEDatasetMap: Map[String, String] = Map(

ModelVersions.Model20M145K2020 -> "simclusters\_v2\_interested\_in\_from\_ape\_20m145k2020"

)

val modelVersionToIIKFLiteDatasetMap: Map[String, String] = Map(

ModelVersions.Model20M145K2020 -> "simclusters\_v2\_interested\_in\_lite\_20m\_145k\_2020"

)

val modelVersionToNextInterestedInDatasetMap: Map[String, String] = Map(

ModelVersions.Model20M145K2020 -> "bet\_consumer\_embedding\_v2"

)

val defaultModelVersion: String = ModelVersions.Model20M145KUpdated

val knownModelVersions: String = modelVersionToDatasetMap.keys.mkString(",")

def defaultStoreWithMtls(

mhMtlsParams: ManhattanKVClientMtlsParams,

modelVersion: String = defaultModelVersion

): ReadableStore[UserId, ClustersUserIsInterestedIn] = {

if (!modelVersionToDatasetMap.contains(modelVersion)) {

throw new IllegalArgumentException(

"Unknown model version: " + modelVersion + ". Known model versions: " + knownModelVersions)

}

this.getStore("simclusters\_v2", mhMtlsParams, modelVersionToDatasetMap(modelVersion))

}

def defaultSimClustersEmbeddingStoreWithMtls(

mhMtlsParams: ManhattanKVClientMtlsParams,

embeddingType: EmbeddingType,

modelVersion: ModelVersion

): ReadableStore[SimClustersEmbeddingId, SimClustersEmbedding] = {

defaultStoreWithMtls(mhMtlsParams, ModelVersions.toKnownForModelVersion(modelVersion))

.composeKeyMapping[SimClustersEmbeddingId] {

case SimClustersEmbeddingId(theEmbeddingType, theModelVersion, InternalId.UserId(userId))

if theEmbeddingType == embeddingType && theModelVersion == modelVersion =>

userId

}.mapValues(

toSimClustersEmbedding(\_, embeddingType, Some(MaxClusterSizeForUserInterestedInDataset)))

}

def defaultIIKFLiteStoreWithMtls(

mhMtlsParams: ManhattanKVClientMtlsParams,

modelVersion: String = defaultModelVersion

): ReadableStore[Long, ClustersUserIsInterestedIn] = {

if (!modelVersionToIIKFLiteDatasetMap.contains(modelVersion)) {

throw new IllegalArgumentException(

"Unknown model version: " + modelVersion + ". Known model versions: " + knownModelVersions)

}

getStore("simclusters\_v2", mhMtlsParams, modelVersionToIIKFLiteDatasetMap(modelVersion))

}

def defaultIIPEStoreWithMtls(

mhMtlsParams: ManhattanKVClientMtlsParams,

modelVersion: String = defaultModelVersion

): ReadableStore[Long, ClustersUserIsInterestedIn] = {

if (!modelVersionToDatasetMap.contains(modelVersion)) {

throw new IllegalArgumentException(

"Unknown model version: " + modelVersion + ". Known model versions: " + knownModelVersions)

}

getStore("simclusters\_v2", mhMtlsParams, modelVersionToDenserDatasetMap(modelVersion))

}

def defaultIIAPEStoreWithMtls(

mhMtlsParams: ManhattanKVClientMtlsParams,

modelVersion: String = defaultModelVersion

): ReadableStore[Long, ClustersUserIsInterestedIn] = {

if (!modelVersionToDatasetMap.contains(modelVersion)) {

throw new IllegalArgumentException(

"Unknown model version: " + modelVersion + ". Known model versions: " + knownModelVersions)

}

getStore("simclusters\_v2", mhMtlsParams, modelVersionToIIAPEDatasetMap(modelVersion))

}

def defaultIIPESimClustersEmbeddingStoreWithMtls(

mhMtlsParams: ManhattanKVClientMtlsParams,

embeddingType: EmbeddingType,

modelVersion: ModelVersion

): ReadableStore[SimClustersEmbeddingId, SimClustersEmbedding] = {

defaultIIPEStoreWithMtls(mhMtlsParams, ModelVersions.toKnownForModelVersion(modelVersion))

.composeKeyMapping[SimClustersEmbeddingId] {

case SimClustersEmbeddingId(theEmbeddingType, theModelVersion, InternalId.UserId(userId))

if theEmbeddingType == embeddingType && theModelVersion == modelVersion =>

userId

}.mapValues(toSimClustersEmbedding(\_, embeddingType))

}

def defaultIIAPESimClustersEmbeddingStoreWithMtls(

mhMtlsParams: ManhattanKVClientMtlsParams,

embeddingType: EmbeddingType,

modelVersion: ModelVersion

): ReadableStore[SimClustersEmbeddingId, SimClustersEmbedding] = {

defaultIIAPEStoreWithMtls(mhMtlsParams, ModelVersions.toKnownForModelVersion(modelVersion))

.composeKeyMapping[SimClustersEmbeddingId] {

case SimClustersEmbeddingId(theEmbeddingType, theModelVersion, InternalId.UserId(userId))

if theEmbeddingType == embeddingType && theModelVersion == modelVersion =>

userId

}.mapValues(toSimClustersEmbedding(\_, embeddingType))

}

def defaultNextInterestedInStoreWithMtls(

mhMtlsParams: ManhattanKVClientMtlsParams,

embeddingType: EmbeddingType,

modelVersion: ModelVersion

): ReadableStore[SimClustersEmbeddingId, SimClustersEmbedding] = {

if (!modelVersionToNextInterestedInDatasetMap.contains(

ModelVersions.toKnownForModelVersion(modelVersion))) {

throw new IllegalArgumentException(

"Unknown model version: " + modelVersion + ". Known model versions: " + knownModelVersions)

}

val datasetName = modelVersionToNextInterestedInDatasetMap(

ModelVersions.toKnownForModelVersion(modelVersion))

new SimClustersManhattanReadableStoreForReadWriteDataset(

appId = "kafka\_beam\_sink\_bet\_consumer\_embedding\_prod",

datasetName = datasetName,

label = datasetName,

mtlsParams = mhMtlsParams,

manhattanCluster = Nash

).mapValues(toSimClustersEmbedding(\_, embeddingType))

}

def getWithMtls(

appId: String,

mtlsParams: ManhattanKVClientMtlsParams,

modelVersion: String = defaultModelVersion

): ReadableStore[Long, ClustersUserIsInterestedIn] = {

if (!modelVersionToDatasetMap.contains(modelVersion)) {

throw new IllegalArgumentException(

"Unknown model version: " + modelVersion + ". Known model versions: " + knownModelVersions)

}

this.getStore(appId, mtlsParams, modelVersionToDatasetMap(modelVersion))

}

/\*\*

\* @param appId Manhattan AppId

\* @param mtlsParams MltsParams for s2s Authentication

\*

\* @return ReadableStore of user to cluster interestedIn data set

\*/

def getStore(

appId: String,

mtlsParams: ManhattanKVClientMtlsParams,

datasetName: String,

manhattanCluster: ManhattanCluster = Athena

): ReadableStore[Long, ClustersUserIsInterestedIn] = {

implicit val keyInjection: Injection[Long, Array[Byte]] = Injection.long2BigEndian

implicit val userInterestsCodec: Injection[ClustersUserIsInterestedIn, Array[Byte]] =

CompactScalaCodec(ClustersUserIsInterestedIn)

ManhattanRO.getReadableStoreWithMtls[Long, ClustersUserIsInterestedIn](

ManhattanROConfig(

HDFSPath(""), // not needed

ApplicationID(appId),

DatasetName(datasetName),

manhattanCluster

),

mtlsParams

)

}

/\*\*

\*

\* @param record ClustersUserIsInterestedIn thrift struct from the MH data set

\* @param embeddingType Embedding Type as defined in com.twitter.simclusters\_v2.thriftscala.EmbeddingType

\* @param maxClusterSizeOpt Option param to set max cluster size.

\* We will not filter out clusters based on cluster size if it is None

\* @return

\*/

def toSimClustersEmbedding(

record: ClustersUserIsInterestedIn,

embeddingType: EmbeddingType,

maxClusterSizeOpt: Option[Int] = None

): SimClustersEmbedding = {

val embedding = record.clusterIdToScores

.collect {

case (clusterId, clusterScores) if maxClusterSizeOpt.forall { maxClusterSize =>

clusterScores.numUsersInterestedInThisClusterUpperBound.exists(\_ < maxClusterSize)

} =>

val score = embeddingType match {

case EmbeddingType.FavBasedUserInterestedIn =>

clusterScores.favScore

case EmbeddingType.FollowBasedUserInterestedIn =>

clusterScores.followScore

case EmbeddingType.LogFavBasedUserInterestedIn =>

clusterScores.logFavScore

case EmbeddingType.FavBasedUserInterestedInFromPE =>

clusterScores.favScore

case EmbeddingType.FollowBasedUserInterestedInFromPE =>

clusterScores.followScore

case EmbeddingType.LogFavBasedUserInterestedInFromPE =>

clusterScores.logFavScore

case EmbeddingType.LogFavBasedUserInterestedInFromAPE =>

clusterScores.logFavScore

case EmbeddingType.FollowBasedUserInterestedInFromAPE =>

clusterScores.followScore

case EmbeddingType.UserNextInterestedIn =>

clusterScores.logFavScore

case EmbeddingType.LogFavBasedUserInterestedMaxpoolingAddressBookFromIIAPE =>

clusterScores.logFavScore

case EmbeddingType.LogFavBasedUserInterestedAverageAddressBookFromIIAPE =>

clusterScores.logFavScore

case EmbeddingType.LogFavBasedUserInterestedBooktypeMaxpoolingAddressBookFromIIAPE =>

clusterScores.logFavScore

case EmbeddingType.LogFavBasedUserInterestedLargestDimMaxpoolingAddressBookFromIIAPE =>

clusterScores.logFavScore

case EmbeddingType.LogFavBasedUserInterestedLouvainMaxpoolingAddressBookFromIIAPE =>

clusterScores.logFavScore

case EmbeddingType.LogFavBasedUserInterestedConnectedMaxpoolingAddressBookFromIIAPE =>

clusterScores.logFavScore

case \_ =>

throw new IllegalArgumentException(s"unknown EmbeddingType: $embeddingType")

}

score.map(clusterId -> \_)

}.flatten.toMap

SimClustersEmbedding(embedding)

}

}