package com.twitter.representation\_manager.migration

import com.twitter.bijection.Injection

import com.twitter.bijection.scrooge.BinaryScalaCodec

import com.twitter.contentrecommender.store.ApeEntityEmbeddingStore

import com.twitter.contentrecommender.store.InterestsOptOutStore

import com.twitter.contentrecommender.store.SemanticCoreTopicSeedStore

import com.twitter.contentrecommender.twistly

import com.twitter.conversions.DurationOps.\_

import com.twitter.decider.Decider

import com.twitter.escherbird.util.uttclient.CacheConfigV2

import com.twitter.escherbird.util.uttclient.CachedUttClientV2

import com.twitter.escherbird.util.uttclient.UttClientCacheConfigsV2

import com.twitter.escherbird.utt.strato.thriftscala.Environment

import com.twitter.finagle.ThriftMux

import com.twitter.finagle.memcached.Client

import com.twitter.finagle.mtls.authentication.ServiceIdentifier

import com.twitter.finagle.mtls.client.MtlsStackClient.MtlsThriftMuxClientSyntax

import com.twitter.finagle.mux.ClientDiscardedRequestException

import com.twitter.finagle.service.ReqRep

import com.twitter.finagle.service.ResponseClass

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.finagle.thrift.ClientId

import com.twitter.frigate.common.store.strato.StratoFetchableStore

import com.twitter.frigate.common.util.SeqLongInjection

import com.twitter.hashing.KeyHasher

import com.twitter.hermit.store.common.DeciderableReadableStore

import com.twitter.hermit.store.common.ObservedCachedReadableStore

import com.twitter.hermit.store.common.ObservedMemcachedReadableStore

import com.twitter.hermit.store.common.ObservedReadableStore

import com.twitter.interests.thriftscala.InterestsThriftService

import com.twitter.relevance\_platform.common.injection.LZ4Injection

import com.twitter.relevance\_platform.common.readablestore.ReadableStoreWithTimeout

import com.twitter.representation\_manager.common.RepresentationManagerDecider

import com.twitter.representation\_manager.store.DeciderConstants

import com.twitter.representation\_manager.store.DeciderKey

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

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

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

import com.twitter.simclusters\_v2.stores.SimClustersEmbeddingStore

import com.twitter.simclusters\_v2.summingbird.stores.PersistentTweetEmbeddingStore

import com.twitter.simclusters\_v2.summingbird.stores.ProducerClusterEmbeddingReadableStores

import com.twitter.simclusters\_v2.summingbird.stores.UserInterestedInReadableStore

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

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

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.ModelVersion.Model20m145k2020

import com.twitter.simclusters\_v2.thriftscala.ModelVersion.Model20m145kUpdated

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

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

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

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

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

import com.twitter.storehaus.ReadableStore

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.util.ApplicationID

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

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

import com.twitter.strato.client.Strato

import com.twitter.strato.client.{Client => StratoClient}

import com.twitter.strato.thrift.ScroogeConvImplicits.\_

import com.twitter.tweetypie.util.UserId

import com.twitter.util.Duration

import com.twitter.util.Future

import com.twitter.util.Throw

import com.twitter.util.Timer

import javax.inject.Inject

import javax.inject.Named

import scala.reflect.ClassTag

class LegacyRMS @Inject() (

serviceIdentifier: ServiceIdentifier,

cacheClient: Client,

stats: StatsReceiver,

decider: Decider,

clientId: ClientId,

timer: Timer,

@Named("cacheHashKeyPrefix") val cacheHashKeyPrefix: String = "RMS",

@Named("useContentRecommenderConfiguration") val useContentRecommenderConfiguration: Boolean =

false) {

private val mhMtlsParams: ManhattanKVClientMtlsParams = ManhattanKVClientMtlsParams(

serviceIdentifier)

private val rmsDecider = RepresentationManagerDecider(decider)

val keyHasher: KeyHasher = KeyHasher.FNV1A\_64

private val embeddingCacheKeyBuilder =

SimClustersEmbeddingIdCacheKeyBuilder(keyHasher.hashKey, cacheHashKeyPrefix)

private val statsReceiver = stats.scope("representation\_management")

// Strato client, default timeout = 280ms

val stratoClient: StratoClient =

Strato.client

.withMutualTls(serviceIdentifier)

.build()

// Builds ThriftMux client builder for Content-Recommender service

private def makeThriftClientBuilder(

requestTimeout: Duration

): ThriftMux.Client = {

ThriftMux.client

.withClientId(clientId)

.withMutualTls(serviceIdentifier)

.withRequestTimeout(requestTimeout)

.withStatsReceiver(statsReceiver.scope("clnt"))

.withResponseClassifier {

case ReqRep(\_, Throw(\_: ClientDiscardedRequestException)) => ResponseClass.Ignorable

}

}

private def makeThriftClient[ThriftServiceType: ClassTag](

dest: String,

label: String,

requestTimeout: Duration = 450.milliseconds

): ThriftServiceType = {

makeThriftClientBuilder(requestTimeout)

.build[ThriftServiceType](dest, label)

}

/\*\* \*\*\* SimCluster Embedding Stores \*\*\*\*\*\*/

implicit val simClustersEmbeddingIdInjection: Injection[SimClustersEmbeddingId, Array[Byte]] =

BinaryScalaCodec(SimClustersEmbeddingId)

implicit val simClustersEmbeddingInjection: Injection[ThriftSimClustersEmbedding, Array[Byte]] =

BinaryScalaCodec(ThriftSimClustersEmbedding)

implicit val simClustersMultiEmbeddingInjection: Injection[SimClustersMultiEmbedding, Array[

Byte

]] =

BinaryScalaCodec(SimClustersMultiEmbedding)

implicit val simClustersMultiEmbeddingIdInjection: Injection[SimClustersMultiEmbeddingId, Array[

Byte

]] =

BinaryScalaCodec(SimClustersMultiEmbeddingId)

def getEmbeddingsDataset(

mhMtlsParams: ManhattanKVClientMtlsParams,

datasetName: String

): ReadableStore[SimClustersEmbeddingId, ThriftSimClustersEmbedding] = {

ManhattanRO.getReadableStoreWithMtls[SimClustersEmbeddingId, ThriftSimClustersEmbedding](

ManhattanROConfig(

HDFSPath(""), // not needed

ApplicationID("content\_recommender\_athena"),

DatasetName(datasetName), // this should be correct

Athena

),

mhMtlsParams

)

}

lazy val logFavBasedLongestL2Tweet20M145K2020EmbeddingStore: ReadableStore[

SimClustersEmbeddingId,

SimClustersEmbedding

] = {

val rawStore =

PersistentTweetEmbeddingStore

.longestL2NormTweetEmbeddingStoreManhattan(

mhMtlsParams,

PersistentTweetEmbeddingStore.LogFavBased20m145k2020Dataset,

statsReceiver,

maxLength = 10,

).mapValues(\_.toThrift)

val memcachedStore = ObservedMemcachedReadableStore.fromCacheClient(

backingStore = rawStore,

cacheClient = cacheClient,

ttl = 15.minutes

)(

valueInjection = LZ4Injection.compose(BinaryScalaCodec(ThriftSimClustersEmbedding)),

statsReceiver =

statsReceiver.scope("log\_fav\_based\_longest\_l2\_tweet\_embedding\_20m145k2020\_mem\_cache"),

keyToString = { k =>

s"scez\_l2:${LogFavBasedTweet}\_${ModelVersions.Model20M145K2020}\_$k"

}

)

val inMemoryCacheStore: ReadableStore[SimClustersEmbeddingId, SimClustersEmbedding] =

memcachedStore

.composeKeyMapping[SimClustersEmbeddingId] {

case SimClustersEmbeddingId(

LogFavLongestL2EmbeddingTweet,

Model20m145k2020,

InternalId.TweetId(tweetId)) =>

tweetId

}

.mapValues(SimClustersEmbedding(\_))

ObservedCachedReadableStore.from[SimClustersEmbeddingId, SimClustersEmbedding](

inMemoryCacheStore,

ttl = 12.minute,

maxKeys = 1048575,

cacheName = "log\_fav\_based\_longest\_l2\_tweet\_embedding\_20m145k2020\_cache",

windowSize = 10000L

)(statsReceiver.scope("log\_fav\_based\_longest\_l2\_tweet\_embedding\_20m145k2020\_store"))

}

lazy val logFavBased20M145KUpdatedTweetEmbeddingStore: ReadableStore[

SimClustersEmbeddingId,

SimClustersEmbedding

] = {

val rawStore =

PersistentTweetEmbeddingStore

.mostRecentTweetEmbeddingStoreManhattan(

mhMtlsParams,

PersistentTweetEmbeddingStore.LogFavBased20m145kUpdatedDataset,

statsReceiver

).mapValues(\_.toThrift)

val memcachedStore = ObservedMemcachedReadableStore.fromCacheClient(

backingStore = rawStore,

cacheClient = cacheClient,

ttl = 10.minutes

)(

valueInjection = LZ4Injection.compose(BinaryScalaCodec(ThriftSimClustersEmbedding)),

statsReceiver = statsReceiver.scope("log\_fav\_based\_tweet\_embedding\_mem\_cache"),

keyToString = { k =>

// SimClusters\_embedding\_LZ4/embeddingType\_modelVersion\_tweetId

s"scez:${LogFavBasedTweet}\_${ModelVersions.Model20M145KUpdated}\_$k"

}

)

val inMemoryCacheStore: ReadableStore[SimClustersEmbeddingId, SimClustersEmbedding] = {

memcachedStore

.composeKeyMapping[SimClustersEmbeddingId] {

case SimClustersEmbeddingId(

LogFavBasedTweet,

Model20m145kUpdated,

InternalId.TweetId(tweetId)) =>

tweetId

}

.mapValues(SimClustersEmbedding(\_))

}

ObservedCachedReadableStore.from[SimClustersEmbeddingId, SimClustersEmbedding](

inMemoryCacheStore,

ttl = 5.minute,

maxKeys = 1048575, // 200MB

cacheName = "log\_fav\_based\_tweet\_embedding\_cache",

windowSize = 10000L

)(statsReceiver.scope("log\_fav\_based\_tweet\_embedding\_store"))

}

lazy val logFavBased20M145K2020TweetEmbeddingStore: ReadableStore[

SimClustersEmbeddingId,

SimClustersEmbedding

] = {

val rawStore =

PersistentTweetEmbeddingStore

.mostRecentTweetEmbeddingStoreManhattan(

mhMtlsParams,

PersistentTweetEmbeddingStore.LogFavBased20m145k2020Dataset,

statsReceiver,

maxLength = 10,

).mapValues(\_.toThrift)

val memcachedStore = ObservedMemcachedReadableStore.fromCacheClient(

backingStore = rawStore,

cacheClient = cacheClient,

ttl = 15.minutes

)(

valueInjection = LZ4Injection.compose(BinaryScalaCodec(ThriftSimClustersEmbedding)),

statsReceiver = statsReceiver.scope("log\_fav\_based\_tweet\_embedding\_20m145k2020\_mem\_cache"),

keyToString = { k =>

// SimClusters\_embedding\_LZ4/embeddingType\_modelVersion\_tweetId

s"scez:${LogFavBasedTweet}\_${ModelVersions.Model20M145K2020}\_$k"

}

)

val inMemoryCacheStore: ReadableStore[SimClustersEmbeddingId, SimClustersEmbedding] =

memcachedStore

.composeKeyMapping[SimClustersEmbeddingId] {

case SimClustersEmbeddingId(

LogFavBasedTweet,

Model20m145k2020,

InternalId.TweetId(tweetId)) =>

tweetId

}

.mapValues(SimClustersEmbedding(\_))

ObservedCachedReadableStore.from[SimClustersEmbeddingId, SimClustersEmbedding](

inMemoryCacheStore,

ttl = 12.minute,

maxKeys = 16777215,

cacheName = "log\_fav\_based\_tweet\_embedding\_20m145k2020\_cache",

windowSize = 10000L

)(statsReceiver.scope("log\_fav\_based\_tweet\_embedding\_20m145k2020\_store"))

}

lazy val favBasedTfgTopicEmbedding2020Store: ReadableStore[

SimClustersEmbeddingId,

SimClustersEmbedding

] = {

val stratoStore =

StratoFetchableStore

.withUnitView[SimClustersEmbeddingId, ThriftSimClustersEmbedding](

stratoClient,

"recommendations/simclusters\_v2/embeddings/favBasedTFGTopic20M145K2020")

val truncatedStore = stratoStore.mapValues { embedding =>

SimClustersEmbedding(embedding, truncate = 50)

}

ObservedCachedReadableStore.from(

ObservedReadableStore(truncatedStore)(

statsReceiver.scope("fav\_tfg\_topic\_embedding\_2020\_cache\_backing\_store")),

ttl = 12.hours,

maxKeys = 262143, // 200MB

cacheName = "fav\_tfg\_topic\_embedding\_2020\_cache",

windowSize = 10000L

)(statsReceiver.scope("fav\_tfg\_topic\_embedding\_2020\_cache"))

}

lazy val logFavBasedApe20M145K2020EmbeddingStore: ReadableStore[

SimClustersEmbeddingId,

SimClustersEmbedding

] = {

ObservedReadableStore(

StratoFetchableStore

.withUnitView[SimClustersEmbeddingId, ThriftSimClustersEmbedding](

stratoClient,

"recommendations/simclusters\_v2/embeddings/logFavBasedAPE20M145K2020")

.composeKeyMapping[SimClustersEmbeddingId] {

case SimClustersEmbeddingId(

AggregatableLogFavBasedProducer,

Model20m145k2020,

internalId) =>

SimClustersEmbeddingId(AggregatableLogFavBasedProducer, Model20m145k2020, internalId)

}

.mapValues(embedding => SimClustersEmbedding(embedding, 50))

)(statsReceiver.scope("aggregatable\_producer\_embeddings\_by\_logfav\_score\_2020"))

}

val interestService: InterestsThriftService.MethodPerEndpoint =

makeThriftClient[InterestsThriftService.MethodPerEndpoint](

"/s/interests-thrift-service/interests-thrift-service",

"interests\_thrift\_service"

)

val interestsOptOutStore: InterestsOptOutStore = InterestsOptOutStore(interestService)

// Save 2 ^ 18 UTTs. Promising 100% cache rate

lazy val defaultCacheConfigV2: CacheConfigV2 = CacheConfigV2(262143)

lazy val uttClientCacheConfigsV2: UttClientCacheConfigsV2 = UttClientCacheConfigsV2(

getTaxonomyConfig = defaultCacheConfigV2,

getUttTaxonomyConfig = defaultCacheConfigV2,

getLeafIds = defaultCacheConfigV2,

getLeafUttEntities = defaultCacheConfigV2

)

// CachedUttClient to use StratoClient

lazy val cachedUttClientV2: CachedUttClientV2 = new CachedUttClientV2(

stratoClient = stratoClient,

env = Environment.Prod,

cacheConfigs = uttClientCacheConfigsV2,

statsReceiver = statsReceiver.scope("cached\_utt\_client")

)

lazy val semanticCoreTopicSeedStore: ReadableStore[

SemanticCoreTopicSeedStore.Key,

Seq[UserId]

] = {

/\*

Up to 1000 Long seeds per topic/language = 62.5kb per topic/language (worst case)

Assume ~10k active topic/languages ~= 650MB (worst case)

\*/

val underlying = new SemanticCoreTopicSeedStore(cachedUttClientV2, interestsOptOutStore)(

statsReceiver.scope("semantic\_core\_topic\_seed\_store"))

val memcacheStore = ObservedMemcachedReadableStore.fromCacheClient(

backingStore = underlying,

cacheClient = cacheClient,

ttl = 12.hours

)(

valueInjection = SeqLongInjection,

statsReceiver = statsReceiver.scope("topic\_producer\_seed\_store\_mem\_cache"),

keyToString = { k => s"tpss:${k.entityId}\_${k.languageCode}" }

)

ObservedCachedReadableStore.from[SemanticCoreTopicSeedStore.Key, Seq[UserId]](

store = memcacheStore,

ttl = 6.hours,

maxKeys = 20e3.toInt,

cacheName = "topic\_producer\_seed\_store\_cache",

windowSize = 5000

)(statsReceiver.scope("topic\_producer\_seed\_store\_cache"))

}

lazy val logFavBasedApeEntity20M145K2020EmbeddingStore: ApeEntityEmbeddingStore = {

val apeStore = logFavBasedApe20M145K2020EmbeddingStore.composeKeyMapping[UserId]({ id =>

SimClustersEmbeddingId(

AggregatableLogFavBasedProducer,

Model20m145k2020,

InternalId.UserId(id))

})

new ApeEntityEmbeddingStore(

semanticCoreSeedStore = semanticCoreTopicSeedStore,

aggregatableProducerEmbeddingStore = apeStore,

statsReceiver = statsReceiver.scope("log\_fav\_based\_ape\_entity\_2020\_embedding\_store"))

}

lazy val logFavBasedApeEntity20M145K2020EmbeddingCachedStore: ReadableStore[

SimClustersEmbeddingId,

SimClustersEmbedding

] = {

val truncatedStore =

logFavBasedApeEntity20M145K2020EmbeddingStore.mapValues(\_.truncate(50).toThrift)

val memcachedStore = ObservedMemcachedReadableStore

.fromCacheClient(

backingStore = truncatedStore,

cacheClient = cacheClient,

ttl = 12.hours

)(

valueInjection = LZ4Injection.compose(BinaryScalaCodec(ThriftSimClustersEmbedding)),

statsReceiver = statsReceiver.scope("log\_fav\_based\_ape\_entity\_2020\_embedding\_mem\_cache"),

keyToString = { k => embeddingCacheKeyBuilder.apply(k) }

).mapValues(SimClustersEmbedding(\_))

val inMemoryCachedStore =

ObservedCachedReadableStore.from[SimClustersEmbeddingId, SimClustersEmbedding](

memcachedStore,

ttl = 6.hours,

maxKeys = 262143,

cacheName = "log\_fav\_based\_ape\_entity\_2020\_embedding\_cache",

windowSize = 10000L

)(statsReceiver.scope("log\_fav\_based\_ape\_entity\_2020\_embedding\_cached\_store"))

DeciderableReadableStore(

inMemoryCachedStore,

rmsDecider.deciderGateBuilder.idGateWithHashing[SimClustersEmbeddingId](

DeciderKey.enableLogFavBasedApeEntity20M145K2020EmbeddingCachedStore),

statsReceiver.scope("log\_fav\_based\_ape\_entity\_2020\_embedding\_deciderable\_store")

)

}

lazy val relaxedLogFavBasedApe20M145K2020EmbeddingStore: ReadableStore[

SimClustersEmbeddingId,

SimClustersEmbedding

] = {

ObservedReadableStore(

StratoFetchableStore

.withUnitView[SimClustersEmbeddingId, ThriftSimClustersEmbedding](

stratoClient,

"recommendations/simclusters\_v2/embeddings/logFavBasedAPERelaxedFavEngagementThreshold20M145K2020")

.composeKeyMapping[SimClustersEmbeddingId] {

case SimClustersEmbeddingId(

RelaxedAggregatableLogFavBasedProducer,

Model20m145k2020,

internalId) =>

SimClustersEmbeddingId(

RelaxedAggregatableLogFavBasedProducer,

Model20m145k2020,

internalId)

}

.mapValues(embedding => SimClustersEmbedding(embedding).truncate(50))

)(statsReceiver.scope(

"aggregatable\_producer\_embeddings\_by\_logfav\_score\_relaxed\_fav\_engagement\_threshold\_2020"))

}

lazy val relaxedLogFavBasedApe20M145K2020EmbeddingCachedStore: ReadableStore[

SimClustersEmbeddingId,

SimClustersEmbedding

] = {

val truncatedStore =

relaxedLogFavBasedApe20M145K2020EmbeddingStore.mapValues(\_.truncate(50).toThrift)

val memcachedStore = ObservedMemcachedReadableStore

.fromCacheClient(

backingStore = truncatedStore,

cacheClient = cacheClient,

ttl = 12.hours

)(

valueInjection = LZ4Injection.compose(BinaryScalaCodec(ThriftSimClustersEmbedding)),

statsReceiver =

statsReceiver.scope("relaxed\_log\_fav\_based\_ape\_entity\_2020\_embedding\_mem\_cache"),

keyToString = { k: SimClustersEmbeddingId => embeddingCacheKeyBuilder.apply(k) }

).mapValues(SimClustersEmbedding(\_))

ObservedCachedReadableStore.from[SimClustersEmbeddingId, SimClustersEmbedding](

memcachedStore,

ttl = 6.hours,

maxKeys = 262143,

cacheName = "relaxed\_log\_fav\_based\_ape\_entity\_2020\_embedding\_cache",

windowSize = 10000L

)(statsReceiver.scope("relaxed\_log\_fav\_based\_ape\_entity\_2020\_embedding\_cache\_store"))

}

lazy val favBasedProducer20M145K2020EmbeddingStore: ReadableStore[

SimClustersEmbeddingId,

SimClustersEmbedding

] = {

val underlyingStore = ProducerClusterEmbeddingReadableStores

.getProducerTopKSimClusters2020EmbeddingsStore(

mhMtlsParams

).composeKeyMapping[SimClustersEmbeddingId] {

case SimClustersEmbeddingId(

FavBasedProducer,

Model20m145k2020,

InternalId.UserId(userId)) =>

userId

}.mapValues { topSimClustersWithScore =>

ThriftSimClustersEmbedding(topSimClustersWithScore.topClusters.take(10))

}

// same memcache config as for favBasedUserInterestedIn20M145K2020Store

val memcachedStore = ObservedMemcachedReadableStore

.fromCacheClient(

backingStore = underlyingStore,

cacheClient = cacheClient,

ttl = 24.hours

)(

valueInjection = LZ4Injection.compose(BinaryScalaCodec(ThriftSimClustersEmbedding)),

statsReceiver = statsReceiver.scope("fav\_based\_producer\_embedding\_20M\_145K\_2020\_mem\_cache"),

keyToString = { k => embeddingCacheKeyBuilder.apply(k) }

).mapValues(SimClustersEmbedding(\_))

ObservedCachedReadableStore.from[SimClustersEmbeddingId, SimClustersEmbedding](

memcachedStore,

ttl = 12.hours,

maxKeys = 16777215,

cacheName = "fav\_based\_producer\_embedding\_20M\_145K\_2020\_embedding\_cache",

windowSize = 10000L

)(statsReceiver.scope("fav\_based\_producer\_embedding\_20M\_145K\_2020\_embedding\_store"))

}

// Production

lazy val interestedIn20M145KUpdatedStore: ReadableStore[UserId, ClustersUserIsInterestedIn] = {

UserInterestedInReadableStore.defaultStoreWithMtls(

mhMtlsParams,

modelVersion = ModelVersions.Model20M145KUpdated

)

}

// Production

lazy val interestedIn20M145K2020Store: ReadableStore[UserId, ClustersUserIsInterestedIn] = {

UserInterestedInReadableStore.defaultStoreWithMtls(

mhMtlsParams,

modelVersion = ModelVersions.Model20M145K2020

)

}

// Production

lazy val InterestedInFromPE20M145KUpdatedStore: ReadableStore[

UserId,

ClustersUserIsInterestedIn

] = {

UserInterestedInReadableStore.defaultIIPEStoreWithMtls(

mhMtlsParams,

modelVersion = ModelVersions.Model20M145KUpdated)

}

lazy val simClustersInterestedInStore: ReadableStore[

(UserId, ModelVersion),

ClustersUserIsInterestedIn

] = {

new ReadableStore[(UserId, ModelVersion), ClustersUserIsInterestedIn] {

override def get(k: (UserId, ModelVersion)): Future[Option[ClustersUserIsInterestedIn]] = {

k match {

case (userId, Model20m145kUpdated) =>

interestedIn20M145KUpdatedStore.get(userId)

case (userId, Model20m145k2020) =>

interestedIn20M145K2020Store.get(userId)

case \_ =>

Future.None

}

}

}

}

lazy val simClustersInterestedInFromProducerEmbeddingsStore: ReadableStore[

(UserId, ModelVersion),

ClustersUserIsInterestedIn

] = {

new ReadableStore[(UserId, ModelVersion), ClustersUserIsInterestedIn] {

override def get(k: (UserId, ModelVersion)): Future[Option[ClustersUserIsInterestedIn]] = {

k match {

case (userId, ModelVersion.Model20m145kUpdated) =>

InterestedInFromPE20M145KUpdatedStore.get(userId)

case \_ =>

Future.None

}

}

}

}

lazy val userInterestedInStore =

new twistly.interestedin.EmbeddingStore(

interestedInStore = simClustersInterestedInStore,

interestedInFromProducerEmbeddingStore = simClustersInterestedInFromProducerEmbeddingsStore,

statsReceiver = statsReceiver

)

// Production

lazy val favBasedUserInterestedIn20M145KUpdatedStore: ReadableStore[

SimClustersEmbeddingId,

SimClustersEmbedding

] = {

val underlyingStore =

UserInterestedInReadableStore

.defaultSimClustersEmbeddingStoreWithMtls(

mhMtlsParams,

EmbeddingType.FavBasedUserInterestedIn,

ModelVersion.Model20m145kUpdated)

.mapValues(\_.toThrift)

val memcachedStore = ObservedMemcachedReadableStore

.fromCacheClient(

backingStore = underlyingStore,

cacheClient = cacheClient,

ttl = 12.hours

)(

valueInjection = LZ4Injection.compose(BinaryScalaCodec(ThriftSimClustersEmbedding)),

statsReceiver = statsReceiver.scope("fav\_based\_user\_interested\_in\_mem\_cache"),

keyToString = { k => embeddingCacheKeyBuilder.apply(k) }

).mapValues(SimClustersEmbedding(\_))

ObservedCachedReadableStore.from[SimClustersEmbeddingId, SimClustersEmbedding](

memcachedStore,

ttl = 6.hours,

maxKeys = 262143,

cacheName = "fav\_based\_user\_interested\_in\_cache",

windowSize = 10000L

)(statsReceiver.scope("fav\_based\_user\_interested\_in\_store"))

}

// Production

lazy val LogFavBasedInterestedInFromAPE20M145K2020Store: ReadableStore[

SimClustersEmbeddingId,

SimClustersEmbedding

] = {

val underlyingStore =

UserInterestedInReadableStore

.defaultIIAPESimClustersEmbeddingStoreWithMtls(

mhMtlsParams,

EmbeddingType.LogFavBasedUserInterestedInFromAPE,

ModelVersion.Model20m145k2020)

.mapValues(\_.toThrift)

val memcachedStore = ObservedMemcachedReadableStore

.fromCacheClient(

backingStore = underlyingStore,

cacheClient = cacheClient,

ttl = 12.hours

)(

valueInjection = LZ4Injection.compose(BinaryScalaCodec(ThriftSimClustersEmbedding)),

statsReceiver = statsReceiver.scope("log\_fav\_based\_user\_interested\_in\_from\_ape\_mem\_cache"),

keyToString = { k => embeddingCacheKeyBuilder.apply(k) }

).mapValues(SimClustersEmbedding(\_))

ObservedCachedReadableStore.from[SimClustersEmbeddingId, SimClustersEmbedding](

memcachedStore,

ttl = 6.hours,

maxKeys = 262143,

cacheName = "log\_fav\_based\_user\_interested\_in\_from\_ape\_cache",

windowSize = 10000L

)(statsReceiver.scope("log\_fav\_based\_user\_interested\_in\_from\_ape\_store"))

}

// Production

lazy val FollowBasedInterestedInFromAPE20M145K2020Store: ReadableStore[

SimClustersEmbeddingId,

SimClustersEmbedding

] = {

val underlyingStore =

UserInterestedInReadableStore

.defaultIIAPESimClustersEmbeddingStoreWithMtls(

mhMtlsParams,

EmbeddingType.FollowBasedUserInterestedInFromAPE,

ModelVersion.Model20m145k2020)

.mapValues(\_.toThrift)

val memcachedStore = ObservedMemcachedReadableStore

.fromCacheClient(

backingStore = underlyingStore,

cacheClient = cacheClient,

ttl = 12.hours

)(

valueInjection = LZ4Injection.compose(BinaryScalaCodec(ThriftSimClustersEmbedding)),

statsReceiver = statsReceiver.scope("follow\_based\_user\_interested\_in\_from\_ape\_mem\_cache"),

keyToString = { k => embeddingCacheKeyBuilder.apply(k) }

).mapValues(SimClustersEmbedding(\_))

ObservedCachedReadableStore.from[SimClustersEmbeddingId, SimClustersEmbedding](

memcachedStore,

ttl = 6.hours,

maxKeys = 262143,

cacheName = "follow\_based\_user\_interested\_in\_from\_ape\_cache",

windowSize = 10000L

)(statsReceiver.scope("follow\_based\_user\_interested\_in\_from\_ape\_store"))

}

// production

lazy val favBasedUserInterestedIn20M145K2020Store: ReadableStore[

SimClustersEmbeddingId,

SimClustersEmbedding

] = {

val underlyingStore: ReadableStore[SimClustersEmbeddingId, ThriftSimClustersEmbedding] =

UserInterestedInReadableStore

.defaultSimClustersEmbeddingStoreWithMtls(

mhMtlsParams,

EmbeddingType.FavBasedUserInterestedIn,

ModelVersion.Model20m145k2020).mapValues(\_.toThrift)

ObservedMemcachedReadableStore

.fromCacheClient(

backingStore = underlyingStore,

cacheClient = cacheClient,

ttl = 12.hours

)(

valueInjection = LZ4Injection.compose(BinaryScalaCodec(ThriftSimClustersEmbedding)),

statsReceiver = statsReceiver.scope("fav\_based\_user\_interested\_in\_2020\_mem\_cache"),

keyToString = { k => embeddingCacheKeyBuilder.apply(k) }

).mapValues(SimClustersEmbedding(\_))

}

// Production

lazy val logFavBasedUserInterestedIn20M145K2020Store: ReadableStore[

SimClustersEmbeddingId,

SimClustersEmbedding

] = {

val underlyingStore =

UserInterestedInReadableStore

.defaultSimClustersEmbeddingStoreWithMtls(

mhMtlsParams,

EmbeddingType.LogFavBasedUserInterestedIn,

ModelVersion.Model20m145k2020)

val memcachedStore = ObservedMemcachedReadableStore

.fromCacheClient(

backingStore = underlyingStore.mapValues(\_.toThrift),

cacheClient = cacheClient,

ttl = 12.hours

)(

valueInjection = LZ4Injection.compose(BinaryScalaCodec(ThriftSimClustersEmbedding)),

statsReceiver = statsReceiver.scope("log\_fav\_based\_user\_interested\_in\_2020\_store"),

keyToString = { k => embeddingCacheKeyBuilder.apply(k) }

).mapValues(SimClustersEmbedding(\_))

ObservedCachedReadableStore.from[SimClustersEmbeddingId, SimClustersEmbedding](

memcachedStore,

ttl = 6.hours,

maxKeys = 262143,

cacheName = "log\_fav\_based\_user\_interested\_in\_2020\_cache",

windowSize = 10000L

)(statsReceiver.scope("log\_fav\_based\_user\_interested\_in\_2020\_store"))

}

// Production

lazy val favBasedUserInterestedInFromPE20M145KUpdatedStore: ReadableStore[

SimClustersEmbeddingId,

SimClustersEmbedding

] = {

val underlyingStore =

UserInterestedInReadableStore

.defaultIIPESimClustersEmbeddingStoreWithMtls(

mhMtlsParams,

EmbeddingType.FavBasedUserInterestedInFromPE,

ModelVersion.Model20m145kUpdated)

.mapValues(\_.toThrift)

val memcachedStore = ObservedMemcachedReadableStore

.fromCacheClient(

backingStore = underlyingStore,

cacheClient = cacheClient,

ttl = 12.hours

)(

valueInjection = LZ4Injection.compose(BinaryScalaCodec(ThriftSimClustersEmbedding)),

statsReceiver = statsReceiver.scope("fav\_based\_user\_interested\_in\_from\_pe\_mem\_cache"),

keyToString = { k => embeddingCacheKeyBuilder.apply(k) }

).mapValues(SimClustersEmbedding(\_))

ObservedCachedReadableStore.from[SimClustersEmbeddingId, SimClustersEmbedding](

memcachedStore,

ttl = 6.hours,

maxKeys = 262143,

cacheName = "fav\_based\_user\_interested\_in\_from\_pe\_cache",

windowSize = 10000L

)(statsReceiver.scope("fav\_based\_user\_interested\_in\_from\_pe\_cache"))

}

private val underlyingStores: Map[

(EmbeddingType, ModelVersion),

ReadableStore[SimClustersEmbeddingId, SimClustersEmbedding]

] = Map(

// Tweet Embeddings

(LogFavBasedTweet, Model20m145kUpdated) -> logFavBased20M145KUpdatedTweetEmbeddingStore,

(LogFavBasedTweet, Model20m145k2020) -> logFavBased20M145K2020TweetEmbeddingStore,

(

LogFavLongestL2EmbeddingTweet,

Model20m145k2020) -> logFavBasedLongestL2Tweet20M145K2020EmbeddingStore,

// Entity Embeddings

(FavTfgTopic, Model20m145k2020) -> favBasedTfgTopicEmbedding2020Store,

(

LogFavBasedKgoApeTopic,

Model20m145k2020) -> logFavBasedApeEntity20M145K2020EmbeddingCachedStore,

// KnownFor Embeddings

(FavBasedProducer, Model20m145k2020) -> favBasedProducer20M145K2020EmbeddingStore,

(

RelaxedAggregatableLogFavBasedProducer,

Model20m145k2020) -> relaxedLogFavBasedApe20M145K2020EmbeddingCachedStore,

// InterestedIn Embeddings

(

LogFavBasedUserInterestedInFromAPE,

Model20m145k2020) -> LogFavBasedInterestedInFromAPE20M145K2020Store,

(

FollowBasedUserInterestedInFromAPE,

Model20m145k2020) -> FollowBasedInterestedInFromAPE20M145K2020Store,

(FavBasedUserInterestedIn, Model20m145kUpdated) -> favBasedUserInterestedIn20M145KUpdatedStore,

(FavBasedUserInterestedIn, Model20m145k2020) -> favBasedUserInterestedIn20M145K2020Store,

(LogFavBasedUserInterestedIn, Model20m145k2020) -> logFavBasedUserInterestedIn20M145K2020Store,

(

FavBasedUserInterestedInFromPE,

Model20m145kUpdated) -> favBasedUserInterestedInFromPE20M145KUpdatedStore,

(FilteredUserInterestedIn, Model20m145kUpdated) -> userInterestedInStore,

(FilteredUserInterestedIn, Model20m145k2020) -> userInterestedInStore,

(FilteredUserInterestedInFromPE, Model20m145kUpdated) -> userInterestedInStore,

(UnfilteredUserInterestedIn, Model20m145kUpdated) -> userInterestedInStore,

(UnfilteredUserInterestedIn, Model20m145k2020) -> userInterestedInStore,

)

val simClustersEmbeddingStore: ReadableStore[SimClustersEmbeddingId, SimClustersEmbedding] = {

val underlying: ReadableStore[SimClustersEmbeddingId, SimClustersEmbedding] =

SimClustersEmbeddingStore.buildWithDecider(

underlyingStores = underlyingStores,

decider = rmsDecider.decider,

statsReceiver = statsReceiver.scope("simClusters\_embeddings\_store\_deciderable")

)

val underlyingWithTimeout: ReadableStore[SimClustersEmbeddingId, SimClustersEmbedding] =

new ReadableStoreWithTimeout(

rs = underlying,

decider = rmsDecider.decider,

enableTimeoutDeciderKey = DeciderConstants.enableSimClustersEmbeddingStoreTimeouts,

timeoutValueKey = DeciderConstants.simClustersEmbeddingStoreTimeoutValueMillis,

timer = timer,

statsReceiver = statsReceiver.scope("simClusters\_embedding\_store\_timeouts")

)

ObservedReadableStore(

store = underlyingWithTimeout

)(statsReceiver.scope("simClusters\_embeddings\_store"))

}

}