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

import com.twitter.bijection.Injection

import com.twitter.bijection.scrooge.CompactScalaCodec

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

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

import com.twitter.relevance\_platform.simclustersann.multicluster.ClusterTweetIndexStoreConfig

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

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

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

import com.twitter.simclusters\_v2.summingbird.common.ClientConfigs

import com.twitter.simclusters\_v2.summingbird.common.Configs

import com.twitter.simclusters\_v2.summingbird.common.EntityUtil

import com.twitter.simclusters\_v2.summingbird.common.Implicits

import com.twitter.simclusters\_v2.summingbird.common.Implicits.batcher

import com.twitter.simclusters\_v2.summingbird.common.Implicits.topKTweetsWithScoresCodec

import com.twitter.simclusters\_v2.summingbird.common.Implicits.topKTweetsWithScoresMonoid

import com.twitter.simclusters\_v2.summingbird.common.SimClustersProfile

import com.twitter.simclusters\_v2.summingbird.common.SimClustersProfile.Environment

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

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

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

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

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

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

import com.twitter.storehaus.ReadableStore

import com.twitter.storehaus.Store

import com.twitter.storehaus.algebra.MergeableStore

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

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

import com.twitter.storehaus\_internal.memcache.Memcache

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.Client

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

import com.twitter.summingbird.batch.BatchID

import com.twitter.summingbird.store.ClientStore

import com.twitter.summingbird\_internal.bijection.BatchPairImplicits

import com.twitter.util.Duration

import com.twitter.util.Future

import com.twitter.util.Time

/\*\*

\* Comparing to underlyingStore, this store decays all the values to current timestamp

\*/

case class TopKTweetsForClusterReadableStore(

underlyingStore: ReadableStore[FullClusterId, TopKTweetsWithScores])

extends ReadableStore[FullClusterId, TopKTweetsWithScores] {

override def multiGet[K1 <: FullClusterId](

ks: Set[K1]

): Map[K1, Future[Option[TopKTweetsWithScores]]] = {

val nowInMs = Time.now.inMilliseconds

underlyingStore

.multiGet(ks)

.mapValues { resFuture =>

resFuture.map { resOpt =>

resOpt.map { tweetsWithScores =>

tweetsWithScores.copy(

topTweetsByFavClusterNormalizedScore = EntityUtil.updateScoreWithLatestTimestamp(

tweetsWithScores.topTweetsByFavClusterNormalizedScore,

nowInMs),

topTweetsByFollowClusterNormalizedScore = EntityUtil.updateScoreWithLatestTimestamp(

tweetsWithScores.topTweetsByFollowClusterNormalizedScore,

nowInMs)

)

}

}

}

}

}

object TopKTweetsForClusterReadableStore {

private[summingbird] final lazy val onlineMergeableStore: (

String,

ServiceIdentifier

) => MergeableStore[(FullClusterId, BatchID), TopKTweetsWithScores] = {

(storePath: String, serviceIdentifier: ServiceIdentifier) =>

Memcache.getMemcacheStore[(FullClusterId, BatchID), TopKTweetsWithScores](

ClientConfigs.clusterTopTweetsMemcacheConfig(storePath, serviceIdentifier)

)(

BatchPairImplicits.keyInjection[FullClusterId](Implicits.fullClusterIdCodec),

topKTweetsWithScoresCodec,

topKTweetsWithScoresMonoid

)

}

final lazy val defaultStore: (

String,

ServiceIdentifier

) => ReadableStore[FullClusterId, TopKTweetsWithScores] = {

(storePath: String, serviceIdentifier: ServiceIdentifier) =>

TopKTweetsForClusterReadableStore(

ClientStore(

TopKTweetsForClusterReadableStore.onlineMergeableStore(storePath, serviceIdentifier),

Configs.batchesToKeep

))

}

}

object MultiModelTopKTweetsForClusterReadableStore {

private[simclusters\_v2] def MultiModelTopKTweetsForClusterReadableStore(

stratoClient: Client,

column: String

): Store[Int, MultiModelTopKTweetsWithScores] = {

StratoStore

.withUnitView[Int, MultiModelTopKTweetsWithScores](stratoClient, column)

}

}

case class ClusterKey(

clusterId: ClusterId,

modelVersion: String,

embeddingType: EmbeddingType = EmbeddingType.FavBasedTweet,

halfLife: Duration = Configs.HalfLife) {

lazy val modelVersionThrift: ModelVersion = ModelVersions.toModelVersion(modelVersion)

}

case class TopKTweetsForClusterKeyReadableStore(

proxyMap: Map[(EmbeddingType, String), ReadableStore[FullClusterId, TopKTweetsWithScores]],

halfLife: Duration,

topKTweetsWithScoresToSeq: TopKTweetsWithScores => Seq[(Long, Double)],

maxResult: Option[Int] = None)

extends ReadableStore[ClusterKey, Seq[(Long, Double)]] {

private val modifiedProxyMap = proxyMap.map {

case (typeModelTuple, proxy) =>

typeModelTuple -> proxy.composeKeyMapping { key: ClusterKey =>

FullClusterId(ModelVersions.toModelVersion(typeModelTuple.\_2), key.clusterId)

}

}

override def multiGet[K1 <: ClusterKey](

keys: Set[K1]

): Map[K1, Future[Option[Seq[(Long, Double)]]]] = {

val (validKeys, invalidKeys) = keys.partition { clusterKey =>

proxyMap.contains(

(clusterKey.embeddingType, clusterKey.modelVersion)) && clusterKey.halfLife == halfLife

}

val resultsFuture = validKeys.groupBy(key => (key.embeddingType, key.modelVersion)).flatMap {

case (typeModelTuple, subKeys) =>

modifiedProxyMap(typeModelTuple).multiGet(subKeys)

}

resultsFuture.mapValues { topKTweetsWithScoresFut =>

for (topKTweetsWithScoresOpt <- topKTweetsWithScoresFut) yield {

for {

topKTweetsWithScores <- topKTweetsWithScoresOpt

} yield {

val results = topKTweetsWithScoresToSeq(topKTweetsWithScores)

maxResult match {

case Some(max) =>

results.take(max)

case None =>

results

}

}

}

} ++ invalidKeys.map { key => (key, Future.None) }.toMap

}

}

object TopKTweetsForClusterKeyReadableStore {

implicit val fullClusterIdInjection: Injection[FullClusterId, Array[Byte]] =

CompactScalaCodec(FullClusterId)

// Use Prod cache by default

def defaultProxyMap(

serviceIdentifier: ServiceIdentifier,

): Map[(EmbeddingType, String), ReadableStore[FullClusterId, TopKTweetsWithScores]] =

SimClustersProfile.tweetJobProfileMap(Environment.Prod).mapValues { profile =>

TopKTweetsForClusterReadableStore

.defaultStore(profile.clusterTopKTweetsPath, serviceIdentifier)

}

val defaultHalfLife: Duration = Configs.HalfLife

def defaultStore(

serviceIdentifier: ServiceIdentifier

): ReadableStore[ClusterKey, Seq[(Long, Double)]] =

TopKTweetsForClusterKeyReadableStore(

defaultProxyMap(serviceIdentifier),

defaultHalfLife,

getTopTweetsWithScoresByFavClusterNormalizedScore

)

def storeUsingFollowClusterNormalizedScore(

serviceIdentifier: ServiceIdentifier

): ReadableStore[ClusterKey, Seq[(Long, Double)]] =

TopKTweetsForClusterKeyReadableStore(

defaultProxyMap(serviceIdentifier),

defaultHalfLife,

getTopTweetsWithScoresByFollowClusterNormalizedScore

)

def overrideLimitDefaultStore(

maxResult: Int,

serviceIdentifier: ServiceIdentifier,

): ReadableStore[ClusterKey, Seq[(Long, Double)]] = {

TopKTweetsForClusterKeyReadableStore(

defaultProxyMap(serviceIdentifier),

defaultHalfLife,

getTopTweetsWithScoresByFavClusterNormalizedScore,

Some(maxResult)

)

}

private def getTopTweetsWithScoresByFavClusterNormalizedScore(

topKTweets: TopKTweetsWithScores

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

{

for {

tweetIdWithScores <- topKTweets.topTweetsByFavClusterNormalizedScore

} yield {

(

for {

(tweetId, scores) <- tweetIdWithScores

favClusterNormalized8HrHalfLifeScore <- scores.favClusterNormalized8HrHalfLifeScore

if favClusterNormalized8HrHalfLifeScore.value > 0.0

} yield {

tweetId -> favClusterNormalized8HrHalfLifeScore.value

}

).toSeq.sortBy(-\_.\_2)

}

}.getOrElse(Nil)

}

private def getTopTweetsWithScoresByFollowClusterNormalizedScore(

topKTweets: TopKTweetsWithScores

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

{

for {

tweetIdWithScores <- topKTweets.topTweetsByFollowClusterNormalizedScore

} yield {

(

for {

(tweetId, scores) <- tweetIdWithScores

followClusterNormalized8HrHalfLifeScore <-

scores.followClusterNormalized8HrHalfLifeScore

if followClusterNormalized8HrHalfLifeScore.value > 0.0

} yield {

tweetId -> followClusterNormalized8HrHalfLifeScore.value

}

).toSeq.sortBy(-\_.\_2)

}

}.getOrElse(Nil)

}

def getClusterToTopKTweetsStoreFromManhattanRO(

maxResults: Int,

manhattanConfig: ClusterTweetIndexStoreConfig.Manhattan,

serviceIdentifier: ServiceIdentifier,

): ReadableStore[ClusterKey, Seq[(TweetId, Double)]] = {

ManhattanRO

.getReadableStoreWithMtls[FullClusterId, TopKTweetsWithScores](

ManhattanROConfig(

HDFSPath(""),

ApplicationID(manhattanConfig.applicationID),

DatasetName(manhattanConfig.datasetName),

manhattanConfig.manhattanCluster

),

ManhattanKVClientMtlsParams(serviceIdentifier)

).composeKeyMapping[ClusterKey] { clusterKey =>

FullClusterId(

modelVersion = ModelVersions.toModelVersion(clusterKey.modelVersion),

clusterId = clusterKey.clusterId

)

}.mapValues { topKTweetsWithScores =>

// Only return maxResults tweets for each cluster Id

getTopTweetsWithScoresByFavClusterNormalizedScore(topKTweetsWithScores).take(maxResults)

}

}

def getClusterToTopKTweetsStoreFromMemCache(

maxResults: Int,

memCacheConfig: ClusterTweetIndexStoreConfig.Memcached,

serviceIdentifier: ServiceIdentifier,

): ReadableStore[ClusterKey, Seq[(TweetId, Double)]] = {

TopKTweetsForClusterReadableStore(

ClientStore(

TopKTweetsForClusterReadableStore

.onlineMergeableStore(memCacheConfig.memcachedDest, serviceIdentifier),

Configs.batchesToKeep

))

.composeKeyMapping[ClusterKey] { clusterKey =>

FullClusterId(

modelVersion = ModelVersions.toModelVersion(clusterKey.modelVersion),

clusterId = clusterKey.clusterId

)

}.mapValues { topKTweetsWithScores =>

// Only return maxResults tweets for each cluster Id

getTopTweetsWithScoresByFavClusterNormalizedScore(topKTweetsWithScores).take(maxResults)

}

}

}