package com.twitter.simclustersann.modules

import com.google.inject.Provides

import com.twitter.conversions.DurationOps.\_

import com.twitter.decider.Decider

import com.twitter.finagle.memcached.Client

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

import com.twitter.finagle.stats.StatsReceiver

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

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

import com.twitter.inject.TwitterModule

import com.twitter.inject.annotations.Flag

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

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

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

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.stores.ClusterKey

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

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

import com.twitter.simclustersann.common.FlagNames

import com.twitter.storehaus.ReadableStore

import javax.inject.Singleton

object ClusterTweetIndexProviderModule extends TwitterModule {

@Singleton

@Provides

// Provides ClusterTweetIndex Store based on different maxResults settings on the same store

// Create a different provider if index is in a different store

def providesClusterTweetIndex(

@Flag(FlagNames.MaxTopTweetPerCluster) maxTopTweetPerCluster: Int,

@Flag(FlagNames.CacheAsyncUpdate) asyncUpdate: Boolean,

clusterConfig: ClusterConfig,

serviceIdentifier: ServiceIdentifier,

stats: StatsReceiver,

decider: Decider,

simClustersANNCacheClient: Client

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

// Build the underling cluster-to-tweet store

val topTweetsForClusterStore = clusterConfig.clusterTweetIndexStoreConfig match {

// If the config returns Manhattan tweet index config, we read from a RO MH store

case manhattanConfig: ClusterTweetIndexStoreConfig.Manhattan =>

TopKTweetsForClusterKeyReadableStore.getClusterToTopKTweetsStoreFromManhattanRO(

maxTopTweetPerCluster,

manhattanConfig,

serviceIdentifier)

case memCacheConfig: ClusterTweetIndexStoreConfig.Memcached =>

TopKTweetsForClusterKeyReadableStore.getClusterToTopKTweetsStoreFromMemCache(

maxTopTweetPerCluster,

memCacheConfig,

serviceIdentifier)

case \_ =>

// Bad instance

ReadableStore.empty

}

val embeddingType: EmbeddingType = clusterConfig.candidateTweetEmbeddingType

val modelVersion: String = ModelVersions.toKnownForModelVersion(clusterConfig.modelVersion)

val store: ReadableStore[ClusterId, Seq[(TweetId, Double)]] =

topTweetsForClusterStore.composeKeyMapping { id: ClusterId =>

ClusterKey(id, modelVersion, embeddingType)

}

val memcachedTopTweetsForClusterStore =

ObservedMemcachedReadableStore.fromCacheClient(

backingStore = store,

cacheClient = simClustersANNCacheClient,

ttl = 15.minutes,

asyncUpdate = asyncUpdate

)(

valueInjection = LZ4Injection.compose(SeqObjectInjection[(Long, Double)]()),

statsReceiver = stats.scope("cluster\_tweet\_index\_mem\_cache"),

keyToString = { k =>

// prod cache key : SimClusters\_LZ4/cluster\_to\_tweet/clusterId\_embeddingType\_modelVersion

s"scz:c2t:${k}\_${embeddingType}\_${modelVersion}\_$maxTopTweetPerCluster"

}

)

val cachedStore: ReadableStore[ClusterId, Seq[(TweetId, Double)]] = {

ObservedCachedReadableStore.from[ClusterId, Seq[(TweetId, Double)]](

memcachedTopTweetsForClusterStore,

ttl = 10.minute,

maxKeys = 150000,

cacheName = "cluster\_tweet\_index\_cache",

windowSize = 10000L

)(stats.scope("cluster\_tweet\_index\_store"))

}

cachedStore

}

}