package com.twitter.cr\_mixer.similarity\_engine

import com.twitter.cr\_mixer.param.decider.CrMixerDecider

import com.twitter.cr\_mixer.thriftscala.SimilarityEngineType

import com.twitter.finagle.GlobalRequestTimeoutException

import com.twitter.finagle.mux.ClientDiscardedRequestException

import com.twitter.finagle.memcached.Client

import com.twitter.finagle.mux.ServerApplicationError

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.frigate.common.util.StatsUtil

import com.twitter.hashing.KeyHasher

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

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

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

import com.twitter.storehaus.ReadableStore

import com.twitter.timelines.configapi.FSParam

import com.twitter.timelines.configapi.Params

import com.twitter.util.Duration

import com.twitter.util.Future

import com.twitter.util.TimeoutException

import com.twitter.util.logging.Logging

import org.apache.thrift.TApplicationException

/\*\*

\* A SimilarityEngine is a wrapper which, given a [[Query]], returns a list of [[Candidate]]

\* The main purposes of a SimilarityEngine is to provide a consistent interface for candidate

\* generation logic, and provides default functions, including:

\* - Identification

\* - Observability

\* - Timeout settings

\* - Exception Handling

\* - Gating by Deciders & FeatureSwitch settings

\* - (coming soon): Dark traffic

\*

\* Note:

\* A SimilarityEngine by itself is NOT meant to be cacheable.

\* Caching should be implemented in the underlying ReadableStore that provides the [[Candidate]]s

\*

\* Please keep extension of this class local this directory only

\*

\*/

trait SimilarityEngine[Query, Candidate] {

/\*\*

\* Uniquely identifies a similarity engine.

\* Avoid using the same engine type for more than one engine, it will cause stats to double count

\*/

private[similarity\_engine] def identifier: SimilarityEngineType

def getCandidates(query: Query): Future[Option[Seq[Candidate]]]

}

object SimilarityEngine extends Logging {

case class SimilarityEngineConfig(

timeout: Duration,

gatingConfig: GatingConfig)

/\*\*

\* Controls for whether or not this Engine is enabled.

\* In our previous design, we were expecting a Sim Engine will only take one set of Params,

\* and that’s why we decided to have GatingConfig and the EnableFeatureSwitch in the trait.

\* However, we now have two candidate generation pipelines: Tweet Rec, Related Tweets

\* and they are now having their own set of Params, but EnableFeatureSwitch can only put in 1 fixed value.

\* We need some further refactor work to make it more flexible.

\*

\* @param deciderConfig Gate the Engine by a decider. If specified,

\* @param enableFeatureSwitch. DO NOT USE IT FOR NOW. It needs some refactorting. Please set it to None (SD-20268)

\*/

case class GatingConfig(

deciderConfig: Option[DeciderConfig],

enableFeatureSwitch: Option[

FSParam[Boolean]

]) // Do NOT use the enableFeatureSwitch. It needs some refactoring.

case class DeciderConfig(

decider: CrMixerDecider,

deciderString: String)

case class MemCacheConfig[K](

cacheClient: Client,

ttl: Duration,

asyncUpdate: Boolean = false,

keyToString: K => String)

private[similarity\_engine] def isEnabled(

params: Params,

gatingConfig: GatingConfig

): Boolean = {

val enabledByDecider =

gatingConfig.deciderConfig.forall { config =>

config.decider.isAvailable(config.deciderString)

}

val enabledByFS = gatingConfig.enableFeatureSwitch.forall(params.apply)

enabledByDecider && enabledByFS

}

// Default key hasher for memcache keys

val keyHasher: KeyHasher = KeyHasher.FNV1A\_64

/\*\*

\* Add a MemCache wrapper to a ReadableStore with a preset key and value injection functions

\* Note: The [[Query]] object needs to be cacheable,

\* i.e. it cannot be a runtime objects or complex objects, for example, configapi.Params

\*

\* @param underlyingStore un-cached store implementation

\* @param keyPrefix a prefix differentiates 2 stores if they share the same key space.

\* e.x. 2 implementations of ReadableStore[UserId, Seq[Candidiate] ]

\* can use prefix "store\_v1", "store\_v2"

\* @return A ReadableStore with a MemCache wrapper

\*/

private[similarity\_engine] def addMemCache[Query, Candidate <: Serializable](

underlyingStore: ReadableStore[Query, Seq[Candidate]],

memCacheConfig: MemCacheConfig[Query],

keyPrefix: Option[String] = None,

statsReceiver: StatsReceiver

): ReadableStore[Query, Seq[Candidate]] = {

val prefix = keyPrefix.getOrElse("")

ObservedMemcachedReadableStore.fromCacheClient[Query, Seq[Candidate]](

backingStore = underlyingStore,

cacheClient = memCacheConfig.cacheClient,

ttl = memCacheConfig.ttl,

asyncUpdate = memCacheConfig.asyncUpdate,

)(

valueInjection = LZ4Injection.compose(SeqObjectInjection[Candidate]()),

keyToString = { k: Query => s"CRMixer:$prefix${memCacheConfig.keyToString(k)}" },

statsReceiver = statsReceiver

)

}

private val timer = com.twitter.finagle.util.DefaultTimer

/\*\*

\* Applies runtime configs, like stats, timeouts, exception handling, onto fn

\*/

private[similarity\_engine] def getFromFn[Query, Candidate](

fn: Query => Future[Option[Seq[Candidate]]],

storeQuery: Query,

engineConfig: SimilarityEngineConfig,

params: Params,

scopedStats: StatsReceiver

): Future[Option[Seq[Candidate]]] = {

if (isEnabled(params, engineConfig.gatingConfig)) {

scopedStats.counter("gate\_enabled").incr()

StatsUtil

.trackOptionItemsStats(scopedStats) {

fn.apply(storeQuery).raiseWithin(engineConfig.timeout)(timer)

}

.rescue {

case \_: TimeoutException | \_: GlobalRequestTimeoutException | \_: TApplicationException |

\_: ClientDiscardedRequestException |

\_: ServerApplicationError // TApplicationException inside

=>

debug("Failed to fetch. request aborted or timed out")

Future.None

case e =>

error("Failed to fetch. request aborted or timed out", e)

Future.None

}

} else {

scopedStats.counter("gate\_disabled").incr()

Future.None

}

}

}