package com.twitter.simclusters\_v2.stores

import com.twitter.decider.Decider

import com.twitter.finagle.stats.StatsReceiver

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

import com.twitter.servo.decider.DeciderKeyEnum

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

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

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

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

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

import com.twitter.storehaus.ReadableStore

import com.twitter.util.Future

/\*\*

\* Facade of all SimClusters Embedding Store.

\* Provide a uniform access layer for all kind of SimClusters Embedding.

\*/

case class SimClustersEmbeddingStore(

stores: Map[

(EmbeddingType, ModelVersion),

ReadableStore[SimClustersEmbeddingId, SimClustersEmbedding]

]) extends ReadableStore[SimClustersEmbeddingId, SimClustersEmbedding] {

private val lookupStores =

stores

.groupBy(\_.\_1.\_1).mapValues(\_.map {

case ((\_, modelVersion), store) =>

modelVersion -> store

})

override def get(k: SimClustersEmbeddingId): Future[Option[SimClustersEmbedding]] = {

findStore(k) match {

case Some(store) => store.get(k)

case None => Future.None

}

}

// Override the multiGet for better batch performance.

override def multiGet[K1 <: SimClustersEmbeddingId](

ks: Set[K1]

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

if (ks.isEmpty) {

Map.empty

} else {

val head = ks.head

val notSameType =

ks.exists(k => k.embeddingType != head.embeddingType || k.modelVersion != head.modelVersion)

if (!notSameType) {

findStore(head) match {

case Some(store) => store.multiGet(ks)

case None => ks.map(\_ -> Future.None).toMap

}

} else {

// Generate a large amount temp objects.

// For better performance, avoid querying the multiGet with more than one kind of embedding

ks.groupBy(id => (id.embeddingType, id.modelVersion)).flatMap {

case ((\_, \_), ks) =>

findStore(ks.head) match {

case Some(store) => store.multiGet(ks)

case None => ks.map(\_ -> Future.None).toMap

}

}

}

}

}

private def findStore(

id: SimClustersEmbeddingId

): Option[ReadableStore[SimClustersEmbeddingId, SimClustersEmbedding]] = {

lookupStores.get(id.embeddingType).flatMap(\_.get(id.modelVersion))

}

}

object SimClustersEmbeddingStore {

/\*

Build a SimClustersEmbeddingStore which wraps all stores in DeciderableReadableStore

\*/

def buildWithDecider(

underlyingStores: Map[

(EmbeddingType, ModelVersion),

ReadableStore[SimClustersEmbeddingId, SimClustersEmbedding]

],

decider: Decider,

statsReceiver: StatsReceiver

): ReadableStore[SimClustersEmbeddingId, SimClustersEmbedding] = {

// To allow for lazy adding of decider config to enable / disable stores, if a value is not found

// fall back on returning true (equivalent to availability of 10000)

// This overrides default availability of 0 when not decider value is not found

val deciderGateBuilder = new DeciderGateBuilderWithIdHashing(decider.orElse(Decider.True))

val deciderKeyEnum = new DeciderKeyEnum {

underlyingStores.keySet.map(key => Value(s"enable\_${key.\_1.name}\_${key.\_2.name}"))

}

def wrapStore(

embeddingType: EmbeddingType,

modelVersion: ModelVersion,

store: ReadableStore[SimClustersEmbeddingId, SimClustersEmbedding]

): ReadableStore[SimClustersEmbeddingId, SimClustersEmbedding] = {

val gate = deciderGateBuilder.idGateWithHashing[SimClustersEmbeddingId](

deciderKeyEnum.withName(s"enable\_${embeddingType.name}\_${modelVersion.name}"))

DeciderableReadableStore(

underlying = store,

gate = gate,

statsReceiver = statsReceiver.scope(embeddingType.name, modelVersion.name)

)

}

val stores = underlyingStores.map {

case ((embeddingType, modelVersion), store) =>

(embeddingType, modelVersion) -> wrapStore(embeddingType, modelVersion, store)

}

new SimClustersEmbeddingStore(stores = stores)

}

}