package com.twitter.simclusters\_v2.score

import com.twitter.finagle.stats.BroadcastStatsReceiver

import com.twitter.finagle.stats.StatsReceiver

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

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

import com.twitter.simclusters\_v2.thriftscala.{ScoreId => ThriftScoreId}

import com.twitter.simclusters\_v2.thriftscala.{Score => ThriftScore}

import com.twitter.storehaus.ReadableStore

import com.twitter.util.Future

/\*\*

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

\* @param readableStores readable stores indexed by the ScoringAlgorithm they implement

\*/

class ScoreFacadeStore private (

stores: Map[ScoringAlgorithm, ReadableStore[ThriftScoreId, ThriftScore]])

extends ReadableStore[ThriftScoreId, ThriftScore] {

override def get(k: ThriftScoreId): Future[Option[ThriftScore]] = {

findStore(k).get(k)

}

// Override the multiGet for better batch performance.

override def multiGet[K1 <: ThriftScoreId](ks: Set[K1]): Map[K1, Future[Option[ThriftScore]]] = {

if (ks.isEmpty) {

Map.empty

} else {

val head = ks.head

val notSameType = ks.exists(k => k.algorithm != head.algorithm)

if (!notSameType) {

findStore(head).multiGet(ks)

} 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.algorithm).flatMap {

case (\_, ks) =>

findStore(ks.head).multiGet(ks)

}

}

}

}

// If not store mapping, fast return a IllegalArgumentException.

private def findStore(id: ThriftScoreId): ReadableStore[ThriftScoreId, ThriftScore] = {

stores.get(id.algorithm) match {

case Some(store) => store

case None =>

throw new IllegalArgumentException(s"The Scoring Algorithm ${id.algorithm} doesn't exist.")

}

}

}

object ScoreFacadeStore {

/\*

Build a ScoreFacadeStore which exposes stats for all requests (under "all") and per scoring algorithm:

score\_facade\_store/all/<observed readable store metrics for all requests>

score\_facade\_store/<scoring algorithm>/<observed readable store metrics for this algorithm's requests>

Stores in aggregatedStores may reference stores in readableStores. An instance of ScoreFacadeStore

is passed to them after instantiation.

\*/

def buildWithMetrics(

readableStores: Map[ScoringAlgorithm, ReadableStore[ThriftScoreId, ThriftScore]],

aggregatedStores: Map[ScoringAlgorithm, AggregatedScoreStore],

statsReceiver: StatsReceiver

) = {

val scopedStatsReceiver = statsReceiver.scope("score\_facade\_store")

def wrapStore(

scoringAlgorithm: ScoringAlgorithm,

store: ReadableStore[ThriftScoreId, ThriftScore]

): ReadableStore[ThriftScoreId, ThriftScore] = {

val sr = BroadcastStatsReceiver(

Seq(

scopedStatsReceiver.scope("all"),

scopedStatsReceiver.scope(scoringAlgorithm.name)

))

ObservedReadableStore(store)(sr)

}

val stores = (readableStores ++ aggregatedStores).map {

case (algo, store) => algo -> wrapStore(algo, store)

}

val store = new ScoreFacadeStore(stores = stores)

/\*

AggregatedScores aggregate scores from multiple non-aggregated stores. They access these via the

ScoreFacadeStore itself, and therefore must be passed an instance of it after it has been

constructed.

\*/

assert(

readableStores.keySet.forall(algorithm => !aggregatedStores.keySet.contains(algorithm)),

"Keys for stores are disjoint")

aggregatedStores.values.foreach(\_.set(store))

store

}

}