package com.twitter.simclusters\_v2.score

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

import com.twitter.storehaus.ReadableStore

import com.twitter.util.Future

/\*\*

\* A Score Store is a readableStore with ScoreId as Key and Score as the Value.

\* It also needs to include the algorithm type.

\* A algorithm type should only be used by one Score Store in the application.

\*/

trait ScoreStore[K <: ScoreId] extends ReadableStore[K, Score] {

def fromThriftScoreId: ThriftScoreId => K

// Convert to a Thrift version.

def toThriftStore: ReadableStore[ThriftScoreId, ThriftScore] = {

this

.composeKeyMapping[ThriftScoreId](fromThriftScoreId)

.mapValues(\_.toThrift)

}

}

/\*\*

\* A generic Pairwise Score store.

\* Requires provide both left and right side feature hydration.

\*/

trait PairScoreStore[K <: PairScoreId, K1, K2, V1, V2] extends ScoreStore[K] {

def compositeKey1: K => K1

def compositeKey2: K => K2

// Left side feature hydration

def underlyingStore1: ReadableStore[K1, V1]

// Right side feature hydration

def underlyingStore2: ReadableStore[K2, V2]

def score: (V1, V2) => Future[Option[Double]]

override def get(k: K): Future[Option[Score]] = {

for {

vs <-

Future.join(underlyingStore1.get(compositeKey1(k)), underlyingStore2.get(compositeKey2(k)))

v <- vs match {

case (Some(v1), Some(v2)) =>

score(v1, v2)

case \_ =>

Future.None

}

} yield {

v.map(buildScore)

}

}

override def multiGet[KK <: K](ks: Set[KK]): Map[KK, Future[Option[Score]]] = {

val v1Map = underlyingStore1.multiGet(ks.map { k => compositeKey1(k) })

val v2Map = underlyingStore2.multiGet(ks.map { k => compositeKey2(k) })

ks.map { k =>

k -> Future.join(v1Map(compositeKey1(k)), v2Map(compositeKey2(k))).flatMap {

case (Some(v1), Some(v2)) =>

score(v1, v2).map(\_.map(buildScore))

case \_ =>

Future.value(None)

}

}.toMap

}

private def buildScore(v: Double): Score = Score(v)

}