package com.twitter.servo.repository

import com.twitter.util.{Future, Try}

object KeyValueRepository {

/\*\*

\* Builds a KeyValueRepository that returns KeyValueResults in which all keys failed with the

\* provided Throwable.

\*/

def alwaysFailing[Q <: Seq[K], K, V](failure: Throwable): KeyValueRepository[Q, K, V] =

(query: Q) =>

Future.value(

KeyValueResult[K, V](

failed = query map { \_ -> failure } toMap

)

)

/\*\*

\* Builds an immutable KeyValueRepository

\*/

def apply[K, V](data: Map[K, Try[V]]): KeyValueRepository[Seq[K], K, V] =

new ImmutableKeyValueRepository(data)

/\*\*

\* Sets up a mapReduce type operation on a KeyValueRepository where the query mapping function

\* breaks the query up into smaller chunks, and the reducing function is just KeyValueResult.sum.

\*/

def chunked[Q, K, V](

repo: KeyValueRepository[Q, K, V],

chunker: Q => Seq[Q]

): KeyValueRepository[Q, K, V] =

Repository.mapReduced(repo, chunker, KeyValueResult.sum[K, V])

/\*\*

\* Wraps a KeyValueRepository with stats recording functionality.

\*/

def observed[Q, K, V](

repo: KeyValueRepository[Q, K, V],

observer: RepositoryObserver,

querySize: Q => Int

): KeyValueRepository[Q, K, V] =

query => {

observer.time(querySize(query)) {

repo(query).respond(observer.observeKeyValueResult)

}

}

/\*\*

\* Creates a new KeyValueRepository that dispatches to onTrueRepo if the key

\* predicate returns true, dispatches to onFalseRepo otherwise.

\*/

def selected[Q <: Seq[K], K, V](

select: K => Boolean,

onTrueRepo: KeyValueRepository[Q, K, V],

onFalseRepo: KeyValueRepository[Q, K, V],

queryBuilder: SubqueryBuilder[Q, K]

): KeyValueRepository[Q, K, V] = selectedByQuery(

predicateFactory = \_ => select,

onTrueRepo = onTrueRepo,

onFalseRepo = onFalseRepo,

queryBuilder = queryBuilder

)

/\*\*

\* Creates a new KeyValueRepository that uses predicateFactory to create a key predicate, then

\* dispatches to onTrueRepo if the key predicate returns true, dispatches to onFalseRepo

\* otherwise.

\*/

def selectedByQuery[Q <: Seq[K], K, V](

predicateFactory: Q => (K => Boolean),

onTrueRepo: KeyValueRepository[Q, K, V],

onFalseRepo: KeyValueRepository[Q, K, V],

queryBuilder: SubqueryBuilder[Q, K]

): KeyValueRepository[Q, K, V] = {

val queryIsEmpty = (q: Q) => q.isEmpty

val r1 = shortCircuitEmpty(queryIsEmpty)(onTrueRepo)

val r2 = shortCircuitEmpty(queryIsEmpty)(onFalseRepo)

(query: Q) => {

val predicate = predicateFactory(query)

val (q1, q2) = query.partition(predicate)

val futureRst1 = r1(queryBuilder(q1, query))

val futureRst2 = r2(queryBuilder(q2, query))

for {

r1 <- futureRst1

r2 <- futureRst2

} yield r1 ++ r2

}

}

/\*\*

\* Creates a new KeyValueRepository that dispatches to onTrueRepo if the query

\* predicate returns true, dispatches to onFalseRepo otherwise.

\*/

def choose[Q, K, V](

predicate: Q => Boolean,

onTrueRepo: KeyValueRepository[Q, K, V],

onFalseRepo: KeyValueRepository[Q, K, V]

): KeyValueRepository[Q, K, V] = { (query: Q) =>

{

if (predicate(query)) {

onTrueRepo(query)

} else {

onFalseRepo(query)

}

}

}

/\*\*

\* Short-circuit a KeyValueRepository to return an empty

\* KeyValueResult when the query is empty rather than calling the

\* backend. It is up to the caller to define empty.

\*

\* The implementation of repo and isEmpty should satisfy:

\*

\* forAll { (q: Q) => !isEmpty(q) || (repo(q).get == KeyValueResult.empty[K, V]) }

\*/

def shortCircuitEmpty[Q, K, V](

isEmpty: Q => Boolean

)(

repo: KeyValueRepository[Q, K, V]

): KeyValueRepository[Q, K, V] = { q =>

if (isEmpty(q)) KeyValueResult.emptyFuture[K, V] else repo(q)

}

/\*\*

\* Short-circuit a KeyValueRepository to return an empty

\* KeyValueResult for any empty Traversable query rather than

\* calling the backend.

\*

\* The implementation of repo should satisfy:

\*

\* forAll { (q: Q) => !q.isEmpty || (repo(q).get == KeyValueResult.empty[K, V]) }

\*/

def shortCircuitEmpty[Q <: Traversable[\_], K, V](

repo: KeyValueRepository[Q, K, V]

): KeyValueRepository[Q, K, V] = shortCircuitEmpty[Q, K, V]((\_: Q).isEmpty)(repo)

/\*\*

\* Turns a bulking KeyValueRepository into a non-bulking Repository. The query to the

\* KeyValueRepository must be nothing more than a Seq[K].

\*/

def singular[K, V](repo: KeyValueRepository[Seq[K], K, V]): Repository[K, Option[V]] =

singular(repo, (key: K) => Seq(key))

/\*\*

\* Turns a bulking KeyValueRepository into a non-bulking Repository.

\*/

def singular[Q, K, V](

repo: KeyValueRepository[Q, K, V],

queryBuilder: K => Q

): Repository[K, Option[V]] =

key => {

repo(queryBuilder(key)) flatMap { results =>

Future.const(results(key))

}

}

/\*\*

\* Converts a KeyValueRepository with value type V to one with value type

\* V2 using a function that maps found values.

\*/

def mapFound[Q, K, V, V2](

repo: KeyValueRepository[Q, K, V],

f: V => V2

): KeyValueRepository[Q, K, V2] =

repo andThen { \_ map { \_ mapFound f } }

/\*\*

\* Converts a KeyValueRepository with value type V to one with value type

\* V2 using a function that maps over results.

\*/

def mapValues[Q, K, V, V2](

repo: KeyValueRepository[Q, K, V],

f: Try[Option[V]] => Try[Option[V2]]

): KeyValueRepository[Q, K, V2] =

repo andThen { \_ map { \_ mapValues f } }

/\*\*

\* Turns a KeyValueRepository which may throw an exception to another

\* KeyValueRepository which always returns Future.value(KeyValueResult)

\* even when there is an exception

\*/

def scatterExceptions[Q <: Traversable[K], K, V](

repo: KeyValueRepository[Q, K, V]

): KeyValueRepository[Q, K, V] =

q =>

repo(q) handle {

case t => KeyValueResult[K, V](failed = q map { \_ -> t } toMap)

}

}