package com.twitter.ann.common

import com.twitter.ann.common.EmbeddingType.EmbeddingVector

import com.twitter.util.Future

import scala.util.Random

trait ShardFunction[T] {

/\*\*

\* Shard function to shard embedding based on total shards and embedding data.

\* @param shards

\* @param entity

\* @return Shard index, from 0(Inclusive) to shards(Exclusive))

\*/

def apply(shards: Int, entity: EntityEmbedding[T]): Int

}

/\*\*

\* Randomly shards the embeddings based on number of total shards.

\*/

class RandomShardFunction[T] extends ShardFunction[T] {

def apply(shards: Int, entity: EntityEmbedding[T]): Int = {

Random.nextInt(shards)

}

}

/\*\*

\* Sharded appendable to shard the embedding into different appendable indices

\* @param indices: Sequence of appendable indices

\* @param shardFn: Shard function to shard data into different indices

\* @param shards: Total shards

\* @tparam T: Type of id.

\*/

class ShardedAppendable[T, P <: RuntimeParams, D <: Distance[D]](

indices: Seq[Appendable[T, P, D]],

shardFn: ShardFunction[T],

shards: Int)

extends Appendable[T, P, D] {

override def append(entity: EntityEmbedding[T]): Future[Unit] = {

val shard = shardFn(shards, entity)

val index = indices(shard)

index.append(entity)

}

override def toQueryable: Queryable[T, P, D] = {

new ComposedQueryable[T, P, D](indices.map(\_.toQueryable))

}

}

/\*\*

\* Composition of sequence of queryable indices, it queries all the indices,

\* and merges the result in memory to return the K nearest neighbours

\* @param indices: Sequence of queryable indices

\* @tparam T: Type of id

\* @tparam P: Type of runtime param

\* @tparam D: Type of distance metric

\*/

class ComposedQueryable[T, P <: RuntimeParams, D <: Distance[D]](

indices: Seq[Queryable[T, P, D]])

extends Queryable[T, P, D] {

private[this] val ordering =

Ordering.by[NeighborWithDistance[T, D], D](\_.distance)

override def query(

embedding: EmbeddingVector,

numOfNeighbors: Int,

runtimeParams: P

): Future[List[T]] = {

val neighbours = queryWithDistance(embedding, numOfNeighbors, runtimeParams)

neighbours.map(list => list.map(nn => nn.neighbor))

}

override def queryWithDistance(

embedding: EmbeddingVector,

numOfNeighbors: Int,

runtimeParams: P

): Future[List[NeighborWithDistance[T, D]]] = {

val futures = Future.collect(

indices.map(index => index.queryWithDistance(embedding, numOfNeighbors, runtimeParams))

)

futures.map { list =>

list.flatten

.sorted(ordering)

.take(numOfNeighbors)

.toList

}

}

}