package com.twitter.ann.hnsw

import com.google.common.annotations.VisibleForTesting

import com.twitter.ann.common.EmbeddingType.\_

import com.twitter.ann.common.Metric.toThrift

import com.twitter.ann.common.\_

import com.twitter.ann.common.thriftscala.DistanceMetric

import com.twitter.ann.hnsw.HnswIndex.RandomProvider

import com.twitter.util.Future

import java.util.Random

import java.util.concurrent.ConcurrentHashMap

import java.util.concurrent.ThreadLocalRandom

import java.util.concurrent.locks.Lock

import java.util.concurrent.locks.ReentrantLock

import scala.collection.JavaConverters.\_

private[hnsw] object Hnsw {

private[hnsw] def apply[T, D <: Distance[D]](

dimension: Int,

metric: Metric[D],

efConstruction: Int,

maxM: Int,

expectedElements: Int,

futurePool: ReadWriteFuturePool,

idEmbeddingMap: IdEmbeddingMap[T]

): Hnsw[T, D] = {

val randomProvider = new RandomProvider {

override def get(): Random = ThreadLocalRandom.current()

}

val distFn =

DistanceFunctionGenerator(metric, (key: T) => idEmbeddingMap.get(key))

val internalIndex = new HnswIndex[T, EmbeddingVector](

distFn.index,

distFn.query,

efConstruction,

maxM,

expectedElements,

randomProvider

)

new Hnsw[T, D](

dimension,

metric,

internalIndex,

futurePool,

idEmbeddingMap,

distFn.shouldNormalize,

LockedAccess.apply(expectedElements)

)

}

}

private[hnsw] object LockedAccess {

protected[hnsw] def apply[T](expectedElements: Int): LockedAccess[T] =

DefaultLockedAccess(new ConcurrentHashMap[T, Lock](expectedElements))

protected[hnsw] def apply[T](): LockedAccess[T] =

DefaultLockedAccess(new ConcurrentHashMap[T, Lock]())

}

private[hnsw] case class DefaultLockedAccess[T](locks: ConcurrentHashMap[T, Lock])

extends LockedAccess[T] {

override def lockProvider(item: T) = locks.computeIfAbsent(item, (\_: T) => new ReentrantLock())

}

private[hnsw] trait LockedAccess[T] {

protected def lockProvider(item: T): Lock

def lock[K](item: T)(fn: => K): K = {

val lock = lockProvider(item)

lock.lock()

try {

fn

} finally {

lock.unlock()

}

}

}

@VisibleForTesting

private[hnsw] class Hnsw[T, D <: Distance[D]](

dimension: Int,

metric: Metric[D],

hnswIndex: HnswIndex[T, EmbeddingVector],

readWriteFuturePool: ReadWriteFuturePool,

idEmbeddingMap: IdEmbeddingMap[T],

shouldNormalize: Boolean,

lockedAccess: LockedAccess[T] = LockedAccess.apply[T]())

extends Appendable[T, HnswParams, D]

with Queryable[T, HnswParams, D]

with Updatable[T] {

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

readWriteFuturePool.write {

val indexDimension = entity.embedding.length

assert(

toThrift(metric) == DistanceMetric.EditDistance || indexDimension == dimension,

s"Dimension mismatch for index(${indexDimension}) and embedding($dimension)"

)

lockedAccess.lock(entity.id) {

// To make this thread-safe, we are using ConcurrentHashMap#putIfAbsent underneath,

// so if there is a pre-existing item, put() will return something that is not null

val embedding = idEmbeddingMap.putIfAbsent(entity.id, updatedEmbedding(entity.embedding))

if (embedding == null) { // New element - insert into the index

hnswIndex.insert(entity.id)

} else { // Existing element - update the embedding and graph structure

throw new IllegalDuplicateInsertException(

"Append method does not permit duplicates (try using update method): " + entity.id)

}

}

} onFailure { e =>

Future.exception(e)

}

}

override def toQueryable: Queryable[T, HnswParams, D] = this

override def query(

embedding: EmbeddingVector,

numOfNeighbours: Int,

runtimeParams: HnswParams

): Future[List[T]] = {

queryWithDistance(embedding, numOfNeighbours, runtimeParams)

.map(\_.map(\_.neighbor))

}

override def queryWithDistance(

embedding: EmbeddingVector,

numOfNeighbours: Int,

runtimeParams: HnswParams

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

val indexDimension = embedding.length

assert(

toThrift(metric) == DistanceMetric.EditDistance || indexDimension == dimension,

s"Dimension mismatch for index(${indexDimension}) and embedding($dimension)"

)

readWriteFuturePool.read {

hnswIndex

.searchKnn(updatedEmbedding(embedding), numOfNeighbours, runtimeParams.ef)

.asScala

.map { nn =>

NeighborWithDistance(

nn.getItem,

metric.fromAbsoluteDistance(nn.getDistance)

)

}

.toList

}

}

private[this] def updatedEmbedding(embedding: EmbeddingVector): EmbeddingVector = {

if (shouldNormalize) {

MetricUtil.norm(embedding)

} else {

embedding

}

}

def getIndex: HnswIndex[T, EmbeddingVector] = hnswIndex

def getDimen: Int = dimension

def getMetric: Metric[D] = metric

def getIdEmbeddingMap: IdEmbeddingMap[T] = idEmbeddingMap

override def update(

entity: EntityEmbedding[T]

): Future[Unit] = {

readWriteFuturePool.write {

val indexDimension = entity.embedding.length

assert(

toThrift(metric) == DistanceMetric.EditDistance || indexDimension == dimension,

s"Dimension mismatch for index(${indexDimension}) and embedding($dimension)"

)

lockedAccess.lock(entity.id) {

val embedding = idEmbeddingMap.put(entity.id, updatedEmbedding(entity.embedding))

if (embedding == null) { // New element - insert into the index

hnswIndex.insert(entity.id)

} else { // Existing element - update the embedding and graph structure

hnswIndex.reInsert(entity.id);

}

}

} onFailure { e =>

Future.exception(e)

}

}

}