package com.twitter.ann.hnsw

import com.google.common.annotations.VisibleForTesting

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

import com.twitter.ann.common.\_

import com.twitter.ann.common.thriftscala.HnswIndexMetadata

import com.twitter.ann.hnsw.HnswCommon.\_

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

import com.twitter.bijection.Injection

import com.twitter.search.common.file.AbstractFile

import com.twitter.search.common.file.FileUtils

import com.twitter.util.Future

import java.io.IOException

import java.util.concurrent.ThreadLocalRandom

import java.util.Random

import org.apache.beam.sdk.io.fs.ResourceId

private[hnsw] object SerializableHnsw {

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

index: Hnsw[T, D],

injection: Injection[T, Array[Byte]]

): SerializableHnsw[T, D] = {

new SerializableHnsw[T, D](

index,

injection

)

}

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

dimension: Int,

metric: Metric[D],

injection: Injection[T, Array[Byte]],

futurePool: ReadWriteFuturePool,

directory: AbstractFile

): SerializableHnsw[T, D] = {

val metadata = HnswIOUtil.loadIndexMetadata(directory.getChild(MetaDataFileName))

validateMetadata(dimension, metric, metadata)

val idEmbeddingMap = JMapBasedIdEmbeddingMap.loadInMemory(

directory.getChild(EmbeddingMappingFileName),

injection,

Some(metadata.numElements)

)

loadIndex(

dimension,

metric,

injection,

futurePool,

directory,

idEmbeddingMap,

metadata

)

}

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

dimension: Int,

metric: Metric[D],

injection: Injection[T, Array[Byte]],

futurePool: ReadWriteFuturePool,

directory: AbstractFile

): SerializableHnsw[T, D] = {

val metadata = HnswIOUtil.loadIndexMetadata(directory.getChild(MetaDataFileName))

validateMetadata(dimension, metric, metadata)

loadIndex(

dimension,

metric,

injection,

futurePool,

directory,

MapDbBasedIdEmbeddingMap

.loadAsReadonly(directory.getChild(EmbeddingMappingFileName), injection),

metadata

)

}

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

dimension: Int,

metric: Metric[D],

injection: Injection[T, Array[Byte]],

futurePool: ReadWriteFuturePool,

directory: AbstractFile,

idEmbeddingMap: IdEmbeddingMap[T],

metadata: HnswIndexMetadata

): SerializableHnsw[T, D] = {

val distFn =

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

val randomProvider = new RandomProvider {

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

}

val internalIndex = HnswIndex.loadHnswIndex[T, EmbeddingVector](

distFn.index,

distFn.query,

directory.getChild(InternalIndexDir),

injection,

randomProvider

)

val index = new Hnsw[T, D](

dimension,

metric,

internalIndex,

futurePool,

idEmbeddingMap,

distFn.shouldNormalize,

LockedAccess.apply(metadata.numElements)

)

new SerializableHnsw(index, injection)

}

private[this] def validateMetadata[D <: Distance[D]](

dimension: Int,

metric: Metric[D],

existingMetadata: HnswIndexMetadata

): Unit = {

assert(

existingMetadata.dimension == dimension,

s"Dimensions do not match. requested: $dimension existing: ${existingMetadata.dimension}"

)

val existingMetric = Metric.fromThrift(existingMetadata.distanceMetric)

assert(

existingMetric == metric,

s"DistanceMetric do not match. requested: $metric existing: $existingMetric"

)

}

}

@VisibleForTesting

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

index: Hnsw[T, D],

injection: Injection[T, Array[Byte]])

extends Appendable[T, HnswParams, D]

with Queryable[T, HnswParams, D]

with Serialization

with Updatable[T] {

override def append(entity: EntityEmbedding[T]) = index.append(entity)

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

override def query(

embedding: EmbeddingVector,

numOfNeighbours: Int,

runtimeParams: HnswParams

) = index.query(embedding, numOfNeighbours, runtimeParams)

override def queryWithDistance(

embedding: EmbeddingVector,

numOfNeighbours: Int,

runtimeParams: HnswParams

) = index.queryWithDistance(embedding, numOfNeighbours, runtimeParams)

def toDirectory(directory: ResourceId): Unit = {

toDirectory(new IndexOutputFile(directory))

}

def toDirectory(directory: AbstractFile): Unit = {

// Create a temp dir with time prefix, and then do a rename after serialization

val tmpDir = FileUtils.getTmpFileHandle(directory)

if (!tmpDir.exists()) {

tmpDir.mkdirs()

}

toDirectory(new IndexOutputFile(tmpDir))

// Rename tmp dir to original directory supplied

if (!tmpDir.rename(directory)) {

throw new IOException(s"Failed to rename ${tmpDir.getPath} to ${directory.getPath}")

}

}

private def toDirectory(indexFile: IndexOutputFile): Unit = {

// Save java based hnsw index

index.getIndex.toDirectory(indexFile.createDirectory(InternalIndexDir), injection)

// Save index metadata

HnswIOUtil.saveIndexMetadata(

index.getDimen,

index.getMetric,

index.getIdEmbeddingMap.size(),

indexFile.createFile(MetaDataFileName).getOutputStream()

)

// Save embedding mapping

index.getIdEmbeddingMap.toDirectory(

indexFile.createFile(EmbeddingMappingFileName).getOutputStream())

// Create \_SUCCESS file

indexFile.createSuccessFile()

}

override def update(

entity: EntityEmbedding[T]

): Future[Unit] = {

index.update(entity)

}

}