package com.twitter.ann.service.query\_server.common

import com.google.common.annotations.VisibleForTesting

import com.google.common.util.concurrent.ThreadFactoryBuilder

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

import com.twitter.ann.common.Distance

import com.twitter.ann.common.NeighborWithDistance

import com.twitter.ann.common.Queryable

import com.twitter.ann.common.QueryableGrouped

import com.twitter.ann.common.RuntimeParams

import com.twitter.conversions.DurationOps.\_

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.logging.Logger

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

import com.twitter.util.Duration

import com.twitter.util.Future

import java.util.concurrent.atomic.AtomicReference

import java.util.concurrent.Executors

import java.util.concurrent.TimeUnit

import scala.util.Random

import scala.util.control.NonFatal

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

grouped: Boolean,

rootDir: AbstractFile,

queryableProvider: QueryableProvider[T, P, D],

indexPathProvider: IndexPathProvider,

statsReceiver: StatsReceiver,

updateInterval: Duration = 10.minutes)

extends QueryableGrouped[T, P, D] {

private val log = Logger.get("RefreshableQueryable")

private val loadCounter = statsReceiver.counter("load")

private val loadFailCounter = statsReceiver.counter("load\_error")

private val newIndexCounter = statsReceiver.counter("new\_index")

protected val random = new Random(System.currentTimeMillis())

private val threadFactory = new ThreadFactoryBuilder()

.setNameFormat("refreshable-queryable-update-%d")

.build()

// single thread to check and load index

private val executor = Executors.newScheduledThreadPool(1, threadFactory)

private[common] val indexPathRef: AtomicReference[AbstractFile] =

new AtomicReference(indexPathProvider.provideIndexPath(rootDir, grouped).get())

private[common] val queryableRef: AtomicReference[Map[Option[String], Queryable[T, P, D]]] = {

if (grouped) {

val mapping = getGroupMapping

new AtomicReference(mapping)

} else {

new AtomicReference(Map(None -> buildIndex(indexPathRef.get())))

}

}

private val servingIndexGauge = statsReceiver.addGauge("serving\_index\_timestamp") {

indexPathRef.get().getName.toFloat

}

log.info("System.gc() before start")

System.gc()

private val reloadTask = new Runnable {

override def run(): Unit = {

innerLoad()

}

}

def start(): Unit = {

executor.scheduleWithFixedDelay(

reloadTask,

// init reloading with random delay

computeRandomInitDelay().inSeconds,

updateInterval.inSeconds,

TimeUnit.SECONDS

)

}

private def buildIndex(indexPath: AbstractFile): Queryable[T, P, D] = {

log.info(s"build index from ${indexPath.getPath}")

queryableProvider.provideQueryable(indexPath)

}

@VisibleForTesting

private[common] def innerLoad(): Unit = {

log.info("Check and load for new index")

loadCounter.incr()

try {

// Find the latest directory

val latestPath = indexPathProvider.provideIndexPath(rootDir, grouped).get()

if (indexPathRef.get() != latestPath) {

log.info(s"loading index from: ${latestPath.getName}")

newIndexCounter.incr()

if (grouped) {

val mapping = getGroupMapping

queryableRef.set(mapping)

} else {

val queryable = buildIndex(latestPath)

queryableRef.set(Map(None -> queryable))

}

indexPathRef.set(latestPath)

} else {

log.info(s"Current index already up to date: ${indexPathRef.get.getName}")

}

} catch {

case NonFatal(err) =>

loadFailCounter.incr()

log.error(s"Failed to load index: $err")

}

log.info(s"Current index loaded from ${indexPathRef.get().getPath}")

}

@VisibleForTesting

private[common] def computeRandomInitDelay(): Duration = {

val bound = 5.minutes

val nextUpdateSec = updateInterval + Duration.fromSeconds(

random.nextInt(bound.inSeconds)

)

nextUpdateSec

}

/\*\*

\* ANN query for ids with key as group id

\* @param embedding: Embedding/Vector to be queried with.

\* @param numOfNeighbors: Number of neighbours to be queried for.

\* @param runtimeParams: Runtime params associated with index to control accuracy/latency etc.

\* @param key: Optional key to lookup specific ANN index and perform query there

\* @return List of approximate nearest neighbour ids.

\*/

override def query(

embedding: EmbeddingVector,

numOfNeighbors: Int,

runtimeParams: P,

key: Option[String]

): Future[List[T]] = {

val mapping = queryableRef.get()

if (!mapping.contains(key)) {

Future.value(List())

} else {

mapping.get(key).get.query(embedding, numOfNeighbors, runtimeParams)

}

}

/\*\*

\* ANN query for ids with key as group id with distance

\* @param embedding: Embedding/Vector to be queried with.

\* @param numOfNeighbors: Number of neighbours to be queried for.

\* @param runtimeParams: Runtime params associated with index to control accuracy/latency etc.

\* @param key: Optional key to lookup specific ANN index and perform query there

\* @return List of approximate nearest neighbour ids with distance from the query embedding.

\*/

override def queryWithDistance(

embedding: EmbeddingVector,

numOfNeighbors: Int,

runtimeParams: P,

key: Option[String]

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

val mapping = queryableRef.get()

if (!mapping.contains(key)) {

Future.value(List())

} else {

mapping.get(key).get.queryWithDistance(embedding, numOfNeighbors, runtimeParams)

}

}

private def getGroupMapping(): Map[Option[String], Queryable[T, P, D]] = {

val groupDirs = indexPathProvider.provideIndexPathWithGroups(rootDir).get()

val mapping = groupDirs.map { groupDir =>

val queryable = buildIndex(groupDir)

Option(groupDir.getName) -> queryable

}.toMap

mapping

}

/\*\*

\* ANN query for ids.

\*

\* @param embedding : Embedding/Vector to be queried with.

\* @param numOfNeighbors : Number of neighbours to be queried for.

\* @param runtimeParams : Runtime params associated with index to control accuracy/latency etc.

\*

\* @return List of approximate nearest neighbour ids.

\*/

override def query(

embedding: EmbeddingVector,

numOfNeighbors: Int,

runtimeParams: P

): Future[List[T]] = {

query(embedding, numOfNeighbors, runtimeParams, None)

}

/\*\*

\* ANN query for ids with distance.

\*

\* @param embedding : Embedding/Vector to be queried with.

\* @param numOfNeighbors : Number of neighbours to be queried for.

\* @param runtimeParams : Runtime params associated with index to control accuracy/latency etc.

\*

\* @return List of approximate nearest neighbour ids with distance from the query embedding.

\*/

override def queryWithDistance(

embedding: EmbeddingVector,

numOfNeighbors: Int,

runtimeParams: P

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

queryWithDistance(embedding, numOfNeighbors, runtimeParams, None)

}

}