package com.twitter.ann.service.loadtest

import com.twitter.ann.annoy.AnnoyCommon

import com.twitter.ann.annoy.AnnoyRuntimeParams

import com.twitter.ann.annoy.TypedAnnoyIndex

import com.twitter.ann.common.\_

import com.twitter.ann.common.thriftscala.{Distance => ServiceDistance}

import com.twitter.ann.common.thriftscala.{RuntimeParams => ServiceRuntimeParams}

import com.twitter.ann.faiss.FaissCommon

import com.twitter.ann.faiss.FaissParams

import com.twitter.ann.hnsw.HnswCommon

import com.twitter.ann.hnsw.HnswParams

import com.twitter.ann.hnsw.TypedHnswIndex

import com.twitter.bijection.Injection

import com.twitter.cortex.ml.embeddings.common.EntityKind

import com.twitter.finagle.mtls.authentication.ServiceIdentifier

import com.twitter.finagle.util.DefaultTimer

import com.twitter.finatra.mtls.modules.ServiceIdentifierModule

import com.twitter.inject.server.TwitterServer

import com.twitter.util.\_

import java.util.concurrent.TimeUnit

/\*\*

\* To build and upload:

\* $ ./bazel bundle ann/src/main/scala/com/twitter/ann/service/loadtest:bin --bundle-jvm-archive=zip

\* $ packer add\_version --cluster=smf1 $USER ann-loadtest dist/ann-loadtest.zip

\*/

object AnnLoadTestMain extends TwitterServer {

private[this] val algo =

flag[String]("algo", "load test server types: [annoy/hnsw]")

private[this] val targetQPS =

flag[Int]("qps", "target QPS for load test")

private[this] val queryIdType =

flag[String](

"query\_id\_type",

"query id type for load test: [long/string/int/user/tweet/word/url/tfwId]")

private[this] val indexIdType =

flag[String](

"index\_id\_type",

"index id type for load test: [long/string/int/user/tweet/word/url/tfwId]")

private[this] val metric =

flag[String]("metric", "metric type for load test: [Cosine/L2/InnerProduct]")

private[this] val durationSec =

flag[Int]("duration\_sec", "duration for the load test in sec")

private[this] val numberOfNeighbors =

flag[Seq[Int]]("number\_of\_neighbors", Seq(), "number of neighbors")

private[this] val dimension = flag[Int]("embedding\_dimension", "dimension of embeddings")

private[this] val querySetDir =

flag[String]("query\_set\_dir", "", "Directory containing the queries")

private[this] val indexSetDir =

flag[String](

"index\_set\_dir",

"",

"Directory containing the embeddings to be indexed"

)

private[this] val truthSetDir =

flag[String]("truth\_set\_dir", "", "Directory containing the truth data")

private[this] val loadTestType =

flag[String]("loadtest\_type", "Load test type [server/local]")

private[this] val serviceDestination =

flag[String]("service\_destination", "wily address of remote query service")

private[this] val concurrencyLevel =

flag[Int]("concurrency\_level", 8, "number of concurrent operations on the index")

// Queries with random embeddings

private[this] val withRandomQueries =

flag[Boolean]("with\_random\_queries", false, "query with random embeddings")

private[this] val randomQueriesCount =

flag[Int]("random\_queries\_count", 50000, "total random queries")

private[this] val randomEmbeddingMinValue =

flag[Float]("random\_embedding\_min\_value", -1.0f, "Min value of random embeddings")

private[this] val randomEmbeddingMaxValue =

flag[Float]("random\_embedding\_max\_value", 1.0f, "Max value of random embeddings")

// parameters for annoy

private[this] val numOfNodesToExplore =

flag[Seq[Int]]("annoy\_num\_of\_nodes\_to\_explore", Seq(), "number of nodes to explore")

private[this] val numOfTrees =

flag[Int]("annoy\_num\_trees", 0, "number of trees to build")

// parameters for HNSW

private[this] val efConstruction = flag[Int]("hnsw\_ef\_construction", "ef for Hnsw construction")

private[this] val ef = flag[Seq[Int]]("hnsw\_ef", Seq(), "ef for Hnsw query")

private[this] val maxM = flag[Int]("hnsw\_max\_m", "maxM for Hnsw")

// FAISS

private[this] val nprobe = flag[Seq[Int]]("faiss\_nprobe", Seq(), "nprobe for faiss query")

private[this] val quantizerEf =

flag[Seq[Int]]("faiss\_quantizerEf", Seq(0), "quantizerEf for faiss query")

private[this] val quantizerKfactorRF =

flag[Seq[Int]]("faiss\_quantizerKfactorRF", Seq(0), "quantizerEf for faiss query")

private[this] val quantizerNprobe =

flag[Seq[Int]]("faiss\_quantizerNprobe", Seq(0), "quantizerNprobe for faiss query")

private[this] val ht =

flag[Seq[Int]]("faiss\_ht", Seq(0), "ht for faiss query")

implicit val timer: Timer = DefaultTimer

override def start(): Unit = {

logger.info("Starting load test..")

logger.info(flag.getAll().mkString("\t"))

assert(numberOfNeighbors().nonEmpty, "number\_of\_neighbors not defined")

assert(dimension() > 0, s"Invalid dimension ${dimension()}")

val inMemoryBuildRecorder = new InMemoryLoadTestBuildRecorder

val queryableFuture = buildQueryable(inMemoryBuildRecorder)

val queryConfig = getQueryRuntimeConfig

val result = queryableFuture.flatMap { queryable =>

performQueries(queryable, queryConfig, getQueries)

}

Await.result(result)

System.out.println(s"Target QPS: ${targetQPS()}")

System.out.println(s"Duration per test: ${durationSec()}")

System.out.println(s"Concurrency Level: ${concurrencyLevel()}")

LoadTestUtils

.printResults(inMemoryBuildRecorder, queryConfig)

.foreach(System.out.println)

Await.result(close())

System.exit(0)

}

private[this] def getQueries[Q, I]: Seq[Query[I]] = {

if (withRandomQueries()) {

assert(

truthSetDir().isEmpty,

"Cannot use truth set when query with random embeddings enabled"

)

val queries = LoadTestUtils.getRandomQuerySet(

dimension(),

randomQueriesCount(),

randomEmbeddingMinValue(),

randomEmbeddingMaxValue()

)

queries.map(Query[I](\_))

} else {

assert(querySetDir().nonEmpty, "Query set path is empty")

assert(queryIdType().nonEmpty, "Query id type is empty")

val queries = LoadTestUtils.getEmbeddingsSet[Q](querySetDir(), queryIdType())

if (truthSetDir().nonEmpty) {

// Join the queries with truth set data.

assert(indexIdType().nonEmpty, "Index id type is empty")

val truthSetMap =

LoadTestUtils.getTruthSetMap[Q, I](truthSetDir(), queryIdType(), indexIdType())

queries.map(entity => Query[I](entity.embedding, truthSetMap(entity.id)))

} else {

queries.map(entity => Query[I](entity.embedding))

}

}

}

private[this] def getQueryRuntimeConfig[

T,

P <: RuntimeParams

]: Seq[QueryTimeConfiguration[T, P]] = {

val queryTimeConfig = algo() match {

case "annoy" =>

assert(numOfNodesToExplore().nonEmpty, "Must specify the num\_of\_nodes\_to\_explore")

logger.info(s"Querying annoy index with num\_of\_nodes\_to\_explore ${numOfNodesToExplore()}")

for {

numNodes <- numOfNodesToExplore()

numOfNeighbors <- numberOfNeighbors()

} yield {

buildQueryTimeConfig[T, AnnoyRuntimeParams](

numOfNeighbors,

AnnoyRuntimeParams(Some(numNodes)),

Map(

"numNodes" -> numNodes.toString,

"numberOfNeighbors" -> numOfNeighbors.toString

)

).asInstanceOf[QueryTimeConfiguration[T, P]]

}

case "hnsw" =>

assert(ef().nonEmpty, "Must specify ef")

logger.info(s"Querying hnsw index with ef ${ef()}")

for {

ef <- ef()

numOfNeighbors <- numberOfNeighbors()

} yield {

buildQueryTimeConfig[T, HnswParams](

numOfNeighbors,

HnswParams(ef),

Map(

"efConstruction" -> ef.toString,

"numberOfNeighbors" -> numOfNeighbors.toString

)

).asInstanceOf[QueryTimeConfiguration[T, P]]

}

case "faiss" =>

assert(nprobe().nonEmpty, "Must specify nprobe")

def toNonZeroOptional(x: Int): Option[Int] = if (x != 0) Some(x) else None

for {

numOfNeighbors <- numberOfNeighbors()

runNProbe <- nprobe()

runQEF <- quantizerEf()

runKFactorEF <- quantizerKfactorRF()

runQNProbe <- quantizerNprobe()

runHT <- ht()

} yield {

val params = FaissParams(

Some(runNProbe),

toNonZeroOptional(runQEF),

toNonZeroOptional(runKFactorEF),

toNonZeroOptional(runQNProbe),

toNonZeroOptional(runHT))

buildQueryTimeConfig[T, FaissParams](

numOfNeighbors,

params,

Map(

"nprobe" -> params.nprobe.toString,

"quantizer\_efSearch" -> params.quantizerEf.toString,

"quantizer\_k\_factor\_rf" -> params.quantizerKFactorRF.toString,

"quantizer\_nprobe" -> params.quantizerNprobe.toString,

"ht" -> params.ht.toString,

"numberOfNeighbors" -> numOfNeighbors.toString,

)

).asInstanceOf[QueryTimeConfiguration[T, P]]

}

case \_ => throw new IllegalArgumentException(s"server type: $algo is not supported yet")

}

queryTimeConfig

}

private def buildQueryable[T, P <: RuntimeParams, D <: Distance[D]](

inMemoryBuildRecorder: InMemoryLoadTestBuildRecorder

): Future[Queryable[T, P, D]] = {

val queryable = loadTestType() match {

case "remote" => {

assert(serviceDestination().nonEmpty, "Service destination not defined")

logger.info(s"Running load test with remote service ${serviceDestination()}")

LoadTestUtils.buildRemoteServiceQueryClient[T, P, D](

serviceDestination(),

"ann-load-test",

statsReceiver,

injector.instance[ServiceIdentifier],

getRuntimeParamInjection[P],

getDistanceInjection[D],

getIndexIdInjection[T]

)

}

case "local" => {

logger.info("Running load test locally..")

assert(indexSetDir().nonEmpty, "Index set path is empty")

val statsLoadTestBuildRecorder = new StatsLoadTestBuildRecorder(statsReceiver)

val buildRecorder =

new ComposedLoadTestBuildRecorder(Seq(inMemoryBuildRecorder, statsLoadTestBuildRecorder))

indexEmbeddingsAndGetQueryable[T, P, D](

buildRecorder,

LoadTestUtils.getEmbeddingsSet(indexSetDir(), indexIdType())

)

}

}

queryable

}

private def indexEmbeddingsAndGetQueryable[T, P <: RuntimeParams, D <: Distance[D]](

buildRecorder: LoadTestBuildRecorder,

indexSet: Seq[EntityEmbedding[T]]

): Future[Queryable[T, P, D]] = {

logger.info(s"Indexing entity embeddings in index set with size ${indexSet.size}")

val metric = getDistanceMetric[D]

val indexIdInjection = getIndexIdInjection[T]

val indexBuilder = new AnnIndexBuildLoadTest(buildRecorder)

val appendable = algo() match {

case "annoy" =>

assert(numOfTrees() > 0, "Must specify the number of trees for annoy")

logger.info(

s"Creating annoy index locally with num\_of\_trees: ${numOfTrees()}"

)

TypedAnnoyIndex

.indexBuilder(

dimension(),

numOfTrees(),

metric,

indexIdInjection,

FuturePool.interruptibleUnboundedPool

)

case "hnsw" =>

assert(efConstruction() > 0 && maxM() > 0, "Must specify ef\_construction and max\_m")

logger.info(

s"Creating hnsw index locally with max\_m: ${maxM()} and ef\_construction: ${efConstruction()}"

)

TypedHnswIndex

.index[T, D](

dimension(),

metric,

efConstruction(),

maxM(),

indexSet.size,

ReadWriteFuturePool(FuturePool.interruptibleUnboundedPool)

)

}

indexBuilder

.indexEmbeddings(appendable, indexSet, concurrencyLevel())

.asInstanceOf[Future[Queryable[T, P, D]]]

}

private[this] def performQueries[T, P <: RuntimeParams, D <: Distance[D]](

queryable: Queryable[T, P, D],

queryTimeConfig: Seq[QueryTimeConfiguration[T, P]],

queries: Seq[Query[T]]

): Future[Unit] = {

val indexQuery = new AnnIndexQueryLoadTest()

val duration = Duration(durationSec().toLong, TimeUnit.SECONDS)

indexQuery.performQueries(

queryable,

targetQPS(),

duration,

queries,

concurrencyLevel(),

queryTimeConfig

)

}

// provide index id injection based on argument

private[this] def getIndexIdInjection[T]: Injection[T, Array[Byte]] = {

val injection = indexIdType() match {

case "long" => AnnInjections.LongInjection

case "string" => AnnInjections.StringInjection

case "int" => AnnInjections.IntInjection

case entityKind => EntityKind.getEntityKind(entityKind).byteInjection

}

injection.asInstanceOf[Injection[T, Array[Byte]]]

}

private[this] def getRuntimeParamInjection[

P <: RuntimeParams

]: Injection[P, ServiceRuntimeParams] = {

val injection = algo() match {

case "annoy" => AnnoyCommon.RuntimeParamsInjection

case "hnsw" => HnswCommon.RuntimeParamsInjection

case "faiss" => FaissCommon.RuntimeParamsInjection

}

injection.asInstanceOf[Injection[P, ServiceRuntimeParams]]

}

// provide distance injection based on argument

private[this] def getDistanceInjection[D <: Distance[D]]: Injection[D, ServiceDistance] = {

Metric.fromString(metric()).asInstanceOf[Injection[D, ServiceDistance]]

}

private[this] def getDistanceMetric[D <: Distance[D]]: Metric[D] = {

Metric.fromString(metric()).asInstanceOf[Metric[D]]

}

private[this] def buildQueryTimeConfig[T, P <: RuntimeParams](

numOfNeighbors: Int,

params: P,

config: Map[String, String]

): QueryTimeConfiguration[T, P] = {

val printableQueryRecorder = new InMemoryLoadTestQueryRecorder[T]()

val scope = config.flatMap { case (key, value) => Seq(key, value.toString) }.toSeq

val statsLoadTestQueryRecorder = new StatsLoadTestQueryRecorder[T](

// Put the run time params in the stats receiver names so that we can tell the difference when

// we look at them later.

statsReceiver.scope(algo()).scope(scope: \_\*)

)

val queryRecorder = new ComposedLoadTestQueryRecorder(

Seq(printableQueryRecorder, statsLoadTestQueryRecorder)

)

QueryTimeConfiguration(

queryRecorder,

params,

numOfNeighbors,

printableQueryRecorder

)

}

override protected def modules: Seq[com.google.inject.Module] = Seq(ServiceIdentifierModule)

}