package com.twitter.ann.service.loadtest

import com.google.common.annotations.VisibleForTesting

import com.twitter.ann.common.Distance

import com.twitter.ann.common.Queryable

import com.twitter.ann.common.RuntimeParams

import com.twitter.concurrent.AsyncMeter

import com.twitter.concurrent.AsyncStream

import com.twitter.finagle.util.DefaultTimer

import com.twitter.logging.Logger

import com.twitter.util.Duration

import com.twitter.util.Future

import com.twitter.util.Stopwatch

import com.twitter.util.Timer

import com.twitter.util.Try

import java.util.concurrent.atomic.AtomicInteger

object QueryTimeConfiguration {

val ResultHeader =

"params\tnumNeighbors\trecall@1\trecall@10\trecall\tavgLatencyMicros\tp50LatencyMicros\tp90LatencyMicros\tp99LatencyMicros\tavgRPS"

}

case class QueryTimeConfiguration[T, P <: RuntimeParams](

recorder: LoadTestQueryRecorder[T],

param: P,

numberOfNeighbors: Int,

private val results: InMemoryLoadTestQueryRecorder[T]) {

override def toString: String =

s"QueryTimeConfiguration(param = $param, numberOfNeighbors = $numberOfNeighbors)"

def printResults: String = {

val snapshot = results.computeSnapshot()

s"$param\t$numberOfNeighbors\t${results.top1Recall}\t${results.top10Recall}\t${results.recall}\t${snapshot.avgQueryLatencyMicros}\t${snapshot.p50QueryLatencyMicros}\t${snapshot.p90QueryLatencyMicros}\t${snapshot.p99QueryLatencyMicros}\t${results.avgRPS}"

}

}

/\*\*

\* Basic worker for ANN benchmark, send query with configured QPS and record results

\*/

class AnnLoadTestWorker(

timer: Timer = DefaultTimer) {

private val logger = Logger()

/\*\*

\* @param queries List of tuple of query embedding and corresponding list of truth set of ids associated with the embedding

\* @param qps the maximum number of request per second to send to the queryable. Note that if you

\* do not set the concurrency level high enough you may not be able to achieve this.

\* @param duration how long to perform the load test.

\* @param configuration Query configuration encapsulating runtime params and recorder.

\* @param concurrencyLevel The maximum number of concurrent requests to the queryable at a time.

\* Note that you may not be able to achieve this number of concurrent

\* requests if you do not have the QPS set high enough.

\*

\* @return a Future that completes when the load test is over. It contains the number of requests

\* sent.

\*/

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

queryable: Queryable[T, P, D],

queries: Seq[Query[T]],

qps: Int,

duration: Duration,

configuration: QueryTimeConfiguration[T, P],

concurrencyLevel: Int

): Future[Int] = {

val elapsed = Stopwatch.start()

val atomicInteger = new AtomicInteger(0)

val fullStream = Stream.continually {

if (elapsed() <= duration) {

logger.ifDebug(s"running with config: $configuration")

Some(atomicInteger.getAndIncrement() % queries.size)

} else {

logger.ifDebug(s"stopping with config: $configuration")

None

}

}

val limitedStream = fullStream.takeWhile(\_.isDefined).flatten

// at most we will have concurrencyLevel concurrent requests. So we should never have more than

// concurrency level waiters.

val asyncMeter = AsyncMeter.perSecond(qps, concurrencyLevel)(timer)

Future.Unit.before {

AsyncStream

.fromSeq(limitedStream).mapConcurrent(concurrencyLevel) { index =>

asyncMeter.await(1).flatMap { \_ =>

performQuery(configuration, queryable, queries(index))

}

}.size

}

}

@VisibleForTesting

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

configuration: QueryTimeConfiguration[T, P],

queryable: Queryable[T, P, D],

query: Query[T]

): Future[Try[Unit]] = {

val elapsed = Stopwatch.start()

queryable

.query(query.embedding, configuration.numberOfNeighbors, configuration.param)

.onSuccess { res: List[T] =>

// underneath LoadTestRecorder will record results for load test

// knnMap should be truncated to be same size as query result

configuration.recorder.recordQueryResult(

query.trueNeighbours,

res,

elapsed.apply()

)

logger.ifDebug(s"Successful query for $query")

}

.onFailure { e =>

logger.error(s"Failed query for $query: " + e)

}

.unit

.liftToTry

}

}