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

import com.twitter.ann.common.RuntimeParams

import com.twitter.ann.common.Task

import com.twitter.ann.faiss.FaissParams

import com.twitter.ann.hnsw.HnswParams

import com.twitter.ann.service.query\_server.common.throttling.ThrottlingBasedQualityTask.SAMPLING\_INTERVAL

import com.twitter.conversions.DurationOps.richDurationFromInt

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.util.Duration

import com.twitter.util.Future

import com.twitter.util.logging.Logging

object ThrottlingBasedQualityTask {

private[throttling] val SAMPLING\_INTERVAL = 100.milliseconds

}

class ThrottlingBasedQualityTask(

override val statsReceiver: StatsReceiver,

// Parameters are taken from OverloadAdmissionController

instrument: ThrottlingInstrument = new WindowedThrottlingInstrument(SAMPLING\_INTERVAL, 5,

new AuroraCPUStatsReader()))

extends Task

with Logging {

import ThrottlingBasedQualityTask.\_

// [0, 1] where 1 is fully throttled

// Quickly throttle, but dampen recovery to make sure we won't enter throttle/GC death spiral

@volatile private var dampenedThrottlingPercentage: Double = 0

protected[throttling] def task(): Future[Unit] = {

if (!instrument.disabled) {

instrument.sample()

val delta = instrument.percentageOfTimeSpentThrottling - dampenedThrottlingPercentage

if (delta > 0) {

// We want to start shedding load, do it quickly

dampenedThrottlingPercentage += delta

} else {

// Recover much slower

// At the rate of 100ms per sample, lookback is 2 minutes

val samplesToConverge = 1200.toDouble

dampenedThrottlingPercentage =

dampenedThrottlingPercentage + delta \* (2 / (samplesToConverge + 1))

}

statsReceiver.stat("dampened\_throttling").add(dampenedThrottlingPercentage.toFloat \* 100)

}

Future.Unit

}

protected def taskInterval: Duration = SAMPLING\_INTERVAL

def discountParams[T <: RuntimeParams](params: T): T = {

// [0, 1] where 1 is best quality and lowest speed

// It's expected to run @1 majority of time

val qualityFactor = math.min(1, math.max(0, 1 - dampenedThrottlingPercentage))

def applyQualityFactor(param: Int) = math.max(1, math.ceil(param \* qualityFactor).toInt)

params match {

case HnswParams(ef) => HnswParams(applyQualityFactor(ef)).asInstanceOf[T]

case FaissParams(nprobe, quantizerEf, quantizerKFactorRF, quantizerNprobe, ht) =>

FaissParams(

nprobe.map(applyQualityFactor),

quantizerEf.map(applyQualityFactor),

quantizerKFactorRF.map(applyQualityFactor),

quantizerNprobe.map(applyQualityFactor),

ht.map(applyQualityFactor)

).asInstanceOf[T]

}

}

}