package com.twitter.ann.service.loadtest

import com.google.common.util.concurrent.AtomicDouble

import com.twitter.finagle.stats.{MetricsBucketedHistogram, Snapshot, StatsReceiver}

import com.twitter.util.{Duration, Stopwatch}

import java.util.concurrent.atomic.{AtomicInteger, AtomicReference}

trait LoadTestQueryRecorder[T] {

def recordQueryResult(

trueNeighbors: Seq[T],

foundNeighbors: Seq[T],

queryLatency: Duration

): Unit

}

case class LoadTestQueryResults(

numResults: Int,

top1Recall: Float,

top10Recall: Option[Float],

overallRecall: Float)

private object LoadTestQueryRecorder {

def recordQueryResult[T](

trueNeighbors: Seq[T],

foundNeighbors: Seq[T]

): LoadTestQueryResults = {

// record number of results returned

val numResults = foundNeighbors.size

if (trueNeighbors.isEmpty) {

LoadTestQueryResults(

numResults,

0f,

Option.empty,

0f

)

} else {

// record top 1, top 10 and overall recall

// recall here is computed as number of true neighbors within the returned points set

// divides by the number of required neighbors

val top1Recall = foundNeighbors.intersect(Seq(trueNeighbors.head)).size

val top10Recall = if (numResults >= 10 && trueNeighbors.size >= 10) {

Some(

trueNeighbors.take(10).intersect(foundNeighbors).size.toFloat / 10

)

} else {

None

}

val overallRecall = trueNeighbors

.take(foundNeighbors.size).intersect(foundNeighbors).size.toFloat /

Math.min(foundNeighbors.size, trueNeighbors.size)

LoadTestQueryResults(

numResults,

top1Recall,

top10Recall,

overallRecall

)

}

}

}

class StatsLoadTestQueryRecorder[T](

statsReceiver: StatsReceiver)

extends LoadTestQueryRecorder[T] {

private[this] val numResultsStats = statsReceiver.stat("number\_of\_results")

private[this] val recallStats = statsReceiver.stat("recall")

private[this] val top1RecallStats = statsReceiver.stat("top\_1\_recall")

private[this] val top10RecallStats = statsReceiver.stat("top\_10\_recall")

private[this] val queryLatencyMicrosStats = statsReceiver.stat("query\_latency\_micros")

override def recordQueryResult(

trueNeighbors: Seq[T],

foundNeighbors: Seq[T],

queryLatency: Duration

): Unit = {

val results = LoadTestQueryRecorder.recordQueryResult(trueNeighbors, foundNeighbors)

numResultsStats.add(results.numResults)

recallStats.add(results.overallRecall \* 100)

results.top10Recall.foreach { top10Recall =>

top10RecallStats.add(top10Recall \* 100)

}

top1RecallStats.add(results.top1Recall \* 100)

queryLatencyMicrosStats.add(queryLatency.inMicroseconds)

}

}

trait LoadTestBuildRecorder {

def recordIndexCreation(

indexSize: Int,

indexLatency: Duration,

toQueryableLatency: Duration

): Unit

}

class StatsLoadTestBuildRecorder(

statsReceiver: StatsReceiver)

extends LoadTestBuildRecorder {

private[this] val indexLatencyGauge = statsReceiver.addGauge("index\_latency\_ms")(\_)

private[this] val indexSizeGauge = statsReceiver.addGauge("index\_size")(\_)

private[this] val toQueryableGauge = statsReceiver.addGauge("to\_queryable\_latency\_ms")(\_)

override def recordIndexCreation(

indexSize: Int,

indexLatency: Duration,

toQueryableLatency: Duration

): Unit = {

indexLatencyGauge(indexLatency.inMillis)

indexSizeGauge(indexSize)

toQueryableGauge(toQueryableLatency.inMillis)

}

}

class QueryRecorderSnapshot(snapshot: Snapshot) {

def avgQueryLatencyMicros: Double = snapshot.average

def p50QueryLatencyMicros: Double =

snapshot.percentiles.find(\_.quantile == .5).get.value

def p90QueryLatencyMicros: Double =

snapshot.percentiles.find(\_.quantile == .9).get.value

def p99QueryLatencyMicros: Double =

snapshot.percentiles.find(\_.quantile == .99).get.value

}

class InMemoryLoadTestQueryRecorder[T](

// You have to specify a name of the histogram even though it is not used

// Use latch period of bottom. We will compute a new snapshot every time we call computeSnapshot

private[this] val latencyHistogram: MetricsBucketedHistogram =

new MetricsBucketedHistogram("latencyhistogram", latchPeriod = Duration.Bottom))

extends LoadTestQueryRecorder[T] {

private[this] val counter = new AtomicInteger(0)

private[this] val countMoreThan10Results = new AtomicInteger(0)

private[this] val recallSum = new AtomicDouble(0.0)

private[this] val top1RecallSum = new AtomicDouble(0.0)

private[this] val top10RecallSum = new AtomicDouble(0.0)

private[this] val elapsedTimeFun = new AtomicReference[(Stopwatch.Elapsed, Duration)]()

private[this] val elapsedTime = new AtomicReference[Duration](Duration.Zero)

/\*\*

\* Compute a snapshot of what happened between the time that this was called and the previous time

\* it was called.

\* @return

\*/

def computeSnapshot(): QueryRecorderSnapshot = {

new QueryRecorderSnapshot(latencyHistogram.snapshot())

}

def recall: Double =

if (counter.get() != 0) {

recallSum.get \* 100 / counter.get()

} else { 0 }

def top1Recall: Double =

if (counter.get() != 0) {

top1RecallSum.get \* 100 / counter.get()

} else { 0 }

def top10Recall: Double =

if (countMoreThan10Results.get() != 0) {

top10RecallSum.get \* 100 / countMoreThan10Results.get()

} else { 0 }

def avgRPS: Double =

if (elapsedTime.get() != Duration.Zero) {

(counter.get().toDouble \* 1e9) / elapsedTime.get().inNanoseconds

} else { 0 }

override def recordQueryResult(

trueNeighbors: Seq[T],

foundNeighbors: Seq[T],

queryLatency: Duration

): Unit = {

elapsedTimeFun.compareAndSet(null, (Stopwatch.start(), queryLatency))

val results = LoadTestQueryRecorder.recordQueryResult(trueNeighbors, foundNeighbors)

top1RecallSum.addAndGet(results.top1Recall)

results.top10Recall.foreach { top10Recall =>

top10RecallSum.addAndGet(top10Recall)

countMoreThan10Results.incrementAndGet()

}

recallSum.addAndGet(results.overallRecall)

latencyHistogram.add(queryLatency.inMicroseconds)

counter.incrementAndGet()

// Requests are assumed to have started around the time time of the first time record was called

// plus the time it took for that query to hhave completed.

val (elapsedSinceFirstCall, firstQueryLatency) = elapsedTimeFun.get()

val durationSoFar = elapsedSinceFirstCall() + firstQueryLatency

elapsedTime.set(durationSoFar)

}

}

class InMemoryLoadTestBuildRecorder extends LoadTestBuildRecorder {

var indexLatency: Duration = Duration.Zero

var indexSize: Int = 0

var toQueryableLatency: Duration = Duration.Zero

override def recordIndexCreation(

size: Int,

indexLatencyArg: Duration,

toQueryableLatencyArg: Duration

): Unit = {

indexLatency = indexLatencyArg

indexSize = size

toQueryableLatency = toQueryableLatencyArg

}

}

/\*\*

\* A LoadTestRecorder that be composed by other recorders

\*/

class ComposedLoadTestQueryRecorder[T](

recorders: Seq[LoadTestQueryRecorder[T]])

extends LoadTestQueryRecorder[T] {

override def recordQueryResult(

trueNeighbors: Seq[T],

foundNeighbors: Seq[T],

queryLatency: Duration

): Unit = recorders.foreach {

\_.recordQueryResult(trueNeighbors, foundNeighbors, queryLatency)

}

}

/\*\*

\* A LoadTestRecorder that be composed by other recorders

\*/

class ComposedLoadTestBuildRecorder(

recorders: Seq[LoadTestBuildRecorder])

extends LoadTestBuildRecorder {

override def recordIndexCreation(

indexSize: Int,

indexLatency: Duration,

toQueryableLatency: Duration

): Unit = recorders.foreach { \_.recordIndexCreation(indexSize, indexLatency, toQueryableLatency) }

}