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

import com.google.common.util.concurrent.MoreExecutors

import com.google.inject.Module

import com.twitter.ann.common.\_

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

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

import com.twitter.app.Flag

import com.twitter.bijection.Injection

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

import com.twitter.finatra.thrift.routing.ThriftRouter

import java.util.concurrent.ExecutorService

import java.util.concurrent.Executors

import java.util.concurrent.ThreadPoolExecutor

import java.util.concurrent.TimeUnit

/\*\*

\* This class is used when you do not know the generic parameters of the Server at compile time.

\* If you want compile time checks that your parameters are correct use QueryIndexServer instead.

\* In particular, when you want to have these id, distance and the runtime params as cli options you

\* should extend this class.

\*/

abstract class UnsafeQueryIndexServer[R <: RuntimeParams] extends BaseQueryIndexServer {

private[this] val metricName = flag[String]("metric", "metric")

private[this] val idType = flag[String]("id\_type", "type of ids to use")

private[query\_server] val queryThreads =

flag[Int](

"threads",

System

.getProperty("mesos.resources.cpu", s"${Runtime.getRuntime.availableProcessors()}").toInt,

"Size of thread pool for concurrent querying"

)

private[query\_server] val dimension = flag[Int]("dimension", "dimension")

private[query\_server] val indexDirectory = flag[String]("index\_directory", "index directory")

private[query\_server] val refreshable =

flag[Boolean]("refreshable", false, "if index is refreshable or not")

private[query\_server] val refreshableInterval =

flag[Int]("refreshable\_interval\_minutes", 10, "refreshable index update interval")

private[query\_server] val sharded =

flag[Boolean]("sharded", false, "if index is sharded")

private[query\_server] val shardedHours =

flag[Int]("sharded\_hours", "how many shards load at once")

private[query\_server] val shardedWatchLookbackIndexes =

flag[Int]("sharded\_watch\_lookback\_indexes", "how many indexes backwards to watch")

private[query\_server] val shardedWatchIntervalMinutes =

flag[Int]("sharded\_watch\_interval\_minutes", "interval at which hdfs is watched for changes")

private[query\_server] val minIndexSizeBytes =

flag[Long]("min\_index\_size\_byte", 0, "min index size in bytes")

private[query\_server] val maxIndexSizeBytes =

flag[Long]("max\_index\_size\_byte", Long.MaxValue, "max index size in bytes")

private[query\_server] val grouped =

flag[Boolean]("grouped", false, "if indexes are grouped")

private[query\_server] val qualityFactorEnabled =

flag[Boolean](

"quality\_factor\_enabled",

false,

"Enable dynamically reducing search complexity when cgroups container is throttled. Useful to disable when load testing"

)

private[query\_server] val warmup\_enabled: Flag[Boolean] =

flag("warmup", false, "Enable warmup before the query server starts up")

// Time to wait for the executor to finish before terminating the JVM

private[this] val terminationTimeoutMs = 100

protected lazy val executor: ExecutorService = MoreExecutors.getExitingExecutorService(

Executors.newFixedThreadPool(queryThreads()).asInstanceOf[ThreadPoolExecutor],

terminationTimeoutMs,

TimeUnit.MILLISECONDS

)

protected lazy val unsafeMetric: Metric[\_] with Injection[\_, ServiceDistance] = {

Metric.fromString(metricName())

}

override protected val additionalModules: Seq[Module] = Seq()

override final def addController(router: ThriftRouter): Unit = {

router.add(queryIndexThriftController)

}

protected def unsafeQueryableMap[T, D <: Distance[D]]: Queryable[T, R, D]

protected val runtimeInjection: Injection[R, ServiceRuntimeParams]

private[this] def queryIndexThriftController[

T,

D <: Distance[D]

]: QueryIndexThriftController[T, R, D] = {

val controller = new QueryIndexThriftController[T, R, D](

statsReceiver.scope("ann\_server"),

unsafeQueryableMap.asInstanceOf[Queryable[T, R, D]],

runtimeInjection,

unsafeMetric.asInstanceOf[Injection[D, ServiceDistance]],

idInjection[T]()

)

logger.info("QueryIndexThriftController created....")

controller

}

protected final def idInjection[T](): Injection[T, Array[Byte]] = {

val idInjection = idType() match {

case "string" => AnnInjections.StringInjection

case "long" => AnnInjections.LongInjection

case "int" => AnnInjections.IntInjection

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

}

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

}

}