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

import com.twitter.ann.common.Distance

import com.twitter.ann.common.QueryableOperations.Map

import com.twitter.ann.common.\_

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

import com.twitter.ann.faiss.FaissCommon

import com.twitter.ann.faiss.FaissIndex

import com.twitter.ann.faiss.FaissParams

import com.twitter.ann.faiss.HourlyShardedIndex

import com.twitter.ann.service.query\_server.common.QueryableProvider

import com.twitter.ann.service.query\_server.common.RefreshableQueryable

import com.twitter.ann.service.query\_server.common.UnsafeQueryIndexServer

import com.twitter.ann.service.query\_server.common.FaissIndexPathProvider

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

import com.twitter.ann.service.query\_server.common.warmup.Warmup

import com.twitter.bijection.Injection

import com.twitter.conversions.DurationOps.richDurationFromInt

import com.twitter.search.common.file.AbstractFile

import com.twitter.search.common.file.FileUtils

import com.twitter.util.Duration

import java.util.concurrent.TimeUnit

object FaissQueryIndexServer extends FaissQueryableServer

class FaissQueryableServer extends UnsafeQueryIndexServer[FaissParams] {

// given a directory, how to load it as a queryable index

def queryableProvider[T, D <: Distance[D]]: QueryableProvider[T, FaissParams, D] =

new QueryableProvider[T, FaissParams, D] {

override def provideQueryable(

directory: AbstractFile

): Queryable[T, FaissParams, D] = {

FaissIndex.loadIndex[T, D](

dimension(),

unsafeMetric.asInstanceOf[Metric[D]],

directory

)

}

}

private def buildSimpleQueryable[T, D <: Distance[D]](

dir: AbstractFile

): Queryable[T, FaissParams, D] = {

val queryable = if (refreshable()) {

logger.info(s"build refreshable queryable")

val updatableQueryable = new RefreshableQueryable(

false,

dir,

queryableProvider.asInstanceOf[QueryableProvider[T, FaissParams, D]],

FaissIndexPathProvider(

minIndexSizeBytes(),

maxIndexSizeBytes(),

statsReceiver.scope("validated\_index\_provider")

),

statsReceiver.scope("refreshable\_queryable"),

updateInterval = refreshableInterval().minutes

)

// init first load of index and also schedule the following reloads

updatableQueryable.start()

updatableQueryable.asInstanceOf[QueryableGrouped[T, FaissParams, D]]

} else {

logger.info(s"build non-refreshable queryable")

logger.info(s"Loading ${dir}")

queryableProvider.provideQueryable(dir).asInstanceOf[Queryable[T, FaissParams, D]]

}

logger.info("Faiss queryable created....")

queryable

}

private def buildShardedQueryable[T, D <: Distance[D]](

dir: AbstractFile

): Queryable[T, FaissParams, D] = {

logger.info(s"build sharded queryable")

val queryable = HourlyShardedIndex.loadIndex[T, D](

dimension(),

unsafeMetric.asInstanceOf[Metric[D]],

dir,

shardedHours(),

Duration(shardedWatchIntervalMinutes(), TimeUnit.MINUTES),

shardedWatchLookbackIndexes(),

statsReceiver.scope("hourly\_sharded\_index")

)

logger.info("Faiss sharded queryable created....")

closeOnExit(queryable)

queryable.startImmediately()

logger.info("Directory watching is scheduled")

queryable

}

// Readings come incorrect if reader is created too early in the lifecycle of a server

// hence lazy

private lazy val throttleSamplingTask = new ThrottlingBasedQualityTask(

statsReceiver.scope("throttling\_task"))

override def unsafeQueryableMap[T, D <: Distance[D]]: Queryable[T, FaissParams, D] = {

val dir = FileUtils.getFileHandle(indexDirectory())

val queryable = if (sharded()) {

require(shardedHours() > 0, "Number of hourly shards must be specified")

require(shardedWatchIntervalMinutes() > 0, "Shard watch interval must be specified")

require(shardedWatchLookbackIndexes() > 0, "Index lookback must be specified")

buildShardedQueryable[T, D](dir)

} else {

buildSimpleQueryable[T, D](dir)

}

if (qualityFactorEnabled()) {

logger.info("Quality Factor throttling is enabled")

closeOnExit(throttleSamplingTask)

throttleSamplingTask.jitteredStart()

queryable.mapRuntimeParameters(throttleSamplingTask.discountParams)

} else {

queryable

}

}

override val runtimeInjection: Injection[FaissParams, ServiceRuntimeParams] =

FaissCommon.RuntimeParamsInjection

protected override def warmup(): Unit =

if (warmup\_enabled())

new FaissWarmup(unsafeQueryableMap, dimension()).warmup()

}

class FaissWarmup(faiss: Queryable[\_, FaissParams, \_], dimension: Int) extends Warmup {

protected def minSuccessfulTries: Int = 100

protected def maxTries: Int = 1000

protected def timeout: Duration = 50.milliseconds

protected def randomQueryDimension: Int = dimension

def warmup(): Unit = {

run(

name = "queryWithDistance",

f = faiss

.queryWithDistance(

randomQuery(),

100,

FaissParams(nprobe = Some(128), None, None, None, None))

)

}

}