package com.twitter.representationscorer.columns

import com.twitter.representationscorer.thriftscala.ListScoreId

import com.twitter.representationscorer.thriftscala.ListScoreResponse

import com.twitter.representationscorer.scorestore.ScoreStore

import com.twitter.representationscorer.thriftscala.ScoreResult

import com.twitter.simclusters\_v2.common.SimClustersEmbeddingId.LongInternalId

import com.twitter.simclusters\_v2.common.SimClustersEmbeddingId.LongSimClustersEmbeddingId

import com.twitter.simclusters\_v2.thriftscala.Score

import com.twitter.simclusters\_v2.thriftscala.ScoreId

import com.twitter.simclusters\_v2.thriftscala.ScoreInternalId

import com.twitter.simclusters\_v2.thriftscala.SimClustersEmbeddingId

import com.twitter.simclusters\_v2.thriftscala.SimClustersEmbeddingPairScoreId

import com.twitter.stitch

import com.twitter.stitch.Stitch

import com.twitter.strato.catalog.OpMetadata

import com.twitter.strato.config.ContactInfo

import com.twitter.strato.config.Policy

import com.twitter.strato.data.Conv

import com.twitter.strato.data.Description.PlainText

import com.twitter.strato.data.Lifecycle

import com.twitter.strato.fed.\_

import com.twitter.strato.thrift.ScroogeConv

import com.twitter.util.Future

import com.twitter.util.Return

import com.twitter.util.Throw

import javax.inject.Inject

class ListScoreColumn @Inject() (scoreStore: ScoreStore)

extends StratoFed.Column("recommendations/representation\_scorer/listScore")

with StratoFed.Fetch.Stitch {

override val policy: Policy = Common.rsxReadPolicy

override type Key = ListScoreId

override type View = Unit

override type Value = ListScoreResponse

override val keyConv: Conv[Key] = ScroogeConv.fromStruct[ListScoreId]

override val viewConv: Conv[View] = Conv.ofType

override val valueConv: Conv[Value] = ScroogeConv.fromStruct[ListScoreResponse]

override val contactInfo: ContactInfo = Info.contactInfo

override val metadata: OpMetadata = OpMetadata(

lifecycle = Some(Lifecycle.Production),

description = Some(

PlainText(

"Scoring for multiple candidate entities against a single target entity"

))

)

override def fetch(key: Key, view: View): Stitch[Result[Value]] = {

val target = SimClustersEmbeddingId(

embeddingType = key.targetEmbeddingType,

modelVersion = key.modelVersion,

internalId = key.targetId

)

val scoreIds = key.candidateIds.map { candidateId =>

val candidate = SimClustersEmbeddingId(

embeddingType = key.candidateEmbeddingType,

modelVersion = key.modelVersion,

internalId = candidateId

)

ScoreId(

algorithm = key.algorithm,

internalId = ScoreInternalId.SimClustersEmbeddingPairScoreId(

SimClustersEmbeddingPairScoreId(target, candidate)

)

)

}

Stitch

.callFuture {

val (keys: Iterable[ScoreId], vals: Iterable[Future[Option[Score]]]) =

scoreStore.uniformScoringStore.multiGet(scoreIds.toSet).unzip

val results: Future[Iterable[Option[Score]]] = Future.collectToTry(vals.toSeq) map {

tryOptVals =>

tryOptVals map {

case Return(Some(v)) => Some(v)

case Return(None) => None

case Throw(\_) => None

}

}

val scoreMap: Future[Map[Long, Double]] = results.map { scores =>

keys

.zip(scores).collect {

case (

ScoreId(

\_,

ScoreInternalId.SimClustersEmbeddingPairScoreId(

SimClustersEmbeddingPairScoreId(

\_,

LongSimClustersEmbeddingId(candidateId)))),

Some(score)) =>

(candidateId, score.score)

}.toMap

}

scoreMap

}

.map { (scores: Map[Long, Double]) =>

val orderedScores = key.candidateIds.collect {

case LongInternalId(id) => ScoreResult(scores.get(id))

case \_ =>

// This will return None scores for candidates which don't have Long ids, but that's fine:

// at the moment we're only scoring for Tweets

ScoreResult(None)

}

found(ListScoreResponse(orderedScores))

}

.handle {

case stitch.NotFound => missing

}

}

}