package com.twitter.cr\_mixer.logging

import com.google.common.base.CaseFormat

import com.twitter.abdecider.ScribingABDeciderUtil

import com.twitter.scribelib.marshallers.ClientDataProvider

import com.twitter.scribelib.marshallers.ScribeSerialization

import com.twitter.timelines.clientevent.MinimalClientDataProvider

import com.twitter.cr\_mixer.model.BlendedCandidate

import com.twitter.cr\_mixer.model.CrCandidateGeneratorQuery

import com.twitter.cr\_mixer.model.InitialCandidate

import com.twitter.cr\_mixer.model.RankedCandidate

import com.twitter.cr\_mixer.logging.ScribeLoggerUtils.\_

import com.twitter.cr\_mixer.model.GraphSourceInfo

import com.twitter.cr\_mixer.param.decider.CrMixerDecider

import com.twitter.cr\_mixer.param.decider.DeciderConstants

import com.twitter.cr\_mixer.scribe.ScribeCategories

import com.twitter.cr\_mixer.thriftscala.CrMixerTweetRequest

import com.twitter.cr\_mixer.thriftscala.CrMixerTweetResponse

import com.twitter.cr\_mixer.thriftscala.FetchCandidatesResult

import com.twitter.cr\_mixer.thriftscala.FetchSignalSourcesResult

import com.twitter.cr\_mixer.thriftscala.GetTweetsRecommendationsScribe

import com.twitter.cr\_mixer.thriftscala.InterleaveResult

import com.twitter.cr\_mixer.thriftscala.PerformanceMetrics

import com.twitter.cr\_mixer.thriftscala.PreRankFilterResult

import com.twitter.cr\_mixer.thriftscala.Product

import com.twitter.cr\_mixer.thriftscala.RankResult

import com.twitter.cr\_mixer.thriftscala.Result

import com.twitter.cr\_mixer.thriftscala.SourceSignal

import com.twitter.cr\_mixer.thriftscala.TopLevelApiResult

import com.twitter.cr\_mixer.thriftscala.TweetCandidateWithMetadata

import com.twitter.cr\_mixer.thriftscala.VITTweetCandidateScribe

import com.twitter.cr\_mixer.thriftscala.VITTweetCandidatesScribe

import com.twitter.cr\_mixer.model.ModuleNames

import com.twitter.cr\_mixer.model.SourceInfo

import com.twitter.cr\_mixer.util.CandidateGenerationKeyUtil

import com.twitter.cr\_mixer.util.MetricTagUtil

import com.twitter.decider.SimpleRecipient

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.finagle.tracing.Trace

import com.twitter.finatra.kafka.producers.KafkaProducerBase

import com.twitter.logging.Logger

import com.twitter.simclusters\_v2.common.UserId

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

import com.twitter.util.Future

import com.twitter.util.Stopwatch

import com.twitter.util.Time

import javax.inject.Inject

import javax.inject.Named

import javax.inject.Singleton

import scala.util.Random

@Singleton

case class CrMixerScribeLogger @Inject() (

decider: CrMixerDecider,

statsReceiver: StatsReceiver,

@Named(ModuleNames.TweetRecsLogger) tweetRecsScribeLogger: Logger,

@Named(ModuleNames.BlueVerifiedTweetRecsLogger) blueVerifiedTweetRecsScribeLogger: Logger,

@Named(ModuleNames.TopLevelApiDdgMetricsLogger) ddgMetricsLogger: Logger,

kafkaProducer: KafkaProducerBase[String, GetTweetsRecommendationsScribe]) {

import CrMixerScribeLogger.\_

private val scopedStats = statsReceiver.scope("CrMixerScribeLogger")

private val topLevelApiStats = scopedStats.scope("TopLevelApi")

private val upperFunnelsStats = scopedStats.scope("UpperFunnels")

private val kafkaMessagesStats = scopedStats.scope("KafkaMessages")

private val topLevelApiDdgMetricsStats = scopedStats.scope("TopLevelApiDdgMetrics")

private val blueVerifiedTweetCandidatesStats = scopedStats.scope("BlueVerifiedTweetCandidates")

private val serialization = new ScribeSerialization {}

def scribeSignalSources(

query: CrCandidateGeneratorQuery,

getResultFn: => Future[(Set[SourceInfo], Map[String, Option[GraphSourceInfo]])]

): Future[(Set[SourceInfo], Map[String, Option[GraphSourceInfo]])] = {

scribeResultsAndPerformanceMetrics(

ScribeMetadata.from(query),

getResultFn,

convertToResultFn = convertFetchSignalSourcesResult

)

}

def scribeInitialCandidates(

query: CrCandidateGeneratorQuery,

getResultFn: => Future[Seq[Seq[InitialCandidate]]]

): Future[Seq[Seq[InitialCandidate]]] = {

scribeResultsAndPerformanceMetrics(

ScribeMetadata.from(query),

getResultFn,

convertToResultFn = convertFetchCandidatesResult

)

}

def scribePreRankFilterCandidates(

query: CrCandidateGeneratorQuery,

getResultFn: => Future[Seq[Seq[InitialCandidate]]]

): Future[Seq[Seq[InitialCandidate]]] = {

scribeResultsAndPerformanceMetrics(

ScribeMetadata.from(query),

getResultFn,

convertToResultFn = convertPreRankFilterResult

)

}

def scribeInterleaveCandidates(

query: CrCandidateGeneratorQuery,

getResultFn: => Future[Seq[BlendedCandidate]]

): Future[Seq[BlendedCandidate]] = {

scribeResultsAndPerformanceMetrics(

ScribeMetadata.from(query),

getResultFn,

convertToResultFn = convertInterleaveResult,

enableKafkaScribe = true

)

}

def scribeRankedCandidates(

query: CrCandidateGeneratorQuery,

getResultFn: => Future[Seq[RankedCandidate]]

): Future[Seq[RankedCandidate]] = {

scribeResultsAndPerformanceMetrics(

ScribeMetadata.from(query),

getResultFn,

convertToResultFn = convertRankResult

)

}

/\*\*

\* Scribe Top Level API Request / Response and performance metrics

\* for the getTweetRecommendations() endpoint.

\*/

def scribeGetTweetRecommendations(

request: CrMixerTweetRequest,

startTime: Long,

scribeMetadata: ScribeMetadata,

getResultFn: => Future[CrMixerTweetResponse]

): Future[CrMixerTweetResponse] = {

val timer = Stopwatch.start()

getResultFn.onSuccess { response =>

val latencyMs = timer().inMilliseconds

val result = convertTopLevelAPIResult(request, response, startTime)

val traceId = Trace.id.traceId.toLong

val scribeMsg = buildScribeMessage(result, scribeMetadata, latencyMs, traceId)

// We use upperFunnelPerStepScribeRate to cover TopLevelApi scribe logs

if (decider.isAvailableForId(

scribeMetadata.userId,

DeciderConstants.upperFunnelPerStepScribeRate)) {

topLevelApiStats.counter(scribeMetadata.product.originalName).incr()

scribeResult(scribeMsg)

}

if (decider.isAvailableForId(

scribeMetadata.userId,

DeciderConstants.topLevelApiDdgMetricsScribeRate)) {

topLevelApiDdgMetricsStats.counter(scribeMetadata.product.originalName).incr()

val topLevelDdgMetricsMetadata = TopLevelDdgMetricsMetadata.from(request)

publishTopLevelDdgMetrics(

logger = ddgMetricsLogger,

topLevelDdgMetricsMetadata = topLevelDdgMetricsMetadata,

latencyMs = latencyMs,

candidateSize = response.tweets.length)

}

}

}

/\*\*

\* Scribe all of the Blue Verified tweets that are candidates from cr-mixer

\* from the getTweetRecommendations() endpoint for stats tracking/debugging purposes.

\*/

def scribeGetTweetRecommendationsForBlueVerified(

scribeMetadata: ScribeMetadata,

getResultFn: => Future[Seq[RankedCandidate]]

): Future[Seq[RankedCandidate]] = {

getResultFn.onSuccess { rankedCandidates =>

if (decider.isAvailable(DeciderConstants.enableScribeForBlueVerifiedTweetCandidates)) {

blueVerifiedTweetCandidatesStats.counter("process\_request").incr()

val blueVerifiedTweetCandidates = rankedCandidates.filter { tweet =>

tweet.tweetInfo.hasBlueVerifiedAnnotation.contains(true)

}

val impressedBuckets = getImpressedBuckets(blueVerifiedTweetCandidatesStats).getOrElse(Nil)

val blueVerifiedCandidateScribes = blueVerifiedTweetCandidates.map { candidate =>

blueVerifiedTweetCandidatesStats

.scope(scribeMetadata.product.name).counter(

candidate.tweetInfo.authorId.toString).incr()

VITTweetCandidateScribe(

tweetId = candidate.tweetId,

authorId = candidate.tweetInfo.authorId,

score = candidate.predictionScore,

metricTags = MetricTagUtil.buildMetricTags(candidate)

)

}

val blueVerifiedScribe =

VITTweetCandidatesScribe(

uuid = scribeMetadata.requestUUID,

userId = scribeMetadata.userId,

candidates = blueVerifiedCandidateScribes,

product = scribeMetadata.product,

impressedBuckets = impressedBuckets

)

publish(

logger = blueVerifiedTweetRecsScribeLogger,

codec = VITTweetCandidatesScribe,

message = blueVerifiedScribe)

}

}

}

/\*\*

\* Scribe Per-step intermediate results and performance metrics

\* for each step: fetch signals, fetch candidates, filters, ranker, etc

\*/

private[logging] def scribeResultsAndPerformanceMetrics[T](

scribeMetadata: ScribeMetadata,

getResultFn: => Future[T],

convertToResultFn: (T, UserId) => Result,

enableKafkaScribe: Boolean = false

): Future[T] = {

val timer = Stopwatch.start()

getResultFn.onSuccess { input =>

val latencyMs = timer().inMilliseconds

val result = convertToResultFn(input, scribeMetadata.userId)

val traceId = Trace.id.traceId.toLong

val scribeMsg = buildScribeMessage(result, scribeMetadata, latencyMs, traceId)

if (decider.isAvailableForId(

scribeMetadata.userId,

DeciderConstants.upperFunnelPerStepScribeRate)) {

upperFunnelsStats.counter(scribeMetadata.product.originalName).incr()

scribeResult(scribeMsg)

}

// forks the scribe as a Kafka message for async feature hydration

if (enableKafkaScribe && shouldScribeKafkaMessage(

scribeMetadata.userId,

scribeMetadata.product)) {

kafkaMessagesStats.counter(scribeMetadata.product.originalName).incr()

val batchedKafkaMessages = downsampleKafkaMessage(scribeMsg)

batchedKafkaMessages.foreach { kafkaMessage =>

kafkaProducer.send(

topic = ScribeCategories.TweetsRecs.scribeCategory,

key = traceId.toString,

value = kafkaMessage,

timestamp = Time.now.inMilliseconds

)

}

}

}

}

private def convertTopLevelAPIResult(

request: CrMixerTweetRequest,

response: CrMixerTweetResponse,

startTime: Long

): Result = {

Result.TopLevelApiResult(

TopLevelApiResult(

timestamp = startTime,

request = request,

response = response

))

}

private def convertFetchSignalSourcesResult(

sourceInfoSetTuple: (Set[SourceInfo], Map[String, Option[GraphSourceInfo]]),

requestUserId: UserId

): Result = {

val sourceSignals = sourceInfoSetTuple.\_1.map { sourceInfo =>

SourceSignal(id = Some(sourceInfo.internalId))

}

// For source graphs, we pass in requestUserId as a placeholder

val sourceGraphs = sourceInfoSetTuple.\_2.map {

case (\_, \_) =>

SourceSignal(id = Some(InternalId.UserId(requestUserId)))

}

Result.FetchSignalSourcesResult(

FetchSignalSourcesResult(

signals = Some(sourceSignals ++ sourceGraphs)

))

}

private def convertFetchCandidatesResult(

candidatesSeq: Seq[Seq[InitialCandidate]],

requestUserId: UserId

): Result = {

val tweetCandidatesWithMetadata = candidatesSeq.flatMap { candidates =>

candidates.map { candidate =>

TweetCandidateWithMetadata(

tweetId = candidate.tweetId,

candidateGenerationKey = Some(

CandidateGenerationKeyUtil.toThrift(candidate.candidateGenerationInfo, requestUserId)),

score = Some(candidate.getSimilarityScore),

numCandidateGenerationKeys = None // not populated yet

)

}

}

Result.FetchCandidatesResult(FetchCandidatesResult(Some(tweetCandidatesWithMetadata)))

}

private def convertPreRankFilterResult(

candidatesSeq: Seq[Seq[InitialCandidate]],

requestUserId: UserId

): Result = {

val tweetCandidatesWithMetadata = candidatesSeq.flatMap { candidates =>

candidates.map { candidate =>

TweetCandidateWithMetadata(

tweetId = candidate.tweetId,

candidateGenerationKey = Some(

CandidateGenerationKeyUtil.toThrift(candidate.candidateGenerationInfo, requestUserId)),

score = Some(candidate.getSimilarityScore),

numCandidateGenerationKeys = None // not populated yet

)

}

}

Result.PreRankFilterResult(PreRankFilterResult(Some(tweetCandidatesWithMetadata)))

}

// We take InterleaveResult for Unconstrained dataset ML ranker training

private def convertInterleaveResult(

blendedCandidates: Seq[BlendedCandidate],

requestUserId: UserId

): Result = {

val tweetCandidatesWithMetadata = blendedCandidates.map { blendedCandidate =>

val candidateGenerationKey =

CandidateGenerationKeyUtil.toThrift(blendedCandidate.reasonChosen, requestUserId)

TweetCandidateWithMetadata(

tweetId = blendedCandidate.tweetId,

candidateGenerationKey = Some(candidateGenerationKey),

authorId = Some(blendedCandidate.tweetInfo.authorId), // for ML pipeline training

score = Some(blendedCandidate.getSimilarityScore),

numCandidateGenerationKeys = Some(blendedCandidate.potentialReasons.size)

) // hydrate fields for light ranking training data

}

Result.InterleaveResult(InterleaveResult(Some(tweetCandidatesWithMetadata)))

}

private def convertRankResult(

rankedCandidates: Seq[RankedCandidate],

requestUserId: UserId

): Result = {

val tweetCandidatesWithMetadata = rankedCandidates.map { rankedCandidate =>

val candidateGenerationKey =

CandidateGenerationKeyUtil.toThrift(rankedCandidate.reasonChosen, requestUserId)

TweetCandidateWithMetadata(

tweetId = rankedCandidate.tweetId,

candidateGenerationKey = Some(candidateGenerationKey),

score = Some(rankedCandidate.getSimilarityScore),

numCandidateGenerationKeys = Some(rankedCandidate.potentialReasons.size)

)

}

Result.RankResult(RankResult(Some(tweetCandidatesWithMetadata)))

}

private def buildScribeMessage(

result: Result,

scribeMetadata: ScribeMetadata,

latencyMs: Long,

traceId: Long

): GetTweetsRecommendationsScribe = {

GetTweetsRecommendationsScribe(

uuid = scribeMetadata.requestUUID,

userId = scribeMetadata.userId,

result = result,

traceId = Some(traceId),

performanceMetrics = Some(PerformanceMetrics(Some(latencyMs))),

impressedBuckets = getImpressedBuckets(scopedStats)

)

}

private def scribeResult(

scribeMsg: GetTweetsRecommendationsScribe

): Unit = {

publish(

logger = tweetRecsScribeLogger,

codec = GetTweetsRecommendationsScribe,

message = scribeMsg)

}

/\*\*

\* Gate for producing messages to Kafka for async feature hydration

\*/

private def shouldScribeKafkaMessage(

userId: UserId,

product: Product

): Boolean = {

val isEligibleUser = decider.isAvailable(

DeciderConstants.kafkaMessageScribeSampleRate,

Some(SimpleRecipient(userId)))

val isHomeProduct = (product == Product.Home)

isEligibleUser && isHomeProduct

}

/\*\*

\* Due to size limits of Strato (see SD-19028), each Kafka message must be downsampled

\*/

private[logging] def downsampleKafkaMessage(

scribeMsg: GetTweetsRecommendationsScribe

): Seq[GetTweetsRecommendationsScribe] = {

val sampledResultSeq: Seq[Result] = scribeMsg.result match {

case Result.InterleaveResult(interleaveResult) =>

val sampledTweetsSeq = interleaveResult.tweets

.map { tweets =>

Random

.shuffle(tweets).take(KafkaMaxTweetsPerMessage)

.grouped(BatchSize).toSeq

}.getOrElse(Seq.empty)

sampledTweetsSeq.map { sampledTweets =>

Result.InterleaveResult(InterleaveResult(Some(sampledTweets)))

}

// if it's an unrecognized type, err on the side of sending no candidates

case \_ =>

kafkaMessagesStats.counter("InvalidKafkaMessageResultType").incr()

Seq(Result.InterleaveResult(InterleaveResult(None)))

}

sampledResultSeq.map { sampledResult =>

GetTweetsRecommendationsScribe(

uuid = scribeMsg.uuid,

userId = scribeMsg.userId,

result = sampledResult,

traceId = scribeMsg.traceId,

performanceMetrics = None,

impressedBuckets = None

)

}

}

/\*\*

\* Handles client\_event serialization to log data into DDG metrics

\*/

private[logging] def publishTopLevelDdgMetrics(

logger: Logger,

topLevelDdgMetricsMetadata: TopLevelDdgMetricsMetadata,

candidateSize: Long,

latencyMs: Long,

): Unit = {

val data = Map[Any, Any](

"latency\_ms" -> latencyMs,

"event\_value" -> candidateSize

)

val label: (String, String) = ("tweetrec", "")

val namespace = getNamespace(topLevelDdgMetricsMetadata, label) + ("action" -> "candidates")

val message =

serialization

.serializeClientEvent(namespace, getClientData(topLevelDdgMetricsMetadata), data)

logger.info(message)

}

private def getClientData(

topLevelDdgMetricsMetadata: TopLevelDdgMetricsMetadata

): ClientDataProvider =

MinimalClientDataProvider(

userId = topLevelDdgMetricsMetadata.userId,

guestId = None,

clientApplicationId = topLevelDdgMetricsMetadata.clientApplicationId,

countryCode = topLevelDdgMetricsMetadata.countryCode

)

private def getNamespace(

topLevelDdgMetricsMetadata: TopLevelDdgMetricsMetadata,

label: (String, String)

): Map[String, String] = {

val productName =

CaseFormat.UPPER\_CAMEL

.to(CaseFormat.LOWER\_UNDERSCORE, topLevelDdgMetricsMetadata.product.originalName)

Map(

"client" -> ScribingABDeciderUtil.clientForAppId(

topLevelDdgMetricsMetadata.clientApplicationId),

"page" -> "cr-mixer",

"section" -> productName,

"component" -> label.\_1,

"element" -> label.\_2

)

}

}

object CrMixerScribeLogger {

val KafkaMaxTweetsPerMessage: Int = 200

val BatchSize: Int = 20

}