package com.twitter.recos.user\_tweet\_entity\_graph

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.finatra.kafka.consumers.FinagleKafkaConsumerBuilder

import com.twitter.graphjet.algorithms.{RecommendationType, TweetIDMask}

import com.twitter.graphjet.bipartite.NodeMetadataLeftIndexedMultiSegmentBipartiteGraph

import com.twitter.graphjet.bipartite.segment.NodeMetadataLeftIndexedBipartiteGraphSegment

import com.twitter.recos.hose.common.UnifiedGraphWriter

import com.twitter.recos.internal.thriftscala.RecosHoseMessage

import com.twitter.recos.serviceapi.Tweetypie.\_

/\*\*

\* The class submits a number of $numBootstrapWriters graph writer threads, BufferedEdgeWriter,

\* during service startup. One of them is live writer thread, and the other $(numBootstrapWriters - 1)

\* are catchup writer threads. All of them consume kafka events from an internal concurrent queue,

\* which is populated by kafka reader threads. At bootstrap time, the kafka reader threads look

\* back kafka offset from several hours ago and populate the internal concurrent queue.

\* Each graph writer thread writes to an individual graph segment separately.

\* The $(numBootstrapWriters - 1) catchup writer threads will stop once all events

\* between current system time at startup and the time in memcache are processed.

\* The live writer thread will continue to write all incoming kafka events.

\* It lives through the entire life cycle of recos graph service.

\*/

case class UserTweetEntityGraphWriter(

shardId: String,

env: String,

hosename: String,

bufferSize: Int,

kafkaConsumerBuilder: FinagleKafkaConsumerBuilder[String, RecosHoseMessage],

clientId: String,

statsReceiver: StatsReceiver)

extends UnifiedGraphWriter[

NodeMetadataLeftIndexedBipartiteGraphSegment,

NodeMetadataLeftIndexedMultiSegmentBipartiteGraph

] {

writer =>

// The max throughput for each kafka consumer is around 25MB/s

// Use 4 processors for 100MB/s catch-up speed.

val consumerNum: Int = 4

// Leave 1 Segments to LiveWriter

val catchupWriterNum: Int = RecosConfig.maxNumSegments - 1

private final val EMTPY\_LEFT\_NODE\_METADATA = new Array[Array[Int]](1)

/\*\*

\* Adds a RecosHoseMessage to the graph. used by live writer to insert edges to the

\* current segment

\*/

override def addEdgeToGraph(

graph: NodeMetadataLeftIndexedMultiSegmentBipartiteGraph,

recosHoseMessage: RecosHoseMessage

): Unit = {

graph.addEdge(

recosHoseMessage.leftId,

getMetaEdge(recosHoseMessage.rightId, recosHoseMessage.card),

UserTweetEdgeTypeMask.actionTypeToEdgeType(recosHoseMessage.action),

recosHoseMessage.edgeMetadata.getOrElse(0L),

EMTPY\_LEFT\_NODE\_METADATA,

extractEntities(recosHoseMessage)

)

}

/\*\*

\* Adds a RecosHoseMessage to the given segment in the graph. Used by catch up writers to

\* insert edges to non-current (old) segments

\*/

override def addEdgeToSegment(

segment: NodeMetadataLeftIndexedBipartiteGraphSegment,

recosHoseMessage: RecosHoseMessage

): Unit = {

segment.addEdge(

recosHoseMessage.leftId,

getMetaEdge(recosHoseMessage.rightId, recosHoseMessage.card),

UserTweetEdgeTypeMask.actionTypeToEdgeType(recosHoseMessage.action),

recosHoseMessage.edgeMetadata.getOrElse(0L),

EMTPY\_LEFT\_NODE\_METADATA,

extractEntities(recosHoseMessage)

)

}

private def getMetaEdge(rightId: Long, cardOption: Option[Byte]): Long = {

cardOption

.map { card =>

if (isPhotoCard(card)) TweetIDMask.photo(rightId)

else if (isPlayerCard(card)) TweetIDMask.player(rightId)

else if (isSummaryCard(card)) TweetIDMask.summary(rightId)

else if (isPromotionCard(card)) TweetIDMask.promotion(rightId)

else rightId

}

.getOrElse(rightId)

}

private def extractEntities(message: RecosHoseMessage): Array[Array[Int]] = {

val entities: Array[Array[Int]] =

new Array[Array[Int]](RecommendationType.METADATASIZE.getValue)

message.entities.foreach {

\_.foreach {

case (entityType, ids) =>

entities.update(entityType, ids.toArray)

}

}

entities

}

}