package src.scala.com.twitter.recos.hose.common

import com.twitter.finagle.stats.StatsReceiver

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

import com.twitter.graphjet.bipartite.LeftIndexedMultiSegmentBipartiteGraph

import com.twitter.graphjet.bipartite.segment.LeftIndexedBipartiteGraphSegment

import com.twitter.kafka.client.processor.AtLeastOnceProcessor

import com.twitter.kafka.client.processor.ThreadSafeKafkaConsumerClient

import com.twitter.logging.Logger

import com.twitter.recos.hose.common.BufferedEdgeCollector

import com.twitter.recos.hose.common.BufferedEdgeWriter

import com.twitter.recos.hose.common.EdgeCollector

import com.twitter.recos.hose.common.RecosEdgeProcessor

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

import com.twitter.recos.util.Action

import java.util.concurrent.atomic.AtomicBoolean

import java.util.concurrent.ConcurrentLinkedQueue

import java.util.concurrent.ExecutorService

import java.util.concurrent.Executors

import java.util.concurrent.Semaphore

/\*\*

\* The class is an variation of UnifiedGraphWriter which allow one instance to hold multiple graphs

\*/

trait UnifiedGraphWriterMulti[

TSegment <: LeftIndexedBipartiteGraphSegment,

TGraph <: LeftIndexedMultiSegmentBipartiteGraph[TSegment]] { writer =>

import UnifiedGraphWriterMulti.\_

def shardId: String

def env: String

def hosename: String

def bufferSize: Int

def consumerNum: Int

def catchupWriterNum: Int

def kafkaConsumerBuilder: FinagleKafkaConsumerBuilder[String, RecosHoseMessage]

def clientId: String

def statsReceiver: StatsReceiver

/\*\*

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

\* current segment

\*/

def addEdgeToGraph(

graphs: Seq[(TGraph, Set[Action.Value])],

recosHoseMessage: RecosHoseMessage

): Unit

/\*\*

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

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

\*/

def addEdgeToSegment(

segment: Seq[(TSegment, Set[Action.Value])],

recosHoseMessage: RecosHoseMessage

): Unit

private val log = Logger()

private val isRunning: AtomicBoolean = new AtomicBoolean(true)

private val initialized: AtomicBoolean = new AtomicBoolean(false)

private var processors: Seq[AtLeastOnceProcessor[String, RecosHoseMessage]] = Seq.empty

private var consumers: Seq[ThreadSafeKafkaConsumerClient[String, RecosHoseMessage]] = Seq.empty

private val threadPool: ExecutorService = Executors.newCachedThreadPool()

def shutdown(): Unit = {

processors.foreach { processor =>

processor.close()

}

processors = Seq.empty

consumers.foreach { consumer =>

consumer.close()

}

consumers = Seq.empty

threadPool.shutdown()

isRunning.set(false)

}

def initHose(liveGraphs: Seq[(TGraph, Set[Action.Value])]): Unit = this.synchronized {

if (!initialized.get) {

initialized.set(true)

val queue: java.util.Queue[Array[RecosHoseMessage]] =

new ConcurrentLinkedQueue[Array[RecosHoseMessage]]()

val queuelimit: Semaphore = new Semaphore(1024)

initRecosHoseKafka(queue, queuelimit)

initGrpahWriters(liveGraphs, queue, queuelimit)

} else {

throw new RuntimeException("attempt to re-init kafka hose")

}

}

private def initRecosHoseKafka(

queue: java.util.Queue[Array[RecosHoseMessage]],

queuelimit: Semaphore,

): Unit = {

try {

consumers = (0 until consumerNum).map { index =>

new ThreadSafeKafkaConsumerClient(

kafkaConsumerBuilder.clientId(s"clientId-$index").enableAutoCommit(false).config)

}

processors = consumers.zipWithIndex.map {

case (consumer, index) =>

val bufferedWriter = BufferedEdgeCollector(bufferSize, queue, queuelimit, statsReceiver)

val processor = RecosEdgeProcessor(bufferedWriter)(statsReceiver)

AtLeastOnceProcessor[String, RecosHoseMessage](

s"recos-injector-kafka-$index",

hosename,

consumer,

processor.process,

maxPendingRequests = MaxPendingRequests \* bufferSize,

workerThreads = ProcessorThreads,

commitIntervalMs = CommitIntervalMs,

statsReceiver = statsReceiver

)

}

log.info(s"starting ${processors.size} recosKafka processors")

processors.foreach { processor =>

processor.start()

}

} catch {

case e: Throwable =>

e.printStackTrace()

log.error(e, e.toString)

processors.foreach { processor =>

processor.close()

}

processors = Seq.empty

consumers.foreach { consumer =>

consumer.close()

}

consumers = Seq.empty

}

}

/\*\*

\* Initialize the graph writers,

\* by first creating catch up writers to bootstrap the older segments,

\* and then assigning a live writer to populate the live segment.

\*/

private def initGrpahWriters(

liveGraphs: Seq[(TGraph, Set[Action.Value])],

queue: java.util.Queue[Array[RecosHoseMessage]],

queuelimit: Semaphore

): Unit = {

// define a number of (numBootstrapWriters - 1) catchup writer threads, each of which will write

// to a separate graph segment.

val catchupWriters = (0 until (catchupWriterNum - 1)).map { index =>

val segments = liveGraphs.map { case (graph, actions) => (graph.getLiveSegment, actions) }

for (liveGraph <- liveGraphs) {

liveGraph.\_1.rollForwardSegment()

}

getCatchupWriter(segments, queue, queuelimit, index)

}

val threadPool: ExecutorService = Executors.newCachedThreadPool()

log.info("starting live graph writer that runs until service shutdown")

// define one live writer thread

val liveWriter = getLiveWriter(liveGraphs, queue, queuelimit)

threadPool.submit(liveWriter)

log.info(

"starting catchup graph writer, which will terminate as soon as the catchup segment is full"

)

catchupWriters.map(threadPool.submit(\_))

}

private def getLiveWriter(

liveGraphs: Seq[(TGraph, Set[Action.Value])],

queue: java.util.Queue[Array[RecosHoseMessage]],

queuelimit: Semaphore,

): BufferedEdgeWriter = {

val liveEdgeCollector = new EdgeCollector {

override def addEdge(message: RecosHoseMessage): Unit =

addEdgeToGraph(liveGraphs, message)

}

BufferedEdgeWriter(

queue,

queuelimit,

liveEdgeCollector,

statsReceiver.scope("liveWriter"),

isRunning.get

)

}

private def getCatchupWriter(

segments: Seq[(TSegment, Set[Action.Value])],

queue: java.util.Queue[Array[RecosHoseMessage]],

queuelimit: Semaphore,

catchupWriterIndex: Int,

): BufferedEdgeWriter = {

val catchupEdgeCollector = new EdgeCollector {

var currentNumEdges = 0

override def addEdge(message: RecosHoseMessage): Unit = {

currentNumEdges += 1

addEdgeToSegment(segments, message)

}

}

val maxEdges = segments.map(\_.\_1.getMaxNumEdges).sum

def runCondition(): Boolean = {

isRunning.get && ((maxEdges - catchupEdgeCollector.currentNumEdges) > bufferSize)

}

BufferedEdgeWriter(

queue,

queuelimit,

catchupEdgeCollector,

statsReceiver.scope("catcher\_" + catchupWriterIndex),

runCondition

)

}

}

private object UnifiedGraphWriterMulti {

// The RecosEdgeProcessor is not thread-safe. Only use one thread to process each instance.

val ProcessorThreads = 1

// Each one cache at most 1000 \* bufferSize requests.

val MaxPendingRequests = 1000

// Short Commit MS to reduce duplicate messages.

val CommitIntervalMs: Long = 5000 // 5 seconds, Default Kafka value.

}