package com.twitter.timelines.data\_processing.ml\_util.aggregation\_framework.heron

import com.twitter.algebird.Monoid

import com.twitter.bijection.Injection

import com.twitter.bijection.thrift.CompactThriftCodec

import com.twitter.conversions.DurationOps.\_

import com.twitter.finagle.mtls.authentication.EmptyServiceIdentifier

import com.twitter.finagle.mtls.authentication.ServiceIdentifier

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.heron.util.CommonMetric

import com.twitter.ml.api.DataRecord

import com.twitter.scalding.Args

import com.twitter.storehaus.algebra.MergeableStore

import com.twitter.storehaus.algebra.StoreAlgebra.\_

import com.twitter.storehaus\_internal.memcache.Memcache

import com.twitter.storehaus\_internal.store.CombinedStore

import com.twitter.storehaus\_internal.store.ReplicatingWritableStore

import com.twitter.summingbird.batch.BatchID

import com.twitter.summingbird.batch.Batcher

import com.twitter.summingbird.online.MergeableStoreFactory

import com.twitter.summingbird.online.option.\_

import com.twitter.summingbird.option.CacheSize

import com.twitter.summingbird.option.JobId

import com.twitter.summingbird.storm.option.FlatMapStormMetrics

import com.twitter.summingbird.storm.option.SummerStormMetrics

import com.twitter.summingbird.storm.Storm

import com.twitter.summingbird.storm.StormMetric

import com.twitter.summingbird.Options

import com.twitter.summingbird.\_

import com.twitter.summingbird\_internal.runner.common.CapTicket

import com.twitter.summingbird\_internal.runner.common.JobName

import com.twitter.summingbird\_internal.runner.common.TeamEmail

import com.twitter.summingbird\_internal.runner.common.TeamName

import com.twitter.summingbird\_internal.runner.storm.ProductionStormConfig

import com.twitter.timelines.data\_processing.ml\_util.aggregation\_framework.\_

import com.twitter.timelines.data\_processing.ml\_util.aggregation\_framework.job.AggregatesV2Job

import com.twitter.timelines.data\_processing.ml\_util.aggregation\_framework.job.AggregatesV2Job

import com.twitter.timelines.data\_processing.ml\_util.aggregation\_framework.job.DataRecordFeatureCounter

import org.apache.heron.api.{Config => HeronConfig}

import org.apache.heron.common.basics.ByteAmount

import org.apache.storm.Config

import scala.collection.JavaConverters.\_

object RealTimeAggregatesJobBase {

lazy val commonMetric: StormMetric[CommonMetric] =

StormMetric(new CommonMetric(), CommonMetric.NAME, CommonMetric.POLL\_INTERVAL)

lazy val flatMapMetrics: FlatMapStormMetrics = FlatMapStormMetrics(Iterable(commonMetric))

lazy val summerMetrics: SummerStormMetrics = SummerStormMetrics(Iterable(commonMetric))

}

trait RealTimeAggregatesJobBase extends Serializable {

import RealTimeAggregatesJobBase.\_

import com.twitter.summingbird\_internal.bijection.BatchPairImplicits.\_

def statsReceiver: StatsReceiver

def aggregatesToCompute: Set[TypedAggregateGroup[\_]]

def jobConfigs: RealTimeAggregatesJobConfigs

implicit lazy val dataRecordCodec: Injection[DataRecord, Array[Byte]] =

CompactThriftCodec[DataRecord]

implicit lazy val monoid: Monoid[DataRecord] = DataRecordAggregationMonoid(aggregatesToCompute)

implicit lazy val aggregationKeyInjection: Injection[AggregationKey, Array[Byte]] =

AggregationKeyInjection

val clusters: Set[String] = Set("atla", "pdxa")

def buildAggregateStoreToStorm(

isProd: Boolean,

serviceIdentifier: ServiceIdentifier,

jobConfig: RealTimeAggregatesJobConfig

): (AggregateStore => Option[Storm#Store[AggregationKey, DataRecord]]) = {

(store: AggregateStore) =>

store match {

case rtaStore: RealTimeAggregateStore if rtaStore.isProd == isProd => {

lazy val primaryStore: MergeableStore[(AggregationKey, BatchID), DataRecord] =

Memcache.getMemcacheStore[(AggregationKey, BatchID), DataRecord](

rtaStore.online(serviceIdentifier))

lazy val mergeableStore: MergeableStore[(AggregationKey, BatchID), DataRecord] =

if (jobConfig.enableUserReindexingNighthawkBtreeStore

|| jobConfig.enableUserReindexingNighthawkHashStore) {

val reindexingNighthawkBtreeWritableDataRecordStoreList =

if (jobConfig.enableUserReindexingNighthawkBtreeStore) {

lazy val cacheClientNighthawkConfig =

jobConfig.userReindexingNighthawkBtreeStoreConfig.online(serviceIdentifier)

List(

UserReindexingNighthawkWritableDataRecordStore.getBtreeStore(

nighthawkCacheConfig = cacheClientNighthawkConfig,

// Choose a reasonably large target size as this will be equivalent to the number of unique (user, timestamp)

// keys that are returned on read on the pKey, and we may have duplicate authors and associated records.

targetSize = 512,

statsReceiver = statsReceiver,

// Assuming trims are relatively expensive, choose a trimRate that's not as aggressive. In this case we trim on

// 10% of all writes.

trimRate = 0.1

))

} else { Nil }

val reindexingNighthawkHashWritableDataRecordStoreList =

if (jobConfig.enableUserReindexingNighthawkHashStore) {

lazy val cacheClientNighthawkConfig =

jobConfig.userReindexingNighthawkHashStoreConfig.online(serviceIdentifier)

List(

UserReindexingNighthawkWritableDataRecordStore.getHashStore(

nighthawkCacheConfig = cacheClientNighthawkConfig,

// Choose a reasonably large target size as this will be equivalent to the number of unique (user, timestamp)

// keys that are returned on read on the pKey, and we may have duplicate authors and associated records.

targetSize = 512,

statsReceiver = statsReceiver,

// Assuming trims are relatively expensive, choose a trimRate that's not as aggressive. In this case we trim on

// 10% of all writes.

trimRate = 0.1

))

} else { Nil }

lazy val replicatingWritableStore = new ReplicatingWritableStore(

stores = List(primaryStore) ++ reindexingNighthawkBtreeWritableDataRecordStoreList

++ reindexingNighthawkHashWritableDataRecordStoreList

)

lazy val combinedStoreWithReindexing = new CombinedStore(

read = primaryStore,

write = replicatingWritableStore

)

combinedStoreWithReindexing.toMergeable

} else {

primaryStore

}

lazy val storeFactory: MergeableStoreFactory[(AggregationKey, BatchID), DataRecord] =

Storm.store(mergeableStore)(Batcher.unit)

Some(storeFactory)

}

case \_ => None

}

}

def buildDataRecordSourceToStorm(

jobConfig: RealTimeAggregatesJobConfig

): (AggregateSource => Option[Producer[Storm, DataRecord]]) = { (source: AggregateSource) =>

{

source match {

case stormAggregateSource: StormAggregateSource =>

Some(stormAggregateSource.build(statsReceiver, jobConfig))

case \_ => None

}

}

}

def apply(args: Args): ProductionStormConfig = {

lazy val isProd = args.boolean("production")

lazy val cluster = args.getOrElse("cluster", "")

lazy val isDebug = args.boolean("debug")

lazy val role = args.getOrElse("role", "")

lazy val service =

args.getOrElse(

"service\_name",

""

) // don't use the argument service, which is a reserved heron argument

lazy val environment = if (isProd) "prod" else "devel"

lazy val s2sEnabled = args.boolean("s2s")

lazy val keyedByUserEnabled = args.boolean("keyed\_by\_user")

lazy val keyedByAuthorEnabled = args.boolean("keyed\_by\_author")

require(clusters.contains(cluster))

if (s2sEnabled) {

require(role.length() > 0)

require(service.length() > 0)

}

lazy val serviceIdentifier = if (s2sEnabled) {

ServiceIdentifier(

role = role,

service = service,

environment = environment,

zone = cluster

)

} else EmptyServiceIdentifier

lazy val jobConfig = {

val jobConfig = if (isProd) jobConfigs.Prod else jobConfigs.Devel

jobConfig.copy(

serviceIdentifier = serviceIdentifier,

keyedByUserEnabled = keyedByUserEnabled,

keyedByAuthorEnabled = keyedByAuthorEnabled)

}

lazy val dataRecordSourceToStorm = buildDataRecordSourceToStorm(jobConfig)

lazy val aggregateStoreToStorm =

buildAggregateStoreToStorm(isProd, serviceIdentifier, jobConfig)

lazy val JaasConfigFlag = "-Djava.security.auth.login.config=resources/jaas.conf"

lazy val JaasDebugFlag = "-Dsun.security.krb5.debug=true"

lazy val JaasConfigString =

if (isDebug) { "%s %s".format(JaasConfigFlag, JaasDebugFlag) }

else JaasConfigFlag

new ProductionStormConfig {

implicit val jobId: JobId = JobId(jobConfig.name)

override val jobName = JobName(jobConfig.name)

override val teamName = TeamName(jobConfig.teamName)

override val teamEmail = TeamEmail(jobConfig.teamEmail)

override val capTicket = CapTicket("n/a")

val configureHeronJvmSettings = {

val heronJvmOptions = new java.util.HashMap[String, AnyRef]()

jobConfig.componentToRamGigaBytesMap.foreach {

case (component, gigabytes) =>

HeronConfig.setComponentRam(

heronJvmOptions,

component,

ByteAmount.fromGigabytes(gigabytes))

}

HeronConfig.setContainerRamRequested(

heronJvmOptions,

ByteAmount.fromGigabytes(jobConfig.containerRamGigaBytes)

)

jobConfig.componentsToKerberize.foreach { component =>

HeronConfig.setComponentJvmOptions(

heronJvmOptions,

component,

JaasConfigString

)

}

jobConfig.componentToMetaSpaceSizeMap.foreach {

case (component, metaspaceSize) =>

HeronConfig.setComponentJvmOptions(

heronJvmOptions,

component,

metaspaceSize

)

}

heronJvmOptions.asScala.toMap ++ AggregatesV2Job

.aggregateNames(aggregatesToCompute).map {

case (prefix, aggNames) => (s"extras.aggregateNames.${prefix}", aggNames)

}

}

override def transformConfig(m: Map[String, AnyRef]): Map[String, AnyRef] = {

super.transformConfig(m) ++ List(

/\*\*

\* Disable acking by setting acker executors to 0. Tuples that come off the

\* spout will be immediately acked which effectively disables retries on tuple

\* failures. This should help topology throughput/availability by relaxing consistency.

\*/

Config.TOPOLOGY\_ACKER\_EXECUTORS -> int2Integer(0),

Config.TOPOLOGY\_WORKERS -> int2Integer(jobConfig.topologyWorkers),

HeronConfig.TOPOLOGY\_CONTAINER\_CPU\_REQUESTED -> int2Integer(8),

HeronConfig.TOPOLOGY\_DROPTUPLES\_UPON\_BACKPRESSURE -> java.lang.Boolean.valueOf(true),

HeronConfig.TOPOLOGY\_WORKER\_CHILDOPTS -> List(

JaasConfigString,

s"-Dcom.twitter.eventbus.client.zoneName=${cluster}",

"-Dcom.twitter.eventbus.client.EnableKafkaSaslTls=true"

).mkString(" "),

"storm.job.uniqueId" -> jobId.get

) ++ configureHeronJvmSettings

}

override lazy val getNamedOptions: Map[String, Options] = jobConfig.topologyNamedOptions ++

Map(

"DEFAULT" -> Options()

.set(flatMapMetrics)

.set(summerMetrics)

.set(MaxWaitingFutures(1000))

.set(FlushFrequency(30.seconds))

.set(UseAsyncCache(true))

.set(AsyncPoolSize(4))

.set(SourceParallelism(jobConfig.sourceCount))

.set(SummerBatchMultiplier(1000)),

"FLATMAP" -> Options()

.set(FlatMapParallelism(jobConfig.flatMapCount))

.set(CacheSize(0)),

"SUMMER" -> Options()

.set(SummerParallelism(jobConfig.summerCount))

/\*\*

\* Sets number of tuples a Summer awaits before aggregation. Set higher

\* if you need to lower qps to memcache at the expense of introducing

\* some (stable) latency.

\*/

.set(CacheSize(jobConfig.cacheSize))

)

val featureCounters: Seq[DataRecordFeatureCounter] =

Seq(DataRecordFeatureCounter.any(Counter(Group("feature\_counter"), Name("num\_records"))))

override def graph: TailProducer[Storm, Any] = AggregatesV2Job.generateJobGraph[Storm](

aggregateSet = aggregatesToCompute,

aggregateSourceToSummingbird = dataRecordSourceToStorm,

aggregateStoreToSummingbird = aggregateStoreToStorm,

featureCounters = featureCounters

)

}

}

}