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

import com.twitter.bijection.Injection

import com.twitter.bijection.thrift.CompactThriftCodec

import com.twitter.cache.client.\_

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.ml.api.DataRecord

import com.twitter.ml.api.constant.SharedFeatures

import com.twitter.ml.api.util.SRichDataRecord

import com.twitter.storehaus.WritableStore

import com.twitter.storehaus\_internal.nighthawk\_kv.CacheClientNighthawkConfig

import com.twitter.storehaus\_internal.nighthawk\_kv.NighthawkStore

import com.twitter.summingbird.batch.BatchID

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

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

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

import com.twitter.timelines.prediction.features.common.TimelinesSharedFeatures

import com.twitter.util.Future

import com.twitter.util.Time

import com.twitter.util.Try

import com.twitter.util.logging.Logger

import java.nio.ByteBuffer

import java.util

import scala.util.Random

object UserReindexingNighthawkWritableDataRecordStore {

implicit val longInjection = Injection.long2BigEndian

implicit val dataRecordInjection: Injection[DataRecord, Array[Byte]] =

CompactThriftCodec[DataRecord]

val arrayToByteBuffer = Injection.connect[Array[Byte], ByteBuffer]

val longToByteBuffer = longInjection.andThen(arrayToByteBuffer)

val dataRecordToByteBuffer = dataRecordInjection.andThen(arrayToByteBuffer)

def getBtreeStore(

nighthawkCacheConfig: CacheClientNighthawkConfig,

targetSize: Int,

statsReceiver: StatsReceiver,

trimRate: Double

): UserReindexingNighthawkBtreeWritableDataRecordStore =

new UserReindexingNighthawkBtreeWritableDataRecordStore(

nighthawkStore = NighthawkStore[UserId, TimestampMs, DataRecord](nighthawkCacheConfig)

.asInstanceOf[NighthawkStore[UserId, TimestampMs, DataRecord]],

tableName = nighthawkCacheConfig.table.toString,

targetSize = targetSize,

statsReceiver = statsReceiver,

trimRate = trimRate

)

def getHashStore(

nighthawkCacheConfig: CacheClientNighthawkConfig,

targetSize: Int,

statsReceiver: StatsReceiver,

trimRate: Double

): UserReindexingNighthawkHashWritableDataRecordStore =

new UserReindexingNighthawkHashWritableDataRecordStore(

nighthawkStore = NighthawkStore[UserId, AuthorId, DataRecord](nighthawkCacheConfig)

.asInstanceOf[NighthawkStore[UserId, AuthorId, DataRecord]],

tableName = nighthawkCacheConfig.table.toString,

targetSize = targetSize,

statsReceiver = statsReceiver,

trimRate = trimRate

)

def buildTimestampedByteBuffer(timestamp: Long, bb: ByteBuffer): ByteBuffer = {

val timestampedBb = ByteBuffer.allocate(getLength(bb) + java.lang.Long.SIZE)

timestampedBb.putLong(timestamp)

timestampedBb.put(bb)

timestampedBb

}

def extractTimestampFromTimestampedByteBuffer(bb: ByteBuffer): Long = {

bb.getLong(0)

}

def extractValueFromTimestampedByteBuffer(bb: ByteBuffer): ByteBuffer = {

val bytes = new Array[Byte](getLength(bb) - java.lang.Long.SIZE)

util.Arrays.copyOfRange(bytes, java.lang.Long.SIZE, getLength(bb))

ByteBuffer.wrap(bytes)

}

def transformAndBuildKeyValueMapping(

table: String,

userId: UserId,

authorIdsAndDataRecords: Seq[(AuthorId, DataRecord)]

): KeyValue = {

val timestamp = Time.now.inMillis

val pkey = longToByteBuffer(userId)

val lkeysAndTimestampedValues = authorIdsAndDataRecords.map {

case (authorId, dataRecord) =>

val lkey = longToByteBuffer(authorId)

// Create a byte buffer with a prepended timestamp to reduce deserialization cost

// when parsing values. We only have to extract and deserialize the timestamp in the

// ByteBuffer in order to sort the value, as opposed to deserializing the DataRecord

// and having to get a timestamp feature value from the DataRecord.

val dataRecordBb = dataRecordToByteBuffer(dataRecord)

val timestampedValue = buildTimestampedByteBuffer(timestamp, dataRecordBb)

(lkey, timestampedValue)

}

buildKeyValueMapping(table, pkey, lkeysAndTimestampedValues)

}

def buildKeyValueMapping(

table: String,

pkey: ByteBuffer,

lkeysAndTimestampedValues: Seq[(ByteBuffer, ByteBuffer)]

): KeyValue = {

val lkeys = lkeysAndTimestampedValues.map { case (lkey, \_) => lkey }

val timestampedValues = lkeysAndTimestampedValues.map { case (\_, value) => value }

val kv = KeyValue(

key = Key(table = table, pkey = pkey, lkeys = lkeys),

value = Value(timestampedValues)

)

kv

}

private def getLength(bb: ByteBuffer): Int = {

// capacity can be an over-estimate of the actual length (remaining - start position)

// but it's the safest to avoid overflows.

bb.capacity()

}

}

/\*\*

\* Implements a NH store that stores aggregate feature DataRecords using userId as the primary key.

\*

\* This store re-indexes user-author keyed real-time aggregate (RTA) features on userId by

\* writing to a userId primary key (pkey) and timestamp secondary key (lkey). To fetch user-author

\* RTAs for a given user from cache, the caller just needs to make a single RPC for the userId pkey.

\* The downside of a re-indexing store is that we cannot store arbitrarily many secondary keys

\* under the primary key. This specific implementation using the NH btree backend also mandates

\* mandates an ordering of secondary keys - we therefore use timestamp as the secondary key

\* as opposed to say authorId.

\*

\* Note that a caller of the btree backed NH re-indexing store receives back a response where the

\* secondary key is a timestamp. The associated value is a DataRecord containing user-author related

\* aggregate features which was last updated at the timestamp. The caller therefore needs to handle

\* the response and dedupe on unique, most recent user-author pairs.

\*

\* For a discussion on this and other implementations, please see:

\* https://docs.google.com/document/d/1yVzAbQ\_ikLqwSf230URxCJmSKj5yZr5dYv6TwBlQw18/edit

\*/

class UserReindexingNighthawkBtreeWritableDataRecordStore(

nighthawkStore: NighthawkStore[UserId, TimestampMs, DataRecord],

tableName: String,

targetSize: Int,

statsReceiver: StatsReceiver,

trimRate: Double = 0.1 // by default, trim on 10% of puts

) extends WritableStore[(AggregationKey, BatchID), Option[DataRecord]] {

private val scope = getClass.getSimpleName

private val failures = statsReceiver.counter(scope, "failures")

private val log = Logger.getLogger(getClass)

private val random: Random = new Random(1729L)

override def put(kv: ((AggregationKey, BatchID), Option[DataRecord])): Future[Unit] = {

val ((aggregationKey, \_), dataRecordOpt) = kv

// Fire-and-forget below because the store itself should just be a side effect

// as it's just making re-indexed writes based on the writes to the primary store.

for {

userId <- aggregationKey.discreteFeaturesById.get(SharedFeatures.USER\_ID.getFeatureId)

dataRecord <- dataRecordOpt

} yield {

SRichDataRecord(dataRecord)

.getFeatureValueOpt(TypedAggregateGroup.timestampFeature)

.map(\_.toLong) // convert to Scala Long

.map { timestamp =>

val trim: Future[Unit] = if (random.nextDouble <= trimRate) {

val trimKey = TrimKey(

table = tableName,

pkey = longToByteBuffer(userId),

targetSize = targetSize,

ascending = true

)

nighthawkStore.client.trim(Seq(trimKey)).unit

} else {

Future.Unit

}

// We should wait for trim to complete above

val fireAndForget = trim.before {

val kvTuple = ((userId, timestamp), Some(dataRecord))

nighthawkStore.put(kvTuple)

}

fireAndForget.onFailure {

case e =>

failures.incr()

log.error("Failure in UserReindexingNighthawkHashWritableDataRecordStore", e)

}

}

}

// Ignore fire-and-forget result above and simply return

Future.Unit

}

}

/\*\*

\* Implements a NH store that stores aggregate feature DataRecords using userId as the primary key.

\*

\* This store re-indexes user-author keyed real-time aggregate (RTA) features on userId by

\* writing to a userId primary key (pkey) and authorId secondary key (lkey). To fetch user-author

\* RTAs for a given user from cache, the caller just needs to make a single RPC for the userId pkey.

\* The downside of a re-indexing store is that we cannot store arbitrarily

\* many secondary keys under the primary key. We have to limit them in some way;

\* here, we do so by randomly (based on trimRate) issuing an HGETALL command (via scan) to

\* retrieve the whole hash, sort by oldest timestamp, and then remove the oldest authors to keep

\* only targetSize authors (aka trim), where targetSize is configurable.

\*

\* @note The full hash returned from scan could be as large (or even larger) than targetSize,

\* which could mean many DataRecords to deserialize, especially at high write qps.

\* To reduce deserialization cost post-scan, we use timestamped values with a prepended timestamp

\* in the value ByteBuffer; this allows us to only deserialize the timestamp and not the full

\* DataRecord when sorting. This is necessary in order to identify the oldest values to trim.

\* When we do a put for a new (user, author) pair, we also write out timestamped values.

\*

\* For a discussion on this and other implementations, please see:

\* https://docs.google.com/document/d/1yVzAbQ\_ikLqwSf230URxCJmSKj5yZr5dYv6TwBlQw18/edit

\*/

class UserReindexingNighthawkHashWritableDataRecordStore(

nighthawkStore: NighthawkStore[UserId, AuthorId, DataRecord],

tableName: String,

targetSize: Int,

statsReceiver: StatsReceiver,

trimRate: Double = 0.1 // by default, trim on 10% of puts

) extends WritableStore[(AggregationKey, BatchID), Option[DataRecord]] {

private val scope = getClass.getSimpleName

private val scanMismatchErrors = statsReceiver.counter(scope, "scanMismatchErrors")

private val failures = statsReceiver.counter(scope, "failures")

private val log = Logger.getLogger(getClass)

private val random: Random = new Random(1729L)

private val arrayToByteBuffer = Injection.connect[Array[Byte], ByteBuffer]

private val longToByteBuffer = Injection.long2BigEndian.andThen(arrayToByteBuffer)

override def put(kv: ((AggregationKey, BatchID), Option[DataRecord])): Future[Unit] = {

val ((aggregationKey, \_), dataRecordOpt) = kv

// Fire-and-forget below because the store itself should just be a side effect

// as it's just making re-indexed writes based on the writes to the primary store.

for {

userId <- aggregationKey.discreteFeaturesById.get(SharedFeatures.USER\_ID.getFeatureId)

authorId <- aggregationKey.discreteFeaturesById.get(

TimelinesSharedFeatures.SOURCE\_AUTHOR\_ID.getFeatureId)

dataRecord <- dataRecordOpt

} yield {

val scanAndTrim: Future[Unit] = if (random.nextDouble <= trimRate) {

val scanKey = ScanKey(

table = tableName,

pkey = longToByteBuffer(userId)

)

nighthawkStore.client.scan(Seq(scanKey)).flatMap { scanResults: Seq[Try[KeyValue]] =>

scanResults.headOption

.flatMap(\_.toOption).map { keyValue: KeyValue =>

val lkeys: Seq[ByteBuffer] = keyValue.key.lkeys

// these are timestamped bytebuffers

val timestampedValues: Seq[ByteBuffer] = keyValue.value.values

// this should fail loudly if this is not true. it would indicate

// there is a mistake in the scan.

if (lkeys.size != timestampedValues.size) scanMismatchErrors.incr()

assert(lkeys.size == timestampedValues.size)

if (lkeys.size > targetSize) {

val numToRemove = targetSize - lkeys.size

// sort by oldest and take top k oldest and remove - this is equivalent to a trim

val oldestKeys: Seq[ByteBuffer] = lkeys

.zip(timestampedValues)

.map {

case (lkey, timestampedValue) =>

val timestamp = extractTimestampFromTimestampedByteBuffer(timestampedValue)

(timestamp, lkey)

}

.sortBy { case (timestamp, \_) => timestamp }

.take(numToRemove)

.map { case (\_, k) => k }

val pkey = longToByteBuffer(userId)

val key = Key(table = tableName, pkey = pkey, lkeys = oldestKeys)

// NOTE: `remove` is a batch API, and we group all lkeys into a single batch (batch

// size = single group of lkeys = 1). Instead, we could separate lkeys into smaller

// groups and have batch size = number of groups, but this is more complex.

// Performance implications of batching vs non-batching need to be assessed.

nighthawkStore.client

.remove(Seq(key))

.map { responses =>

responses.map(resp => nighthawkStore.processValue(resp))

}.unit

} else {

Future.Unit

}

}.getOrElse(Future.Unit)

}

} else {

Future.Unit

}

// We should wait for scan and trim to complete above

val fireAndForget = scanAndTrim.before {

val kv = transformAndBuildKeyValueMapping(tableName, userId, Seq((authorId, dataRecord)))

nighthawkStore.client

.put(Seq(kv))

.map { responses =>

responses.map(resp => nighthawkStore.processValue(resp))

}.unit

}

fireAndForget.onFailure {

case e =>

failures.incr()

log.error("Failure in UserReindexingNighthawkHashWritableDataRecordStore", e)

}

}

// Ignore fire-and-forget result above and simply return

Future.Unit

}

}