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

import com.twitter.bijection.Injection

import com.twitter.scalding.commons.source.VersionedKeyValSource

import com.twitter.scalding.TypedPipe

import com.twitter.scalding.{Hdfs => HdfsMode}

import com.twitter.summingbird.batch.store.HDFSMetadata

import com.twitter.summingbird.batch.BatchID

import com.twitter.summingbird.batch.Batcher

import com.twitter.summingbird.batch.OrderedFromOrderingExt

import com.twitter.summingbird.batch.PrunedSpace

import com.twitter.summingbird.scalding.\_

import com.twitter.summingbird.scalding.store.VersionedBatchStore

import org.slf4j.LoggerFactory

object MostRecentLagCorrectingVersionedStore {

def apply[Key, ValInStore, ValInMemory](

rootPath: String,

packer: ValInMemory => ValInStore,

unpacker: ValInStore => ValInMemory,

versionsToKeep: Int = VersionedKeyValSource.defaultVersionsToKeep,

prunedSpace: PrunedSpace[(Key, ValInMemory)] = PrunedSpace.neverPruned

)(

implicit injection: Injection[(Key, (BatchID, ValInStore)), (Array[Byte], Array[Byte])],

batcher: Batcher,

ord: Ordering[Key],

lagCorrector: (ValInMemory, Long) => ValInMemory

): MostRecentLagCorrectingVersionedBatchStore[Key, ValInMemory, Key, (BatchID, ValInStore)] = {

new MostRecentLagCorrectingVersionedBatchStore[Key, ValInMemory, Key, (BatchID, ValInStore)](

rootPath,

versionsToKeep,

batcher

)(lagCorrector)({ case (batchID, (k, v)) => (k, (batchID.next, packer(v))) })({

case (k, (\_, v)) => (k, unpacker(v))

}) {

override def select(b: List[BatchID]) = List(b.last)

override def pruning: PrunedSpace[(Key, ValInMemory)] = prunedSpace

}

}

}

/\*\*

\* @param lagCorrector lagCorrector allows one to take data from one batch and pretend as if it

\* came from a different batch.

\* @param pack Converts the in-memory tuples to the type used by the underlying key-val store.

\* @param unpack Converts the key-val tuples from the store in the form used by the calling object.

\*/

class MostRecentLagCorrectingVersionedBatchStore[KeyInMemory, ValInMemory, KeyInStore, ValInStore](

rootPath: String,

versionsToKeep: Int,

override val batcher: Batcher

)(

lagCorrector: (ValInMemory, Long) => ValInMemory

)(

pack: (BatchID, (KeyInMemory, ValInMemory)) => (KeyInStore, ValInStore)

)(

unpack: ((KeyInStore, ValInStore)) => (KeyInMemory, ValInMemory)

)(

implicit @transient injection: Injection[(KeyInStore, ValInStore), (Array[Byte], Array[Byte])],

override val ordering: Ordering[KeyInMemory])

extends VersionedBatchStore[KeyInMemory, ValInMemory, KeyInStore, ValInStore](

rootPath,

versionsToKeep,

batcher)(pack)(unpack)(injection, ordering) {

import OrderedFromOrderingExt.\_

@transient private val logger =

LoggerFactory.getLogger(classOf[MostRecentLagCorrectingVersionedBatchStore[\_, \_, \_, \_]])

override protected def lastBatch(

exclusiveUB: BatchID,

mode: HdfsMode

): Option[(BatchID, FlowProducer[TypedPipe[(KeyInMemory, ValInMemory)]])] = {

val batchToPretendAs = exclusiveUB.prev

val versionToPretendAs = batchIDToVersion(batchToPretendAs)

logger.info(

s"Most recent lag correcting versioned batched store at $rootPath entering lastBatch method versionToPretendAs = $versionToPretendAs")

val meta = new HDFSMetadata(mode.conf, rootPath)

meta.versions

.map { ver => (versionToBatchID(ver), readVersion(ver)) }

.filter { \_.\_1 < exclusiveUB }

.reduceOption { (a, b) => if (a.\_1 > b.\_1) a else b }

.map {

case (

lastBatchID: BatchID,

flowProducer: FlowProducer[TypedPipe[(KeyInMemory, ValInMemory)]]) =>

val lastVersion = batchIDToVersion(lastBatchID)

val lagToCorrectMillis: Long =

batchIDToVersion(batchToPretendAs) - batchIDToVersion(lastBatchID)

logger.info(

s"Most recent available version is $lastVersion, so lagToCorrectMillis is $lagToCorrectMillis")

val lagCorrectedFlowProducer = flowProducer.map {

pipe: TypedPipe[(KeyInMemory, ValInMemory)] =>

pipe.map { case (k, v) => (k, lagCorrector(v, lagToCorrectMillis)) }

}

(batchToPretendAs, lagCorrectedFlowProducer)

}

}

}