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

import com.twitter.bijection.thrift.CompactThriftCodec

import com.twitter.bijection.Codec

import com.twitter.bijection.Injection

import com.twitter.ml.api.\_

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

import com.twitter.ml.api.util.CompactDataRecordConverter

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

import com.twitter.scalding.Args

import com.twitter.scalding\_internal.dalv2.DALWrite.D

import com.twitter.storehaus\_internal.manhattan.ManhattanROConfig

import com.twitter.summingbird.batch.option.Reducers

import com.twitter.summingbird.batch.BatchID

import com.twitter.summingbird.batch.Batcher

import com.twitter.summingbird.batch.Timestamp

import com.twitter.summingbird.option.\_

import com.twitter.summingbird.scalding.Scalding

import com.twitter.summingbird.scalding.batch.{BatchedStore => ScaldingBatchedStore}

import com.twitter.summingbird.Options

import com.twitter.summingbird.Producer

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

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

import com.twitter.summingbird\_internal.runner.scalding.GenericRunner

import com.twitter.summingbird\_internal.runner.scalding.ScaldingConfig

import com.twitter.summingbird\_internal.runner.scalding.StatebirdState

import com.twitter.summingbird\_internal.dalv2.DAL

import com.twitter.summingbird\_internal.runner.store\_config.\_

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

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

import job.AggregatesV2Job

import org.apache.hadoop.conf.Configuration

/\*

\* Offline scalding version of summingbird job to compute aggregates v2.

\* This is loosely based on the template created by sb-gen.

\* Extend this trait in your own scalding job, and override the val

\* "aggregatesToCompute" with your own desired set of aggregates.

\*/

trait AggregatesV2ScaldingJob {

val aggregatesToCompute: Set[TypedAggregateGroup[\_]]

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

AggregationKeyInjection

implicit val aggregationKeyOrdering: AggregationKeyOrdering.type = AggregationKeyOrdering

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

private implicit val compactDataRecordCodec: Injection[CompactDataRecord, Array[Byte]] =

CompactThriftCodec[CompactDataRecord]

private val compactDataRecordConverter = new CompactDataRecordConverter()

def numReducers: Int = -1

/\*\*

\* Function that maps from a logical ''AggregateSource''

\* to an underlying physical source. The physical source

\* for the scalding platform is a ScaldingAggregateSource.

\*/

def dataRecordSourceToScalding(

source: AggregateSource

): Option[Producer[Scalding, DataRecord]] = {

source match {

case offlineSource: OfflineAggregateSource =>

Some(ScaldingAggregateSource(offlineSource).source)

case \_ => None

}

}

/\*\*

\* Creates and returns a versioned store using the config parameters

\* with a specific number of versions to keep, and which can read from

\* the most recent available version on HDFS rather than a specific

\* version number. The store applies a timestamp correction based on the

\* number of days of aggregate data skipped over at read time to ensure

\* that skipping data plays nicely with halfLife decay.

\*

\* @param config specifying the Manhattan store parameters

\* @param versionsToKeep number of old versions to keep

\*/

def getMostRecentLagCorrectingVersionedStoreWithRetention[

Key: Codec: Ordering,

ValInStore: Codec,

ValInMemory

](

config: OfflineStoreOnlyConfig[ManhattanROConfig],

versionsToKeep: Int,

lagCorrector: (ValInMemory, Long) => ValInMemory,

packer: ValInMemory => ValInStore,

unpacker: ValInStore => ValInMemory

): ScaldingBatchedStore[Key, ValInMemory] = {

MostRecentLagCorrectingVersionedStore[Key, ValInStore, ValInMemory](

config.offline.hdfsPath.toString,

packer = packer,

unpacker = unpacker,

versionsToKeep = versionsToKeep)(

Injection.connect[(Key, (BatchID, ValInStore)), (Array[Byte], Array[Byte])],

config.batcher,

implicitly[Ordering[Key]],

lagCorrector

).withInitialBatch(config.batcher.batchOf(config.startTime.value))

}

def mutablyCorrectDataRecordTimestamp(

record: DataRecord,

lagToCorrectMillis: Long

): DataRecord = {

val richRecord = SRichDataRecord(record)

if (richRecord.hasFeature(TIMESTAMP)) {

val timestamp = richRecord.getFeatureValue(TIMESTAMP).toLong

richRecord.setFeatureValue(TIMESTAMP, timestamp + lagToCorrectMillis)

}

record

}

/\*\*

\* Function that maps from a logical ''AggregateStore''

\* to an underlying physical store. The physical store for

\* scalding is a HDFS VersionedKeyValSource dataset.

\*/

def aggregateStoreToScalding(

store: AggregateStore

): Option[Scalding#Store[AggregationKey, DataRecord]] = {

store match {

case offlineStore: OfflineAggregateDataRecordStore =>

Some(

getMostRecentLagCorrectingVersionedStoreWithRetention[

AggregationKey,

DataRecord,

DataRecord](

offlineStore,

versionsToKeep = offlineStore.batchesToKeep,

lagCorrector = mutablyCorrectDataRecordTimestamp,

packer = Injection.identity[DataRecord],

unpacker = Injection.identity[DataRecord]

)

)

case offlineStore: OfflineAggregateDataRecordStoreWithDAL =>

Some(

DAL.versionedKeyValStore[AggregationKey, DataRecord](

dataset = offlineStore.dalDataset,

pathLayout = D.Suffix(offlineStore.offline.hdfsPath.toString),

batcher = offlineStore.batcher,

maybeStartTime = Some(offlineStore.startTime),

maxErrors = offlineStore.maxKvSourceFailures

))

case \_ => None

}

}

def generate(args: Args): ScaldingConfig = new ScaldingConfig {

val jobName = JobName(args("job\_name"))

/\*

\* Add registrars for chill serialization for user-defined types.

\* We use the default: an empty List().

\*/

override def registrars = List()

/\* Use transformConfig to set Hadoop options. \*/

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

super.transformConfig(config) ++ Map(

"mapreduce.output.fileoutputformat.compress" -> "true",

"mapreduce.output.fileoutputformat.compress.codec" -> "com.hadoop.compression.lzo.LzoCodec",

"mapreduce.output.fileoutputformat.compress.type" -> "BLOCK"

)

/\*

\* Use getNamedOptions to set Summingbird runtime options

\* The options we set are:

\* 1) Set monoid to non-commutative to disable map-side

\* aggregation and force all aggregation to reducers (provides a 20% speedup)

\*/

override def getNamedOptions: Map[String, Options] = Map(

"DEFAULT" -> Options()

.set(MonoidIsCommutative(false))

.set(Reducers(numReducers))

)

implicit val batcher: Batcher = Batcher.ofHours(24)

/\* State implementation that uses Statebird (go/statebird) to track the batches processed. \*/

def getWaitingState(hadoopConfig: Configuration, startDate: Option[Timestamp], batches: Int) =

StatebirdState(

jobName,

startDate,

batches,

args.optional("statebird\_service\_destination"),

args.optional("statebird\_client\_id\_name")

)(batcher)

val sourceNameFilter: Option[Set[String]] =

args.optional("input\_sources").map(\_.split(",").toSet)

val storeNameFilter: Option[Set[String]] =

args.optional("output\_stores").map(\_.split(",").toSet)

val filteredAggregates =

AggregatesV2Job.filterAggregates(

aggregates = aggregatesToCompute,

sourceNames = sourceNameFilter,

storeNames = storeNameFilter

)

override val graph =

AggregatesV2Job.generateJobGraph[Scalding](

filteredAggregates,

dataRecordSourceToScalding,

aggregateStoreToScalding

)(DataRecordAggregationMonoid(filteredAggregates))

}

def main(args: Array[String]): Unit = {

GenericRunner(args, generate(\_))

}

}