import logging

import confluent\_kafka

from confluent\_kafka import Consumer, OFFSET\_BEGINNING

from confluent\_kafka.avro import AvroConsumer,CachedSchemaRegistryClient

from confluent\_kafka.avro.serializer import SerializerError

from tornado import gen

logger = logging.getLogger(\_\_name\_\_)

class KafkaConsumer:

&quot;&quot;&quot;Defines the base kafka consumer class&quot;&quot;&quot;

&quot;&quot;&quot;Defines the base kafka consumer class&quot;&quot;&quot;

def \_\_init\_\_(

self,

topic\_name\_pattern,

message\_handler,

is\_avro=True,

offset\_earliest=False,

sleep\_secs=1.0,

consume\_timeout=0.1,

):

&quot;&quot;&quot;Creates a consumer object for asynchronous use&quot;&quot;&quot;

self.topic\_name\_pattern = topic\_name\_pattern

self.message\_handler = message\_handler

self.sleep\_secs = sleep\_secs

self.consume\_timeout = consume\_timeout

self.offset\_earliest = offset\_earliest

# Configure broker # Broker properties

self.broker\_properties = {

&quot;bootstrap.servers&quot; : &quot;localhost:9092&quot;,

&quot;group.id&quot; : &quot;my-consumer-group&quot;,}

if is\_avro is True:

self.broker\_properties[&quot;schema.registry.url&quot;] = &quot;http://localhost:8081&quot;

self.consumer = AvroConsumer(config = self.broker\_properties)

else:

self.consumer = Consumer(self.broker\_properties)

self.consumer.subscribe([self.topic\_name\_pattern], on\_assign=self.on\_assign)

def on\_assign(self, consumer, partitions):

for partition in partitions:

partition.offset = OFFSET\_BEGINNING

logger.info(&quot;partitions assigned for %s&quot;, self.topic\_name\_pattern)

consumer.assign(partitions)

async def consume(self):

while True:

num\_results = 1

while num\_results &gt; 0:

num\_results = self.\_consume()

await gen.sleep(self.sleep\_secs)

def \_consume(self):

message = self.consumer.poll(1.0)

if message is None:

logger.info(&quot;No message received by consumer.&quot;)

return 0

elif message.error() is not None:

logger.debug(f&quot;error from consumer {message.error()}&quot;)

return 0

else:

try:

logger.info(message.value())

return 1

except KeyError as e:

logger.info(f&quot;Failed to unpack message {e}&quot;)

return 0

def close(self):

self.consumer.close()

logger.info(&quot;Shutting down consumer.&quot;)

import logging

import faust

logger = logging.getLogger(\_\_name\_\_)

class Station(faust.Record):

stop\_id: int

direction\_id: str

stop\_name: str

station\_name: str

station\_descriptive\_name: str

station\_id: int

order: int

red: bool

blue: bool

green: bool

class TransformedStation(faust.Record):

station\_id: int

station\_name: str

order: int

line:

out\_topic = app.topic(&quot;faust.stations.transformed&quot;, partitions=1, value\_type=TransformedStation)

table = app.Table(

&quot;stations.transformation.table&quot;,

default=int,

partitions=1,

changelog\_topic=out\_topic

@app.agent(topic)

async def StationProcess(stations):

async for station in stations:

transformed\_line = &quot;&quot;

if(station.red == True):

transformed\_line = &quot;red&quot;

elif(station.blue == True):

transformed\_line = &quot;blue&quot;

elif(station.green == True):

transformed\_line = &quot;green&quot;

else:

transformed\_line = &quot;null&quot;

transformed\_station = TransformedStation(

station\_id=station.station\_id,

station\_name=station.station\_name,

order=station.order,

line=transformed\_line

)

await out\_topic.send(value=transformed\_station)

if \_\_name\_\_ == &quot;\_\_main\_\_&quot;:app.main()

def run\_server():

if topic\_check.topic\_exists(&quot;TURNSTILE\_SUMMARY&quot;) is False:

logger.fatal(

&quot;Ensure that the KSQL Command has run successfully before running the web server!&quot;

)

exit(1)

if topic\_check.topic\_pattern\_match(&quot;faust.stations.transformed&quot;) is False:

logger.fatal(

&quot;Ensure that Faust Streaming is running successfully before running the web server!&quot;

)

exit(1)

weather\_model = Weather()

lines = Lines()

application = tornado.web.Application(

[(r&quot;/&quot;, MainHandler, {&quot;weather&quot;: weather\_model, &quot;lines&quot;: lines})]

)

application.listen(WEB\_SERVER\_PORT)

print(&quot;Building consumers....&quot;)

consumers = [

KafkaConsumer(

&quot;weather&quot;,

weather\_model.process\_message,

offset\_earliest=True,

),

KafkaConsumer(

&quot;faust.stations.transformed&quot;,

lines.process\_message,

offset\_earliest=True,

is\_avro=False,

),

SIMULATION

import datetime

import time

from enum import IntEnum

import logging

import logging.config

from pathlib import Path

import pandas as pd

# Import logging before models to ensure configuration is picked up

logging.config.fileConfig(f&quot;{Path(\_\_file\_\_).parents[0]}/logging.ini&quot;)

from connector import configure\_connector

from models import Line, Weather

logger = logging.getLogger(\_\_name\_\_)

class TimeSimulation:

weekdays = IntEnum(&quot;weekdays&quot;, &quot;mon tue wed thu fri sat sun&quot;, start=0)

ten\_min\_frequency = datetime.timedelta(minutes=10)

def \_\_init\_\_(self, sleep\_seconds=5, time\_step=None, schedule=None):

&quot;&quot;&quot;Initializes the time simulation&quot;&quot;&quot;

self.sleep\_seconds = sleep\_seconds

self.time\_step = time\_step

if self.time\_step is None:

self.time\_step = datetime.timedelta(minutes=self.sleep\_seconds)

# Read data from disk

self.raw\_df = pd.read\_csv(

f&quot;{Path(\_\_file\_\_).parents[0]}/data/cta\_stations.csv&quot;

).sort\_values(&quot;order&quot;)

# Define the train schedule (same for all trains)

self.schedule = schedule

if schedule is None:

self.schedule = {

TimeSimulation.weekdays.mon: {0: TimeSimulation.ten\_min\_frequency},

TimeSimulation.weekdays.tue: {0: TimeSimulation.ten\_min\_frequency},

TimeSimulation.weekdays.wed: {0: TimeSimulation.ten\_min\_frequency},

TimeSimulation.weekdays.thu: {0: TimeSimulation.ten\_min\_frequency},

TimeSimulation.weekdays.fri: {0: TimeSimulation.ten\_min\_frequency},

TimeSimulation.weekdays.sat: {0: TimeSimulation.ten\_min\_frequency},

TimeSimulation.weekdays.sun: {0: TimeSimulation.ten\_min\_frequency},

}

self.train\_lines = [

Line(Line.colors.blue, self.raw\_df[self.raw\_df[&quot;blue&quot;]]),

Line(Line.colors.red, self.raw\_df[self.raw\_df[&quot;red&quot;]]),

Line(Line.colors.green, self.raw\_df[self.raw\_df[&quot;green&quot;]]),

]

def run(self):

curr\_time = datetime.datetime.utcnow().replace(

hour=0, minute=0, second=0, microsecond=0

)

logger.info(&quot;Beginning simulation, press Ctrl+C to exit at any time&quot;)

logger.info(&quot;loading kafka connect jdbc source connector&quot;)

configure\_connector()

logger.info(&quot;beginning cta train simulation&quot;)

weather = Weather(curr\_time.month)

try:

while True:

logger.debug(&quot;simulation running: %s&quot;, curr\_time.isoformat())

# Send weather on the top of the hour

if curr\_time.minute == 0:

weather.run(curr\_time.month)

\_ = [line.run(curr\_time, self.time\_step) for line in self.train\_lines]

curr\_time = curr\_time + self.time\_step

time.sleep(self.sleep\_seconds)

except KeyboardInterrupt as e:

logger.info(&quot;Shutting down&quot;)= [line.close() for line in self.train\_lines]

if \_\_name\_\_ == &quot;\_\_main\_\_&quot;: TimeSimulation().run()