Task d: Streaming Application

Reads stream in batches of 10 second intervals. Stores it to a separate collection in mongoDB, called "climate_streaming" which is a different collection, "climate" from that in part A.

Clarification

- As explained the Ed forum posts, we will handle misaligned dates of hotspot and climate data by updating the hotspot data to match that of climate data which are in the SAME BATCH.
- This is perfectly legal, since streaming data by its original nature, has no date and datetime information yet.
- note that climate_id and hotspot_id are not meant to be UNIQUE within the mongo collection
- however, the "_id" field OR the combination of "producer" field and "climate_id" or "hotspot_id" field makes a particular record unique within the collection

In [ ]:
```python
strings = ['a', 'b', 'c']
def add_comma(lst):
    string = ""
    for i in range(0, len(lst) - 1):
        string += f'{lst[i]},'
    string += str(lst[len(lst) - 1])
    return string
print(add_comma(strings))
```

imports

In [ ]:
```python
from pymongo import MongoClient
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, split, element_at, when, from_json
from pyspark.sql.types import *

import datetime
from datetime import date
from datetime import datetime
from dateutil.parser import parse

import pygeohash as pgh

import copy
```

mongoDB stuff

In [ ]:
```python
# MongoDB stuff
client = MongoClient() # connect on the default host and port
db = client.fit3182_assignment_db
climate_streaming = db.climate_streaming # if not exist yet, then create
```

Fieldname constants

In [ ]:
```python
# FIELDNAMES
PRODUCER = "producer"
# common
DATE = "date"
LAT = "latitude"
LNG = "longitude"
# climate
CLIMATE_ID = "climate_id"
AIR_TEMP = "air_temperature_celcius"
RELATIVE_HUMIDITY = "relative_humidity"
WINDSPEED = "windspeed_knots"
MAX_WINDSPEED = "max_wind_speed"
PRECIPITATION = "precipitation"
GHI = "GHI_w/m2"
HOTSPOT_ENTRIES = "hotspot_entries"
# hotspot
DATETIME = "datetime"
HOTSPOT_ID = "hotspot_id"
SURFACE_TEMP = "surface_temperature_celcius"
CONFIDENCE = "confidence"
```

Other Constants

In [ ]:
```python
# NOTE: currently using a new collection called climate_streaming
HOST_NAME = 'localhost'
PORT_NUMBER =  9092
TOPIC =  "assignment"
```

Initialize our spark session with `#threads = #logicalCPU` and the given application name.

In [ ]:
```python
# run locally use all available threads
spark = (
    SparkSession.builder
    .master('local[*]')
    .appName('Assignment Streaming')      # localhost: 4040
    .getOrCreate()
)
```

Create a streaming dataframe with options providing the bootstrap server(s) and topic name.

In [ ]:
```python
# spark.read is for batch data spark.readStream is for stream data

# original stream from the topics
kafka_sdf = (
    spark.readStream.format('kafka')     # specify the data source
    .option('kafka.bootstrap.servers', f'{HOST_NAME}:{PORT_NUMBER}')     # can have multiple brokers
    .option('subscribe', f'{TOPIC}')       # subscribes to the topics, can subscribe to multiple topics
    .load()                                # create a streaming dataframe
)
```

Struct Type Object

StructField https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.types.StructField.html?highlight=structfield#pyspark.sql.types.StructField

Types https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.functions.from_json.html?highlight=from_json

StructType https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.types.StructType.html?highlight=structtype

In [ ]:
```python
# adds StructFields to StructType, nullable=True by default
# https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.sql.functions.from_json.html?highlight=from_json

data_schema = (
    StructType()
    # common stuff (wouldnt have nulls)
    .add("producer",StringType())
    .add("date", StringType())
    # climate stuff
    .add("climate_id", IntegerType())
    .add("station", StringType())
    .add("air_temperature_celcius", IntegerType())
    .add("relative_humidity", FloatType())
    .add("windspeed_knots", FloatType())
    .add("max_wind_speed", FloatType())
    .add("precipitation", StringType())
    .add("GHI_w/m2", IntegerType())
    # hotspot stuff
    .add("hotspot_id", IntegerType())
    .add("latitude", FloatType())
    .add("longitude", FloatType())
    .add("datetime", StringType())
    .add("confidence", IntegerType())
    .add("surface_temperature_celcius", IntegerType())
)
```

In [ ]:
```python
# converts the original df to a values spread across multiple columns
converted_stream_df = (
    kafka_sdf
    # get json_string
    .select(
        kafka_sdf.value.cast("string").alias("json_string")
    )
    # convert from json_string to columns data
    .select(
        from_json("json_string", data_schema).alias("columns")
    )
    # spreading out columns
    .select("columns.*")
)
```

In [ ]:
```python
kafka_sdf
```

In [ ]:
```python
kafka_sdf.printSchema()
```

In [ ]:
```python
converted_stream_df
```

In [ ]:
```python
converted_stream_df.printSchema()
```

Utility functions

In [ ]:
```python
HISTORIC_PRODUCER = "climate_historic"

def fallback_historic(historic_producer = HISTORIC_PRODUCER):
    result = climate.delete_many({"producer": {"$ne": historic_producer}})
    return result.deleted_count

def format_dict_list(lst):
    ret = "[\n"
    for _dict in lst:
        ret += str(_dict)
        ret += "\n"
    ret += "]"
    return ret

def format_pgh_group(groups):
    ret = "{\n"
    for key in groups:
        d_lst = groups[key]
        ret += f'"{key}": {format_dict_list(d_lst)} \n'
    ret += "}"
    return ret

def format_c_dict(dic):
    ret = "{\n"
    for key in dic:
        if key != HOTSPOT_ENTRIES:
            ret += f'"{key}": {dic[key]}, '
        else:
            ret += f'"{key}": {format_dict_list(dic[key])}, '
    ret += "\n}"
    return ret

def debug_print(d_nested_lst, batch_id, print_f, stage=None, stage_dataset_names = [], producer_name="None"):
    assert_msg = f'd_nested_lst: {d_nested_lst} \n stage_dataset_names: {stage_dataset_names}'
    assert(len(stage_dataset_names) == 0 or len(d_nested_lst)==len(stage_dataset_names))
    if stage is not None:
        print('-'*20)
        print(f'stage: {stage}')
        print('-'*20)
    elif stage is None:
        print("\n")
        print('-'*20)
        print(f"batch: {batch_id}")
        print('-'*20)
    # assumes that dict_lst has been filtered to have same producer for each data entry
    if len(stage_dataset_names) == 0:
        for dict_lst in d_nested_lst:
            producer = dict_lst[0]["producer"] if len(dict_lst) > 0 else producer_name
            print(f'{len(dict_lst)}, producer: {producer}, data: {print_f(dict_lst)}')

    elif len(stage_dataset_names) > 0 and len(stage_dataset_names) == len(d_nested_lst):
        try:
            for i in range(len(d_nested_lst)):
                dict_lst = d_nested_lst[i]
                string = f'{len(dict_lst)}, '
                string += f'stage_dataset: {stage_dataset_names[i]}, '
                string += f' data: {print_f(dict_lst)}'
                print(string)
        except IndexError:
            errmsg = f'd_nested_lst: {d_nested_lst} \n stage_dataset_names: {stage_dataset_names}'
            raise IndexError(errmsg)
    else:
        errmsg = f'd_nested_lst: {d_nested_lst} \n stage_dataset_names: {stage_dataset_names}'
        len1, len2 = len(stage_dataset_names), len(d_nested_lst)
        raise Exception(f"len(stage_dataset_names) {len1} != len(d_nested_lst) {len2}, , {errmsg}")


def separate_producer(lst, producer_name_lst):
    res = {} # list of dictionaries, key is producer name
    for producer in producer_name_lst:
        res[producer] = []

    for dict_obj in lst:
        producer = dict_obj["producer"]
        res[producer].append(dict_obj)

    # can access different producer partitions using producer name as key
    return res

def str2date(x, field_name):
    try:
        if x[field_name] is not None:
            parsed = parse(x[field_name], ignoretz=None, dayfirst=True)
        x[field_name] = parsed
        return x
    except TypeError:
        errmsg = f'{x}, {field_name}'
        raise TypeError(errmsg)

# def fallback_historic(collection, start):
#     result = collection.delete_many({'x': {"$gte": start}})
#     return result.deleted_count
def my_sum(lst):
    res = 0
    for x in lst:
        res += x
    return res

def my_avg(lst, intFlag = True):
    try:
        # average is an integer in this case
        int_avg = my_sum(lst) / len(lst)
        if intFlag:
            int_avg = round(int_avg)
        return int_avg
    except TypeError:
        errmsg = str(lst)
        raise TypeError(errmsg)

# def sort_based_aggregate(lst1, lst2, field_name):
#     sorted1 = sorted(lst1, lambda x: x[field_name])
#     sorted2 = sorted(lst2, lambda x: x[field_name])

#     i, j = 0, 0
#     while i <= len(sorted1) and j <= len(sorted2):
#         rec1, rec2 = sorted1[i], sorted2[j]
#         if rec1 == rec2:

def check_same_pgh_code(d_lst, lat_lng_to_check, precision_val):
    pgh_check_code = pgh.encode(*lat_lng_to_check, precision = precision_val)
    for dic in d_lst:
        code = pgh.encode(dic[LAT], dic[LNG], precision=precision_val)
        if pgh_check_code != code:
            return False
    return True


def check_all_same_pgh_code(d_lst, lat_lng_to_check, precision_val):
    pgh_check_code = pgh.encode(*lat_lng_to_check, precision = precision_val)
    for dic in d_lst:
        code = pgh.encode(dic[LAT], dic[LNG], precision=precision_val)
        if pgh_check_code != code:
            return False
    return True

def pgh_from_pair(pair, precision_val):
    return pgh.encode(*pair, precision = precision_val)

def check_pair_same_pgh_code(pair1, pair2, precision_val):
    code1 = pgh_from_pair(pair1, precision_val)
    code2 = pgh_from_pair(pair2, precision_val)
    return code1 == code2

def lat_lng_pair_from_dict(dic):
    return dic[LAT], dic[LNG]


def sort_dict_lst_on_field(d_lst, field_name):
    f = lambda dic : dic[field_name]
    return sorted(d_lst, f)


def pgh_grouping(d_lst):
    """
    Inspired by hash-based serial group by
    returns a dictionary object, with key-value pairs of:
    key: gph code,
    value: a lst of records with matching gph code
    """
    groups = {}
    for rec in d_lst:
        pgh_code = pgh.encode(rec[LAT], rec[LNG], precision=5)
        # if not exist yet, create new group
        if pgh_code not in groups:
            groups[pgh_code] = [rec]
        # else append to group
        else:
            groups[pgh_code].append(rec)
    return groups



def agg_pgh_groups(pgh_groups, field_name_lst, agg_f_lst, producer_lst):
    """
    # GOAL: compress group records into only 1 record per group, on the key_f used
    # 1: computes the aggregated value, based on all given field_name_lst
    # 2: replace the group lst of dicts, with only 1 aggregated record
    """
    assert(len(field_name_lst) == len(agg_f_lst))
    agg_groups = copy.deepcopy(pgh_groups) # make a copy first to prevent mutations
    for pgh_key in agg_groups:
        group = agg_groups[pgh_key]
        # only aggregate if more than 1 rec in group
        if len(group) > 1:
            print("agg_pgh_groups: going to run aggregate on more than 1 records! ")
            agg_rec = copy.deepcopy(group[0]) # initialize to first record's copy

            for i in range(len(field_name_lst)):
                print(f"agg_pgh_groups: ran agg loop {i+1} times !")
                field, agg_f = field_name_lst[i], agg_f_lst[i]
                field_values = [dic[field] for dic in group] # extract fields value
                agg_val = agg_f(field_values)
                agg_rec[field] = agg_val # update value in agg record

            agg_rec[PRODUCER] = f'merged: {producer_lst}' # update producer to indicate merged
            # update pgh_key group to only contain agg_rec
            print(f"agg_rec: {agg_rec} !")
            agg_groups[pgh_key] = [agg_rec]
            print(f"updated groups with agg_rec: {agg_groups} !")
    return agg_groups


# update the fire_cause for fire data
def unpack_all(nested):
    ret = []
    for lst in nested:
        ret += lst
    return ret
```

Db_writer and Console_logger

In [ ]:
```python
# 1: takes the stream DF
# 2: write stream
# 3: specify output mode (append)
# 4: run a custom function row by row, the logic is specified in DbWriter object

# TODO: check whether this should be 10 seconds
BATCH_PROCESSING_TIME = 10

# db_writer = (
#     output_stream_df
#     .writeStream
#     .outputMode('append')
#     .foreach(DbWriter())      # specify foreach() instead of format()
#                               # instead of passing an entire batch of data, pass a SINGLE row
#     .trigger(processingTime = f'{BATCH_PROCESSING_TIME} seconds')
# )


# just for debugging in the console
# output_stream_df OR topic_stream_df
# console_logger = (
#     converted_stream_df
#     .writeStream
#     .outputMode('append')
#     .format('console')     # specify where we want to send it
#     # .option("truncate", False)
#     .trigger(processingTime = f'{BATCH_PROCESSING_TIME} seconds')
# )


def batchFunc(batch_df, batch_id):
    raw_data = batch_df.collect() # returns a list of Row objects
    dict_lst = [row.asDict() for row in raw_data]
    # print(type(dict_lst))



    # PREPROCESSING
    # filter for climate and hotspot
    climate_data = [x for x in dict_lst if x["climate_id"] is not None]
    hotspot_data = [x for x in dict_lst if x["hotspot_id"] is not None]
    debug_print([climate_data, hotspot_data],
                batch_id,
                stage=None,
                print_f = format_dict_list,
                stage_dataset_names = ["climate_data", "hotspot_data"])

    # convert strings to datetime objects
    if len(climate_data) > 0:
        climate_data = [str2date(x, "date") for x in climate_data]
    if len(hotspot_data) > 0:
        hotspot_data = [str2date(x, "date") for x in hotspot_data]
        hotspot_data = [str2date(x, "datetime") for x in hotspot_data]

    debug_print([climate_data, hotspot_data],
                batch_id,
                print_f = format_dict_list,
                stage=f"after convert str to datetime",
                stage_dataset_names = ["climate_data", "hotspot_data"])

    # debug_print(climate_data, hotspot_data, batch_id)

    # ASSIGNMENT LOGIC
    # 1: OUTLIERS,
    # 1.1 NO climate data in data array, discard all results in batch
    # 1.1 check if there is climate data in data array
    # if not, then return
    if len(climate_data) == 0:
        # print("filter hotspot loc tally climate's")
        debug_print([climate_data],
                    batch_id,
                    print_f = format_dict_list,
                    stage=f"no climate data")
        return True

    elif len(climate_data) > 1:
        raise ValueError("there is more than 1 climate data in this batch!")

    elif len(climate_data) == 1:
        # 2:  update the date for all of the hotspot data to tally with data of climate date
        # climate_date = parse(climate_data[0]["date"], ignoretz=True)
        c_dict = climate_data[0]

        climate_date = c_dict["date"]
        for row_dict in hotspot_data:
            date_time = row_dict['datetime']
            # datetime = parse(row_dict['datetime'], ignoretz=True)

            # 2.1: update for date field
            row_dict["date"] = climate_date
            # 2.2: update for datetime field
            # update the day-mon-yea
            year = climate_date.year
            month = climate_date.month
            day = climate_date.day
            # maintain the previous HH:MI:SS
            hour, minute, second = date_time.hour, date_time.minute, date_time.second

            new_datetime = datetime(year, month, day, hour, minute, second)
            row_dict['datetime'] = new_datetime

        # 3.2:
        hotspot_aqua_and_terra = separate_producer(hotspot_data, ["hotspot_AQUA", "hotspot_TERRA"])
        hotspot_aqua = hotspot_aqua_and_terra["hotspot_AQUA"]
        hotspot_terra = hotspot_aqua_and_terra["hotspot_TERRA"]

        debug_print([climate_data, hotspot_aqua, hotspot_terra],
                    batch_id,
                    print_f = format_dict_list,
                    stage = "updated hotspot date and datetime to match climate's date")

        # climate = {}, hotspots = [{}, {}, {}, ...]
        # 3: filter hotspot data with same location as climate
        c_dict = climate_data[0]
        c_geohash = pgh.encode(c_dict[LAT], c_dict[LNG], precision=3)

        loc_filtered_hotspots = []

        for h_dict in hotspot_data:
            h_geohash = pgh.encode(h_dict[LAT], h_dict[LNG], precision=3)
            # filter both aqua and terra data to have same location as climate data
            if h_geohash == c_geohash:
                loc_filtered_hotspots.append(h_dict)

        # print("filter hotspot loc tally climate's")
        debug_print([loc_filtered_hotspots],
                    batch_id,
                    print_f = format_dict_list,
                    stage=f"hotspot loc filter {c_dict[LAT]}, {c_dict[LNG]}",
                    stage_dataset_names=["loc_filtered_hotspots"])

        # 4: if hotspots = [], insert climate data to mongoDB, return
        if len(loc_filtered_hotspots) == 0:
            # insert empty hotspot_entries lst into the dict object
            embedded = c_dict
            embedded["hotspot_entries"] = []
            climate_streaming.insert_one(embedded)
            # self.db[topic].replace_one({'_id': data.get('bay_id')}, db_record, upsert=True)
            debug_print([climate_data, loc_filtered_hotspots], batch_id,
                        stage= f"only climate data after loc_filter, inserted to db",
                        print_f = format_dict_list,
                        stage_dataset_names = ["climate_data", "loc_filtered_hotspots"])
            return True

        # 5: if hotspots = [{}, ...]
        # group hotspot data by location and aggregate average of surface temperature and confidence
        # [{pgh_encode: ab123,...}, {pgh_encode: ab123, ...}, {pgh_encode: ab123, ...}]
        elif len(loc_filtered_hotspots) > 0:
#             filtered_aqua_and_terra = separate_producer(loc_filtered_hotspots,
#                                                         ["hotspot_AQUA", "hotspot_TERRA"])
#             filtered_aqua = filtered_aqua_and_terra["hotspot_AQUA"]
#             filtered_terra = filtered_aqua_and_terra["hotspot_TERRA"]

            # compiled_hotspot = []
            #
            # if len(filtered_aqua) == 0 or len(filtered_terra) == 0:
            #     compiled_hotspot = loc_filtered_hotspots


            # perform serial group by on gph code , returns
            pgh_groups = pgh_grouping(loc_filtered_hotspots)
            debug_print([pgh_groups],
                        batch_id,
                        print_f = format_pgh_group,
                        stage=f"after groups, before agg",
                        stage_dataset_names = ["pgh_groups"])
            # perform aggregate on each groups
            agg_groups = agg_pgh_groups(pgh_groups,
                                        [SURFACE_TEMP, CONFIDENCE],
                                        [my_avg, my_avg],
                                        ["AQUA", "TERRA"])
            debug_print([agg_groups],
                        batch_id,
                        print_f = format_pgh_group,
                        stage=f"groups after aggregate",
                        stage_dataset_names = ["aggregated_h_groups"])

            agg_groups = [agg_groups[pgh_key] for pgh_key in agg_groups]
            agg_hotspot = unpack_all(agg_groups)

            # 6: detect cause of fire
            natural_fire_flag = c_dict["air_temperature_celcius"] > 20 and c_dict["GHI_w/m2"] > 180
            fire_cause = "natural" if natural_fire_flag else "other"

            for h_dict in agg_hotspot:
                h_dict["fire_cause"] = fire_cause

            # 7: do embedding of documents, hotspot_entries field within climate_data
            c_dict = climate_data[0]
            c_dict["hotspot_entries"] = agg_hotspot

            # print("compiled")
            debug_print([agg_hotspot],
                        batch_id,
                        print_f = format_dict_list,
                        stage=f"final",
                        stage_dataset_names = ["finalized data"])

            # climate = {}, hotspots = [{}] or [{}, {}, {}]
            # 8: insert climate and hotspot data in mongoDB
            climate_streaming.insert_one(c_dict)

            debug_print([c_dict],
                        batch_id,
                        print_f = format_c_dict,
                        stage=f"inserted to db",
                        stage_dataset_names = ["c_dict"])

            return True
    raise Exception("shouldnt come here, shouldve returned earlier")

db_writer = (
    converted_stream_df
    .writeStream
    .outputMode('append')
    .trigger(processingTime = f'{BATCH_PROCESSING_TIME} seconds')
#     .format('console')     # specify where we want to send it
    .foreachBatch(batchFunc)
)
```

In [ ]:
```python
db_writer
```

start streaming query

In [ ]:
```python
writer = db_writer
try:
    query = writer.start()
    query.awaitTermination()
except KeyboardInterrupt:
    print('Interrupted by CTRL-C. Stopped query')
except StreamingQueryException as exc:
    print(exc)
finally:
    query.stop()
```