import pandas as pd

import boto3

import io

import time

def process\_dataframe\_with\_athena(df\_input, access\_key, secret\_key):

# Set up AWS clients

s3\_client = boto3.client('s3', region\_name='us-east-1',

aws\_access\_key\_id=access\_key,

aws\_secret\_access\_key=secret\_key)

athena\_client = boto3.client('athena', region\_name='us-east-1',

aws\_access\_key\_id=access\_key,

aws\_secret\_access\_key=secret\_key)

source\_db = 'devoted\_health\_prod'

QUERY\_OUTPUT\_LOCATION = "s3://zignaai-deidentified-claimsdata/query output/"

WORKGROUP\_NAME = 'SelectionQueries-Production'

def run\_query(query\_string):

try:

response = athena\_client.start\_query\_execution(

QueryString=query\_string,

QueryExecutionContext={'Database': source\_db},

ResultConfiguration={'OutputLocation': QUERY\_OUTPUT\_LOCATION},

WorkGroup=WORKGROUP\_NAME

)

query\_execution\_id = response['QueryExecutionId']

# Wait for completion

while True:

status = athena\_client.get\_query\_execution(QueryExecutionId=query\_execution\_id)

state = status['QueryExecution']['Status']['State']

if state in ['SUCCEEDED', 'FAILED', 'CANCELLED']:

break

time.sleep(2)

if state == 'SUCCEEDED':

result\_path = status['QueryExecution']['ResultConfiguration']['OutputLocation']

bucket = result\_path.split('//')[1].split('/')[0]

key = '/'.join(result\_path.split('//')[1].split('/')[1:])

buffer = io.BytesIO(s3\_client.get\_object(Bucket=bucket, Key=key)['Body'].read())

return pd.read\_csv(buffer, dtype=str)

else:

print(f"Query failed with status: {state}")

except Exception as e:

print("Error running Athena query:", e)

return None

# Athena query

query\_string = """SELECT \* FROM "zigna\_reference\_data"."reference\_table\_add\_on\_codes" WHERE aoc\_edit\_type = 1"""

reference\_data = run\_query(query\_string)

# Process reference data

ref = reference\_data[['add\_on\_code', 'primary\_code', 'aoc\_edit\_effdt']].drop\_duplicates()

ref['aoc\_edit\_effdt'] = ref['aoc\_edit\_effdt'].apply(lambda x: pd.to\_datetime(str(x), format='%Y%j').strftime('%Y-%m-%d'))

overlapping = set(ref['add\_on\_code']).intersection(set(ref['primary\_code']))

ref = ref[~ref['add\_on\_code'].isin(overlapping)]

ref = ref[~ref['primary\_code'].isin(overlapping)]

# Prepare mappings

df\_input['plan\_paid\_amount'] = df\_input['plan\_paid\_amount'].astype(float)

add\_on\_map = ref.groupby('add\_on\_code').apply(lambda x: set(x['primary\_code']))

add\_on\_map = add\_on\_map.reset\_index(name='primary\_code\_array')

df = df\_input.merge(add\_on\_map, left\_on='procedure\_code', right\_on='add\_on\_code', how='left')

df['is\_add\_on'] = df.apply(lambda x: '1' if x['plan\_paid\_amount'] >= 75 and pd.notna(x['add\_on\_code']) else '0', axis=1)

# Map add\_on to primary codes

temp = df[(df['is\_add\_on'] == '1') & (df['plan\_paid\_amount'] > 75)][['payer\_control\_number', 'add\_on\_code', 'primary\_code\_array']]

grouped = temp.groupby('payer\_control\_number')['primary\_code\_array'].apply(lambda x: x.tolist()[0]).reset\_index()

grouped['primary\_code'] = grouped['primary\_code\_array']

grouped.drop(columns=['primary\_code\_array'], inplace=True)

df = df.merge(grouped, how='left', on='payer\_control\_number')

df['is\_primary'] = df.apply(lambda row: '1' if row['procedure\_code'] in row['primary\_code'] else '0', axis=1)

df['service\_date'] = pd.to\_datetime(df['service\_date'])

df['proc\_code\_flag'] = 'other'

for idx, row in df[df['is\_add\_on'] == '1'].iterrows():

match = df[

(df['payer\_control\_number'] == row['payer\_control\_number']) &

(df['service\_date'] == row['service\_date']) &

(df['is\_primary'] == '1')

]

if not match.empty:

df.at[idx, 'proc\_code\_flag'] = 'target'

for m\_idx in match.index:

df.at[m\_idx, 'proc\_code\_flag'] = 'reference'

# Drop PCNs without targets

valid\_pcns = set(df[df['proc\_code\_flag'] == 'target']['payer\_control\_number'].unique())

df = df[df['payer\_control\_number'].isin(valid\_pcns)]

# Drop PCNs where procedure code is in primary\_code

conflicted = set(ref["add\_on\_code"]).intersection(ref["primary\_code"])

drop\_pcns = df[(df['proc\_code\_flag'] == 'target') & (df['procedure\_code'].isin(conflicted))]['payer\_control\_number'].unique()

df = df[~df['payer\_control\_number'].isin(drop\_pcns)]

# Rule 3 drop

drop\_pcns = df[(df['proc\_code\_flag'] == 'reference') & (df['plan\_paid\_amount'] > 0)]['payer\_control\_number'].unique()

df = df[~df['payer\_control\_number'].isin(drop\_pcns)]

# Check if any add\_on has CCCCC

temp\_check = ref.groupby('add\_on\_code')['primary\_code'].unique().reset\_index()

temp\_check['is\_ccccc'] = temp\_check['primary\_code'].astype(str).apply(lambda x: '1' if "CCCCC" in x else '0')

codes\_to\_drop = temp\_check[temp\_check['is\_ccccc'] == '1']['add\_on\_code'].unique()

# PCN has both target and reference

for i in set(df["payer\_control\_number"]):

z = df[df["payer\_control\_number"] == i]

try:

flags = z['proc\_code\_flag'].unique()

if not ('target' in flags and 'reference' in flags):

print(f"PCN missing both target/reference: {i}")

except Exception as e:

print(f"Error with PCN {i}: {e}")

df.sort\_values(by=[

'payer\_control\_number', 'adjudication\_record\_locator', 'member\_record\_locator',

'member\_medicare\_id', 'rendering\_provider\_npi', 'service\_date'

], inplace=True)

df.reset\_index(drop=True, inplace=True)

return df