```
from pyspark.sql.types import *
import pyspark.sql.functions as f

from pyspark import SparkContext
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("DMA popularity - Project Part 1
- Section 2").getOrCreate()
sc = spark.sparkContext

# Read a CSV into a dataframe
# There is a smarter version, that will first check if there is a
Parquet file and use it
def load_csv_file(filename, schema):
  # Reads the relevant file from distributed file system using the
given schema

  allowed_files = {'Daily program data': ('Daily program data', "|"),
                   'demographic': ('demographic', "|")}

  if filename not in allowed_files.keys():
    print(f'You were trying to access unknown file \"{filename}\".
Only valid options are {allowed_files.keys()}')
    return None

  filepath = allowed_files[filename][0]
  dataPath = f"dbfs:/mnt/coursedata2024/fwm-stb-data/{filepath}"
  delimiter = allowed_files[filename][1]

  df = spark.read.format("csv")\
    .option("header","false")\
    .option("delimiter",delimiter)\
    .schema(schema)\
    .load(dataPath)
  return df

# This dict holds the correct schemata for easily loading the CSVs
schemas_dict = {'Daily program data':
                  StructType([
                    StructField('prog_code', StringType()),
                    StructField('title', StringType()),
                    StructField('genre', StringType()),
                    StructField('air_date', StringType()),
                    StructField('air_time', StringType()),
                    StructField('Duration', FloatType())
                  ]),
                'viewing':
                  StructType([
                    StructField('device_id', StringType()),
                    StructField('event_date', StringType()),
                    StructField('event_time', IntegerType()),
                    StructField('mso_code', StringType()),
                    StructField('prog_code', StringType()),
                    StructField('station_num', StringType())
                  ]),
                'viewing_full':
                  StructType([
                    StructField('mso_code', StringType()),
                    StructField('device_id', StringType()),
                    StructField('event_date', IntegerType()),
                    StructField('event_time', IntegerType()),
                    StructField('station_num', StringType()),
                    StructField('prog_code', StringType())
                  ]),
                'demographic':

StructType([StructField('household_id',StringType()),
                    StructField('household_size',IntegerType()),
                    StructField('num_adults',IntegerType()),
                    StructField('num_generations',IntegerType()),
                    StructField('adult_range',StringType()),
                    StructField('marital_status',StringType()),
                    StructField('race_code',StringType()),
                    StructField('presence_children',StringType()),
                    StructField('num_children',IntegerType()),
                    StructField('age_children',StringType()), #format
like range - 'bitwise'
                    StructField('age_range_children',StringType()),
                    StructField('dwelling_type',StringType()),
                    StructField('home_owner_status',StringType()),
                    StructField('length_residence',IntegerType()),
                    StructField('home_market_value',StringType()),
                    StructField('num_vehicles',IntegerType()),
                    StructField('vehicle_make',StringType()),
                    StructField('vehicle_model',StringType()),
                    StructField('vehicle_year',IntegerType()),
                    StructField('net_worth',IntegerType()),
                    StructField('income',StringType()),
                    StructField('gender_individual',StringType()),
                    StructField('age_individual',IntegerType()),
                    StructField('education_highest',StringType()),
                    StructField('occupation_highest',StringType()),
                    StructField('education_1',StringType()),
                    StructField('occupation_1',StringType()),
                    StructField('age_2',IntegerType()),
                    StructField('education_2',StringType()),
                    StructField('occupation_2',StringType()),
                    StructField('age_3',IntegerType()),
                    StructField('education_3',StringType()),
                    StructField('occupation_3',StringType()),
                    StructField('age_4',IntegerType()),
                    StructField('education_4',StringType()),
                    StructField('occupation_4',StringType()),
                    StructField('age_5',IntegerType()),
                    StructField('education_5',StringType()),
                    StructField('occupation_5',StringType()),
                    StructField('polit_party_regist',StringType()),
                    StructField('polit_party_input',StringType()),
                    StructField('household_clusters',StringType()),
                    StructField('insurance_groups',StringType()),
                    StructField('financial_groups',StringType()),
                    StructField('green_living',StringType())
                  ])
}

%%time
# daily_program data filename is 'Daily program data'
daily_prog_df = load_csv_file('Daily program data',
schemas_dict['Daily program data'])

daily_prog_df.printSchema()
print(f'daily_prog_df contains {daily_prog_df.count()} records!')
display(daily_prog_df.limit(6))

root
 |-- prog_code: string (nullable = true)
 |-- title: string (nullable = true)
 |-- genre: string (nullable = true)
 |-- air_date: string (nullable = true)
 |-- air_time: string (nullable = true)
 |-- Duration: float (nullable = true)

daily_prog_df contains 13194849 records!

CPU times: user 109 ms, sys: 9.38 ms, total: 118 ms
Wall time: 11 s

%%time
# reference data is stored in parquet for your convinence.

ref_df = spark.read.parquet('dbfs:/refxml_new_parquet')

ref_df.printSchema()
print(f'ref_df contains {ref_df.count()} records!')
display(ref_df.limit(6))

root
 |-- device_id: string (nullable = true)
 |-- dma: string (nullable = true)
 |-- dma_code: long (nullable = true)
 |-- household_id: long (nullable = true)
 |-- household_type: string (nullable = true)
 |-- system_type: string (nullable = true)
 |-- zipcode: long (nullable = true)

ref_df contains 1268071 records!

CPU times: user 10.2 ms, sys: 458 µs, total: 10.6 ms
Wall time: 868 ms

# Sample of 10 Million viewing entries

dataPath = f"dbfs:/viewing_10M"
viewing10m_df = spark.read.format("csv")\
    .option("header","true")\
    .option("delimiter",",")\
    .schema(schemas_dict['viewing_full'])\
    .load(dataPath)

display(viewing10m_df.limit(6))
print(f'viewing10m_df contains {viewing10m_df.count()} rows!')

viewing10m_df contains 10042340 rows!
```

Question 2

2.1

```
from pyspark.sql.functions import count, col, desc, sum

# clean df's
reference_data = ref_df\
    .dropDuplicates(['device_id'])\
        .dropna(subset=["device_id", "DMA"])\
            .select('device_id', 'DMA')\
                .filter(col("DMA") != "Unknown")


viewing_data = viewing10m_df\
    .dropDuplicates()\
        .dropna(subset=["device_id", "prog_code"])\
            .select(["device_id", "prog_code"])

daily_data = daily_prog_df\
    .dropDuplicates()\
        .dropna(subset=["genre", "prog_code"])\
            .select(["genre", "prog_code"])
```

```
viewing_with_dma = viewing_data\
    .join(reference_data, "device_id")\
        .select("DMA", "device_id", "prog_code")

# Get the top 10 DMAs by unique device count
top_dmas = reference_data\
    .groupBy("DMA")\
        .count()\
            .orderBy("count",ascending=False)\
                .limit(10)

# top 10 DMAs into a list
top_dmas_list = top_dmas.select('DMA')

# clean df's and take subset attributes that are relevant to the
question
all_viewings = viewing10m_df\
    .dropDuplicates()\
        .dropna(subset=["device_id", "prog_code", "event_date",
"event_time"])\
            .select(["device_id", "prog_code", "event_date",
"event_time"])

top_10_dma_with_genres = top_dmas_list\
    .join(reference_data, "DMA")\
        .join(all_viewings, "device_id")\
            .join(daily_data, "prog_code")\
                .dropDuplicates()\
                    .dropna(subset=["DMA", "genre", "event_date",
"event_time"])\
                        .select(["DMA", "genre", "event_date",
"event_time"])

# explode Genres to their matching DMA
top_10_dma_with_splitted_genres = top_10_dma_with_genres\
    .withColumn('genre', f.explode(f.split('genre',',')))

dma_results = []

# Iterate ofver the top DMA's and calculate the popularity for each
genre
for dma_row in top_dmas_list.rdd.toLocalIterator():
    dma = dma_row["DMA"]
    current_dma_genres = top_10_dma_with_splitted_genres\
        .filter(top_10_dma_with_splitted_genres.DMA == dma)\
            .groupBy("genre")\
                .count()\
                    .orderBy("count",ascending=False)
    dma_results.append((dma, current_dma_genres))
```

Top 10 Genres for 1,5,9-th by DMA size

```
# showing top 10 genres for 1, 5, 9th dma results by popularity
for i, (dma, result_df) in enumerate(dma_results):
    if i in [0, 4, 8]:
        print(f"Looking at DMA - {dma}")
        result_df.show(10, truncate=False)
```

```
Looking at DMA - Wilkes Barre-Scranton-Hztn
+-----------+-----+
|genre      |count|
+-----------+-----+
|News       |48754|
|Reality    |45432|
|Sitcom     |28218|
|Talk       |25046|
|Comedy     |23830|
|Crime drama|21192|
|Documentary|20894|
|Drama      |20604|
|Children   |20596|
|Action     |18877|
+-----------+-----+
only showing top 10 rows

Looking at DMA - Washington, DC (Hagrstwn)
+-----------+-----+
|genre      |count|
+-----------+-----+
|Reality    |14795|
|News       |12619|
|Sitcom     |9916 |
|Comedy     |8550 |
|Children   |7514 |
|Drama      |7337 |
|Talk       |7058 |
|Animated   |6543 |
|Documentary|6298 |
|Adventure  |6130 |
+-----------+-----+
only showing top 10 rows

Looking at DMA - Bend, OR
+------------+-----+
|genre       |count|
+------------+-----+
|News        |23612|
|Reality     |20220|
|Talk        |14944|
|Sitcom      |13122|
|Comedy      |10516|
|Documentary |8462 |
|Crime drama |8070 |
|Sports event|7907 |
|Children    |7789 |
|Drama       |7707 |
+------------+-----+
only showing top 10 rows
```

```
for i, (dma_name, result_df) in enumerate(dma_results):
    print(dma_name)
    file_name = f"project1_part21_{dma_name.replace(' ',
'_').lower()}_337604821_326922390.csv"
    result_df.write.format("csv").option("header",
"true").mode("overwrite").save(file_name)
    print(f"Saved DataFrame to {file_name}")
```

```
Wilkes Barre-Scranton-Hztn
Saved DataFrame to project1_part21_wilkes_barre-scranton-
hztn_337604821_326922390.csv
Charleston-Huntington
Saved DataFrame to project1_part21_charleston-
huntington_337604821_326922390.csv
Seattle-Tacoma
Saved DataFrame to project1_part21_seattle-
tacoma_337604821_326922390.csv
Little Rock-Pine Bluff
Saved DataFrame to project1_part21_little_rock-
pine_bluff_337604821_326922390.csv
Washington, DC (Hagrstwn)
Saved DataFrame to
project1_part21_washington,_dc_(hagrstwn)_337604821_326922390.csv
Toledo
Saved DataFrame to project1_part21_toledo_337604821_326922390.csv
Amarillo
Saved DataFrame to project1_part21_amarillo_337604821_326922390.csv
Greenville-N.Bern-Washngtn
Saved DataFrame to project1_part21_greenville-n.bern-
washngtn_337604821_326922390.csv
Bend, OR
Saved DataFrame to project1_part21_bend,_or_337604821_326922390.csv
Lubbock
Saved DataFrame to project1_part21_lubbock_337604821_326922390.csv
```

```python
from pyspark.sql.types import *
import pyspark.sql.functions as f

from pyspark import SparkContext
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("my_project_1").getOrCreate()
sc = spark.sparkContext

# Read a CSV into a dataframe
# There is a smarter version, that will first check if there is a
Parquet file and use it
def load_csv_file(filename, schema):
  # Reads the relevant file from distributed file system using the
given schema

  allowed_files = {'Daily program data': ('Daily program data', "|"),
                   'demographic': ('demographic', "|")}

  if filename not in allowed_files.keys():
    print(f'You were trying to access unknown file \"{filename}\".
Only valid options are {allowed_files.keys()}')
    return None

  filepath = allowed_files[filename][0]
  dataPath = f"dbfs:/mnt/coursedata2024/fwm-stb-data/{filepath}"
  delimiter = allowed_files[filename][1]

  df = spark.read.format("csv")\
    .option("header","false")\
    .option("delimiter",delimiter)\
    .schema(schema)\
    .load(dataPath)
  return df

# This dict holds the correct schemata for easily loading the CSVs
schemas_dict = {'Daily program data':
                  StructType([
                    StructField('prog_code', StringType()),
                    StructField('title', StringType()),
                    StructField('genre', StringType()),
                    StructField('air_date', StringType()),
                    StructField('air_time', StringType()),
                    StructField('Duration', FloatType())
                  ]),
                'viewing':
                  StructType([
                    StructField('device_id', StringType()),
                    StructField('event_date', StringType()),
                    StructField('event_time', IntegerType()),
                    StructField('mso_code', StringType()),
                    StructField('prog_code', StringType()),
                    StructField('station_num', StringType())
                  ]),
                'viewing_full':
                  StructType([
                    StructField('mso_code', StringType()),
                    StructField('device_id', StringType()),
                    StructField('event_date', IntegerType()),
                    StructField('event_time', IntegerType()),
                    StructField('station_num', StringType()),
                    StructField('prog_code', StringType())
                  ]),
                'demographic':

StructType([StructField('household_id',StringType()),
                    StructField('household_size',IntegerType()),
                    StructField('num_adults',IntegerType()),
                    StructField('num_generations',IntegerType()),
                    StructField('adult_range',StringType()),
                    StructField('marital_status',StringType()),
                    StructField('race_code',StringType()),
                    StructField('presence_children',StringType()),
                    StructField('num_children',IntegerType()),
                    StructField('age_children',StringType()), #format
like range - 'bitwise'
                    StructField('age_range_children',StringType()),
                    StructField('dwelling_type',StringType()),
                    StructField('home_owner_status',StringType()),
                    StructField('length_residence',IntegerType()),
                    StructField('home_market_value',StringType()),
                    StructField('num_vehicles',IntegerType()),
                    StructField('vehicle_make',StringType()),
                    StructField('vehicle_model',StringType()),
                    StructField('vehicle_year',IntegerType()),
                    StructField('net_worth',IntegerType()),
                    StructField('income',StringType()),
                    StructField('gender_individual',StringType()),
                    StructField('age_individual',IntegerType()),
                    StructField('education_highest',StringType()),
                    StructField('occupation_highest',StringType()),
                    StructField('education_1',StringType()),
                    StructField('occupation_1',StringType()),
                    StructField('age_2',IntegerType()),
                    StructField('education_2',StringType()),
                    StructField('occupation_2',StringType()),
                    StructField('age_3',IntegerType()),
                    StructField('education_3',StringType()),
                    StructField('occupation_3',StringType()),
                    StructField('age_4',IntegerType()),
                    StructField('education_4',StringType()),
                    StructField('occupation_4',StringType()),
                    StructField('age_5',IntegerType()),
                    StructField('education_5',StringType()),
                    StructField('occupation_5',StringType()),
                    StructField('polit_party_regist',StringType()),
                    StructField('polit_party_input',StringType()),
                    StructField('household_clusters',StringType()),
                    StructField('insurance_groups',StringType()),
                    StructField('financial_groups',StringType()),
                    StructField('green_living',StringType())
                  ])
}

%%time
# demographic data filename is 'demographic'
demo_df = load_csv_file('demographic', schemas_dict['demographic'])

demo_df.printSchema()
print(f'demo_df contains {demo_df.count()} records!')
display(demo_df.limit(6))
```

```
root
 |-- household_id: string (nullable = true)
 |-- household_size: integer (nullable = true)
 |-- num_adults: integer (nullable = true)
 |-- num_generations: integer (nullable = true)
 |-- adult_range: string (nullable = true)
 |-- marital_status: string (nullable = true)
 |-- race_code: string (nullable = true)
 |-- presence_children: string (nullable = true)
 |-- num_children: integer (nullable = true)
 |-- age_children: string (nullable = true)
 |-- age_range_children: string (nullable = true)
 |-- dwelling_type: string (nullable = true)
 |-- home_owner_status: string (nullable = true)
 |-- length_residence: integer (nullable = true)
 |-- home_market_value: string (nullable = true)
 |-- num_vehicles: integer (nullable = true)
 |-- vehicle_make: string (nullable = true)
 |-- vehicle_model: string (nullable = true)
 |-- vehicle_year: integer (nullable = true)
 |-- net_worth: integer (nullable = true)
 |-- income: string (nullable = true)
 |-- gender_individual: string (nullable = true)
 |-- age_individual: integer (nullable = true)
 |-- education_highest: string (nullable = true)
 |-- occupation_highest: string (nullable = true)
 |-- education_1: string (nullable = true)
 |-- occupation_1: string (nullable = true)
 |-- age_2: integer (nullable = true)
 |-- education_2: string (nullable = true)
 |-- occupation_2: string (nullable = true)
 |-- age_3: integer (nullable = true)
 |-- education_3: string (nullable = true)
 |-- occupation_3: string (nullable = true)
 |-- age_4: integer (nullable = true)
 |-- education_4: string (nullable = true)
 |-- occupation_4: string (nullable = true)
 |-- age_5: integer (nullable = true)
 |-- education_5: string (nullable = true)
 |-- occupation_5: string (nullable = true)
 |-- polit_party_regist: string (nullable = true)
 |-- polit_party_input: string (nullable = true)
 |-- household_clusters: string (nullable = true)
 |-- insurance_groups: string (nullable = true)
 |-- financial_groups: string (nullable = true)
 |-- green_living: string (nullable = true)

demo_df contains 357721 records!

CPU times: user 10.5 ms, sys: 8.15 ms, total: 18.6 ms
Wall time: 1.88 s
```

```
%%time
# daily_program data filename is 'Daily program data'
daily_prog_df = load_csv_file('Daily program data',
schemas_dict['Daily program data'])

daily_prog_df.printSchema()
print(f'daily_prog_df contains {daily_prog_df.count()} records!')
display(daily_prog_df.limit(6))
```

```
root
 |-- prog_code: string (nullable = true)
 |-- title: string (nullable = true)
 |-- genre: string (nullable = true)
 |-- air_date: string (nullable = true)
 |-- air_time: string (nullable = true)
 |-- Duration: float (nullable = true)

daily_prog_df contains 13194849 records!

CPU times: user 3.01 ms, sys: 19.4 ms, total: 22.4 ms
Wall time: 6.34 s
```

```
# Sample of 10 Million viewing entries

dataPath = f"dbfs:/viewing_10M"
viewing10m_df = spark.read.format("csv")\
    .option("header","true")\
    .option("delimiter",",")\
    .schema(schemas_dict['viewing_full'])\
    .load(dataPath)

display(viewing10m_df.limit(6))
print(f'viewing10m_df contains {viewing10m_df.count()} rows!')
```

```
viewing10m_df contains 10042340 rows!
```

```
%%time
# reference data is stored in parquet for your convinence.

ref_df = spark.read.parquet('dbfs:/refxml_new_parquet')
ref_df.printSchema()
print(f'ref_df contains {ref_df.count()} records!')
display(ref_df.limit(6))
```

```
root
 |-- device_id: string (nullable = true)
 |-- dma: string (nullable = true)
 |-- dma_code: long (nullable = true)
 |-- household_id: long (nullable = true)
 |-- household_type: string (nullable = true)
 |-- system_type: string (nullable = true)
 |-- zipcode: long (nullable = true)
```

```
ref_df contains 1268071 records!
```

```
CPU times: user 6.1 ms, sys: 3.87 ms, total: 9.97 ms
Wall time: 770 ms
```

```
reference_data = ref_df\
    .dropDuplicates(['device_id'])\
        .dropna(subset=["device_id", "DMA"])\
            .select('device_id', 'DMA')

viewing_data = viewing10m_df\
        .dropna(subset=["device_id", "prog_code"])\
            .select(["device_id", "prog_code"])

daily_data = daily_prog_df\
        .dropna(subset=["genre", "prog_code"])\
            .select(["genre", "prog_code"])

viewing_with_dma = viewing_data\
    .join(reference_data, "device_id")\
        .select("DMA", "device_id", "prog_code")
```

2.2

```python
from pyspark.sql.functions import col, avg, max as max_, lit, when
from pyspark.sql.window import Window
from pyspark.sql import Row

# Remove duplicates and nulls from ref_df and demo_df
ref_df_clean = ref_df.dropDuplicates().dropna(subset=["household_id",
"DMA"]).select('household_id', 'DMA')
demo_df_clean =
demo_df.dropDuplicates().dropna(subset=["household_id", "net_worth",
"income"]).select('household_id', 'net_worth', 'income').withColumn(
    "income",
    when(col('income') == 'A', 10)
    .when(col('income') == 'B', 11)
    .when(col('income') == 'C', 12)
    .when(col('income') == 'D', 13)
    .otherwise(col('income').cast('int'))
)

# Ensure income is correctly typed
demo_df_clean = demo_df_clean.withColumn("income",
col("income").cast("float"))

# Join demo_df with ref_df to get the DMA for each household
data = demo_df_clean.join(ref_df_clean, "household_id")

# Calculate the maximum net worth and maximum income across all data
max_net_worth_all_data = data.select(max_("net_worth")).collect()[0]
[0]
max_income_all_data = data.select(max_("income")).collect()[0][0]
print(max_net_worth_all_data)
print(max_income_all_data)

# Broadcast the maximum values to ensure they are used efficiently
max_net_worth_all_data_bc =
spark.sparkContext.broadcast(max_net_worth_all_data)
max_income_all_data_bc =
spark.sparkContext.broadcast(max_income_all_data)

# Calculate average net worth and average income for each DMA
dma_averages = data.groupBy("DMA").agg(
    (avg("net_worth") /
max_net_worth_all_data_bc.value).alias("avg_net_worth_in_dma"),
    (avg("income") /
max_income_all_data_bc.value).alias("avg_income_in_dma")
)

# Calculate the wealth score for each DMA
wealth_score = dma_averages.withColumn(
    "wealth_score",
    col("avg_net_worth_in_dma") + col("avg_income_in_dma")
)

# Get the top 10 wealthiest DMAs
top_10_dmas =
wealth_score.orderBy(col("wealth_score").desc()).limit(10).select("DMA
", "wealth_score")
```

```
9
13.0
```

```
wealth_score.orderBy(col("wealth_score").desc()).display()

top_10_dmas.display()

# Clean DataFrames and select relevant attributes (excluding
event_date and event_time)
all_viewings = (viewing10m_df
                .dropDuplicates()
                .dropna(subset=["device_id", "prog_code"])
                .select("device_id", "prog_code"))

top_10_dma_with_genres = (top_10_dmas
                          .join(reference_data, "DMA")
                          .join(all_viewings, "device_id")
                          .join(daily_data, "prog_code")
                          .dropDuplicates()
                          .dropna(subset=["DMA", "genre"])
                          .select("DMA", "genre"))

# Explode genres into individual rows
top_10_dma_with_splitted_genres = (top_10_dma_with_genres
                                   .withColumn('genre',
f.explode(f.split(f.col('genre'), ',')))
                                  )

dma_results, used_genres = [], []
check = top_10_dma_with_splitted_genres.select("*")
#for current_dma in top_dmas_list:
for dma_row in top_10_dmas.rdd.toLocalIterator():
    dma = dma_row["DMA"]
    current_dma_genres = check\
        .filter(\
                (check.DMA == dma)\
                & \
                (~check.genre.isin(used_genres)))\
                .groupBy("genre")\
                    .count()\
                        .orderBy("count",ascending=False)\
                            .select("genre")\
                                .limit(11)

    current_dma_genres_list = current_dma_genres\
        .rdd.flatMap(lambda x: x)\
            .collect()

    for used_genre in current_dma_genres_list:
        used_genres.append(used_genre)

    if len(used_genres) > 110:
        break

    dma_results.append((dma, current_dma_genres))


# Define the schema explicitly
schema = StructType([
    StructField("DMA NAME", StringType(), nullable=False),
    StructField("WEALTH SCORE", FloatType(), nullable=True),
    StructField("ORDERED LIST OF GENRES", StringType(),
nullable=False)
])

for i, (dma, genres) in enumerate(dma_results):
    # Get the wealth score for the current DMA
    wealth_score_row = top_10_dmas.filter(col("DMA") ==
dma).select("wealth_score").collect()
    if wealth_score_row:
        wealth_score = wealth_score_row[0]["wealth_score"]
    else:
        wealth_score = None  # Handle case where DMA is not found

    # Check if genres is empty and handle accordingly
    if not genres:
        print(f"No genres found for DMA: {dma}")
        continue

    # Create a list of Rows
    rows = [Row(dma, wealth_score, str(genre["genre"])) for genre in
genres.collect()]


    # Create a DataFrame from the list of Rows
    temp_df = spark.createDataFrame(rows, schema=schema)

    file_name = f"project1_part22_{dma.replace(' ',
'_').lower()}_337604821_326922390.csv"
    temp_df.write.format("csv").option("header",
"true").mode("overwrite").save(file_name)

    # Displaying the DF for the 1, 5, 9th most popular DMA by wealth
```

```
score
    if i in [0, 4, 8]:
        print(f"showing DMA - {dma}")
        temp_df.show()
```

```
showing DMA - San Antonio
+-----------+------------+---------------------+
|   DMA NAME|WEALTH SCORE|ORDERED LIST OF GENRES|
+-----------+------------+---------------------+
|San Antonio|   1.6239316|                 News|
|San Antonio|   1.6239316|               Sitcom|
|San Antonio|   1.6239316|              Weather|
|San Antonio|   1.6239316|                 Talk|
|San Antonio|   1.6239316|              Reality|
|San Antonio|   1.6239316|                 Auto|
|San Antonio|   1.6239316|              Cooking|
|San Antonio|   1.6239316|                Drama|
|San Antonio|   1.6239316|              Western|
|San Antonio|   1.6239316|               Comedy|
|San Antonio|   1.6239316|         Newsmagazine|
+-----------+------------+---------------------+

showing DMA - Bend, OR
+--------+------------+---------------------+
|DMA NAME|WEALTH SCORE|ORDERED LIST OF GENRES|
+--------+------------+---------------------+
|Bend, OR|    1.456719|             Outdoors|
|Bend, OR|    1.456719|        Bus./financial|
|Bend, OR|    1.456719|              History|
|Bend, OR|    1.456719|              Science|
|Bend, OR|    1.456719|               How-to|
|Bend, OR|    1.456719|              Animals|
|Bend, OR|    1.456719|       Playoff sports|
|Bend, OR|    1.456719|              Medical|
|Bend, OR|    1.456719|                 Golf|
|Bend, OR|    1.456719|               Nature|
|Bend, OR|    1.456719|           Paranormal|
+--------+------------+---------------------+

showing DMA - Seattle-Tacoma
+--------------+------------+---------------------+
|      DMA NAME|WEALTH SCORE|ORDERED LIST OF GENRES|
+--------------+------------+---------------------+
|Seattle-Tacoma|   1.4160091|              Auction|
|Seattle-Tacoma|   1.4160091|              Fishing|
|Seattle-Tacoma|   1.4160091|               Hockey|
|Seattle-Tacoma|   1.4160091|        Action sports|
|Seattle-Tacoma|   1.4160091|            Parenting|
|Seattle-Tacoma|   1.4160091|                Poker|
|Seattle-Tacoma|   1.4160091|             Aviation|
|Seattle-Tacoma|   1.4160091|            Card games|
|Seattle-Tacoma|   1.4160091|      Self improvement|
|Seattle-Tacoma|   1.4160091|                 Anime|
|Seattle-Tacoma|   1.4160091|           Environment|
+--------------+------------+----------------------+
```