```
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'])
# bonus points ?
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 3.28 s, sys: 67.3 ms, total: 3.35 s
Wall time: 23.4 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 3.77 s, sys: 61.5 ms, total: 3.83 s
Wall time: 11.2 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 1.29 s, sys: 3.19 ms, total: 1.29 s
Wall time: 6.46 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!
```

#Part 1 - Spam Detection

Query 1

```
def q1():
    # clean df
    cleaned_viewing_df = viewing10m_df\
        .dropDuplicates()\
            .dropna(subset=["device_id", "event_date", "prog_code"])\
                .select("device_id", "event_date", "prog_code")

    #For each device count number of it's views on each date
    #for each tuple of device and date, match num of views of this
device on this date
    daily_views_df = cleaned_viewing_df.groupBy("device_id",
"event_date")\
        .count()\
            .withColumnRenamed("count", "view_count")
```

```
    # for each device aggregate the number of views (at all dates
together)
    # and the number of days with views of this device
    device_stats_df = daily_views_df.groupBy("device_id").agg(
        f.sum("view_count").alias("total_views"),
        f.countDistinct("event_date").alias("active_days")
    )

    #given the daily average of the event get only the ones that have
a daily average > 5
    device_avg_views_df = device_stats_df.withColumn(
        "avg_daily_views", f.col("total_views") / f.col("active_days")
    ).filter(f.col("avg_daily_views") > 5)

    # get codes of bad programs according to query 1
    prog_codes_1 = cleaned_viewing_df.join(device_avg_views_df,
on="device_id", how="inner")\
        .select("prog_code").distinct()
    return prog_codes_1
```

Query 2

```
def q2():
    # clean df's and find DMA's with z or Z in their titles
    filtered_ref_df =
ref_df.dropDuplicates(["device_id"]).dropna(subset=["device_id",
"DMA"]).filter(f.col("DMA").rlike("[zZ]")).select('device_id', 'DMA')

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

    # Join df's
    joined_df = cleaned_viewing.alias("v").join(
        filtered_ref_df.alias("r"),
        f.col("v.device_id") == f.col("r.device_id"),
        "inner"
    )

    prog_codes_2 = joined_df.select("v.prog_code").distinct()
    return prog_codes_2


def q3():
    # cleaning df's
    filtered_ref_df =
ref_df.dropDuplicates(["device_id","household_id"]).dropna(subset=["de
vice_id",  "household_id"]).select('device_id', 'household_id')
```

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

    demo_df_clean = demo_df.dropDuplicates(["household_id",
"net_worth", "num_adults"]).dropna(subset=["household_id",
"net_worth", "num_adults"])\
        .select("household_id", "net_worth", "num_adults")

    joined_df = cleaned_viewing.join(filtered_ref_df, "device_id")

    # Apply filters such that only family with less than 3 adults and
networth that is greater than 8
    joined_with_demo = joined_df.join(demo_df_clean, "household_id")
    filtered_df = joined_with_demo.filter((f.col("num_adults") < 3) &
(f.col("net_worth") > 8))

    prog_codes_3 = filtered_df.select("prog_code").distinct()
    return prog_codes_3
```

Query 4

```
def q4():

    #clean df
    cleaned_viewing = daily_prog_df.withColumn("air_date",
f.dayofweek(f.to_date(daily_prog_df.air_date, "yyyyMMdd")))\
        .dropDuplicates(["air_date", "air_time",
"prog_code"]).dropna(subset=["air_date", "air_time",
"prog_code"]).select("air_date", "air_time", "prog_code")

    # Find shows that were aired within given date and time ranges
    prog_codes_4 = cleaned_viewing.filter(
        (
            (f.col("air_date") == 6) &  # 6 corresponds to Friday
            (f.col("air_time").cast("int") >= 180000)  # Between 6pm
to 11:59pm
        ) |
        (
            (f.col("air_date") == 7) &  # 7 corresponds to Saturday
            (f.col("air_time").cast("int") <= 190000) # Between 12am
to 7pm
        )
    ).select("prog_code").distinct()
    return prog_codes_4
```

Query 5

```
def q5():

    # clean df's
    filtered_ref_df =
ref_df.dropDuplicates(["device_id","household_id"]).dropna(subset=["de
vice_id",  "household_id"]).select('device_id', 'household_id')

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

    demo_df_clean = demo_df.dropDuplicates(["household_id",
"household_size"]).dropna(subset=["household_id",
"household_size"]).select("household_id", "household_size")
    joined_df = cleaned_viewing.join(filtered_ref_df, "device_id")

    joined_with_demo = joined_df.join(demo_df_clean, "household_id")

    # filter out so only families with a household > 8
    filtered_df = joined_with_demo.filter(f.col("household_size") >=
8)

    prog_codes_5 = filtered_df.select("prog_code").distinct()
    return prog_codes_5
```

Query 6

```
from pyspark.sql.functions import col, when, avg

def q6():

    # mapping letters to integer values
    demo_income_df = demo_df.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'))
    )

    # calculating average income
    income_avg = demo_income_df\
        .select(avg(col("income")).alias('avg_income'))\
            .collect()[0]['avg_income']

    # household and device are in ref_df
    # household and income are in demo_income_df
    # device_id and prog_code are in viewing10m_df
```

```
    # filtering out by query requirements

    bad_households = ref_df\
        .join(demo_income_df, "household_id")\
            .dropDuplicates(["device_id"])\
                .select('household_id', 'device_id', 'income')\
                    .filter(col("income") < income_avg)\
                        .select('household_id', 'device_id')\
                            .groupBy('household_id')\
                                .count()\
                                    .orderBy("count",
ascending=False)\
                                        .filter(col("count") > 3)\

    bad_devices = bad_households\
        .join(ref_df, on="household_id", how="inner")\
            .select('household_id', 'device_id')\
                .dropDuplicates(["device_id"])

    prog_codes_6 = bad_devices\
        .join(viewing10m_df, on="device_id", how="inner")\
            .dropDuplicates(["prog_code"])

    return prog_codes_6
```

Query 7

```
def q7():

    # cleaning df's
    prog_cleaned = daily_prog_df\
        .dropDuplicates(["prog_code"])\
            .dropna(subset=["prog_code", "genre"])\
                .select(["prog_code", "genre"])


    genres_to_check = ['Hydroplane racing', 'Biathlon', 'Snowmobile',
'Community', 'Agriculture', 'Music']

    # filtering according to requirements so that the program has at
least one
    # of the genres in the list above.
    prog_codes_7 = prog_cleaned\
        .withColumn('genre',f.explode(f.split('genre',',')))\
            .withColumn('isBadGenre',f.col('genre').isin(genres_to_che
ck))\
                .where(f.col('isBadGenre'))\
                    .dropDuplicates(['prog_code'])\
```

```
                        .select(["prog_code"])

    return prog_codes_7

# Create a distinct dataframe of prog_codes
prog_df = daily_prog_df.select(["prog_code"]).distinct().dropna()

# Transformations for each query
progs = [q1, q2, q3, q4, q5, q6, q7]

result_df = prog_df

for i, fun in enumerate(progs):
    name_col = 'prog_codes_' + str(i + 1)

    # Apply the transformation function to get a DataFrame with the
condition
    # such that 1 for that prog_code if in the transformation for the
matching query
    # otherwise 0
    df = fun()
    transformed_df = df.withColumn(name_col, f.lit(1))
    result_df = result_df.join(transformed_df, "prog_code", "left")

    result_df = result_df.fillna({name_col: 0})

# sum on each row to see for each record how many conditions it
fufills
sum_col = sum(f.col('prog_codes_' + str(i + 1)) for i in range(7))

check = result_df\
    .dropDuplicates(['prog_code'])\
        .withColumn("malicious", sum_col)
result = daily_prog_df.join(check, "prog_code", "inner")


for i in range(7):
    print("condition ", i+1, ":",
result.filter(f.col('prog_codes_'+str(i+1)) == 1).count())
```

```
condition  1 : 118503
condition  2 : 5743890
condition  3 : 6132426
condition  4 : 10499146
condition  5 : 4569284
condition  6 : 8645709
condition  7 : 1348644
```

```
# filter out to find the malicious records
malicious_records = result.filter(f.col("malicious") >= 4)\
  .select("prog_code", "malicious").dropDuplicates()
```

```
print(f"Total amount of malicious records:
{malicious_records.count()}")

malicious_records.orderBy(col("prog_code").asc()).show(50,
truncate=False)
```

```
Total amount of malicious records: 34108
+--------------+---------+
|prog_code     |malicious|
+--------------+---------+
|EP000000211576|4        |
|EP000000211639|4        |
|EP000000211645|5        |
|EP000000211646|5        |
|EP000000211647|4        |
|EP000000211648|5        |
|EP000000211649|4        |
|EP000000211650|4        |
|EP000000211654|4        |
|EP000000211659|4        |
|EP000000211661|4        |
|EP000000211662|4        |
|EP000000211665|4        |
|EP000000211666|4        |
|EP000000211667|4        |
|EP000000211669|4        |
|EP000000211670|4        |
|EP000000211672|5        |
|EP000000211676|5        |
|EP000000211679|4        |
|EP000000211680|5        |
|EP000000211681|4        |
|EP000000211682|4        |
|EP000000211683|4        |
|EP000000211684|4        |
|EP000000211685|5        |
|EP000000211686|4        |
|EP000000211688|4        |
|EP000000211689|4        |
|EP000000211690|5        |
|EP000000211691|4        |
|EP000000211692|5        |
|EP000000211694|5        |
|EP000000211696|4        |
|EP000000211698|4        |
|EP000000260097|4        |
|EP000000351218|4        |
|EP000000351219|4        |
|EP000000351223|4        |
```

```
|EP000000351224|4         |
|EP000000351225|4         |
|EP000000351228|4         |
|EP000000351230|4         |
|EP000000351235|4         |
|EP000000351240|4         |
|EP000000351247|4         |
|EP000000351250|4         |
|EP000000351251|4         |
|EP000000351254|4         |
|EP000000351255|4         |
+--------------+----------+
only showing top 50 rows


malicious_records.write.parquet("project1_part1_malicious_337604821_326922390")
```