Part 1 - Starter

In [0]:
```
from pyspark import SparkContext
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("my_project_1").getOrCreate()
```

Importing all spark data types and spark functions for your convenience.

In [0]:
```
from pyspark.sql.types import *
from pyspark.sql.functions import *
```

In [0]:
```
# Read a CSV into a dataframe
# There is a smarter version, that will first check if there is a Parquet fi
def load_csv_file(filename, schema):
  # Reads the relevant file from distributed file system using the given sch

  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 vali
    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 r
                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())
                  ])
}
```

Read demogrphic data

In [0]:

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

household_id	household_size	num_adults	num_generations	adu
00000015	2	2	1	0000000000001C
00000024	2	2	1	0000000001000C
00000026	null	null	null	0000000000000C
00000028	3	2	2	0000001100000C
00000035	1	1	1	0000000001000C
00000036	null	null	null	0000000000000C

```
CPU times: user 195 ms, sys: 23.7 ms, total: 219 ms
Wall time: 2.84 s
```

Read Daily program data

In [0]:

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

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!
```

prog_code	title	genre	air_date	air_time	Duration
EP000000250035	21 Jump Street	Crime drama	20151219	050000	60.0
EP000000250035	21 Jump Street	Crime drama	20151219	110000	60.0
EP000000250063	21 Jump Street	Crime drama	20151219	180000	60.0
EP000000510007	A Different World	Sitcom	20151219	100000	30.0
EP000000510008	A Different World	Sitcom	20151219	103000	30.0
EP000000510159	A Different World	Sitcom	20151219	080300	29.0

```
CPU times: user 50.6 ms, sys: 12.4 ms, total: 63 ms
Wall time: 9.19 s
```

Read viewing data

In [0]:
```
dataPath = "dbfs:/FileStore/ddm/10m_viewing"

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!')
```

mso_code	device_id	event_date	event_time	station_num	prog_co
01540	0000000050f3	20150222	193802	61812	EP00927978OC
01540	0000000050f3	20150222	195314	31709	EP02105643OC
01540	0000000050f3	20150222	200151	61812	EP00927978OC
01540	000000005518	20150222	111139	46784	EP00489137OC
01540	000000005518	20150222	190000	14771	EP01212407O1
01540	000000005518	20150222	200000	14771	EP01023732O1

```
viewing10m_df contains 9935852 rows!
```

Read reference data

Note that we removed the 'System Type' column.

In [0]:
```
# Read the new parquet
ref_data_schema = StructType([
    StructField('device_id', StringType()),
    StructField('dma', StringType()),
    StructField('dma_code', StringType()),
    StructField('household_id', IntegerType()),
    StructField('zipcode', IntegerType())
])

# Reading as a Parquet
dataPath = f"dbfs:/FileStore/ddm/ref_data"
ref_data = spark.read.format('parquet') \
                    .option("inferSchema","true")\
                    .load(dataPath)

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

device_id	dma	dma_code	household_id	zipcode
0000000050f3	Toledo	547	1471346	43609
000000006785	Amarillo	634	1924512	79119
000000007320	Lake Charles	643	3154808	70634
000000007df9	Lake Charles	643	1924566	70601
000000009595	Lexington	541	1600886	40601
000000009c6a	Houston	618	1924713	77339

```
ref_data contains 704172 rows!
```

Filtering useful data

In this step, we reduce each dataset by keeping only the necessary columns and filtering out invalid or duplicate rows.
This improves performance and simplifies the transformations required for the analysis in Part 1.

Reference Data (`clean_reference_df`)

We use this table to link `device_id` to `household_id`. Therefore:

- Rows with null values in either `device_id` or `household_id` are removed, as they cannot be used for reliable joins.
- Duplicate rows are removed to ensure that each `(device_id, household_id)` pair appears only once.

Demographic Data (`clean_demo_df`)

This data is used to evaluate household-level conditions. We:

- Remove any row where `household_id` is null, as such rows cannot be linked to a device.
- Drop duplicates to eliminate unnecessary repetition of demographic attributes.

Daily Program Data (`clean_daily_prog_df`)

We retain only rows with valid program codes:

- Rows with null `prog_code` are excluded, since the program cannot be identified or evaluated.
  (Note: in theory a null `prog_code` could still correspond to a malicious airing, but in practice, such cases are either nonexistent or untrackable.)

- We do not drop duplicates here, as repeated rows may represent different airings of the same program. Differences in other columns (e.g., timestamps) may justify their presence.

Viewing Data (`clean_viewing_df`)

This dataset connects program codes to devices. We:

- Remove rows where either `prog_code` or `device_id` is null, as they cannot be reliably joined.
- Drop duplicates to ensure a clean association between programs and devices.

This cleaning phase ensures that only valid, unique, and joinable records are kept for the core analysis.

In [0]:

```
# Reference Data
ref_data = ref_data.filter(col("household_id").isNotNull() & col("device_id"
clean_reference_df = ref_data.select("device_id", "household_id").dropDuplic

# Demographic Data
demo_df = demo_df.filter(col("household_id").isNotNull())
clean_demo_df = demo_df.select(
    "household_id",
    "vehicle_make",
    "income",
    "num_adults",
    "age_individual",
    "age_2"
).dropDuplicates()

# Daily Program Data
daily_prog_df = daily_prog_df.select(
    "prog_code",
    "Duration",
    "air_date",
    "air_time",
    "genre",
    "title"
)
clean_daily_prog_df = daily_prog_df.filter(col("prog_code").isNotNull())

# Program Viewing Data
viewing10m_df = viewing10m_df.filter(col("device_id").isNotNull() & col("pro
clean_viewing_df = viewing10m_df.select("prog_code", "device_id").dropDuplic


print(f'clean_reference_df contains {clean_reference_df.count()} rows!')
print(f'clean_demo_df contains {clean_demo_df.count()} rows!')
print(f'clean_daily_prog_df contains {clean_daily_prog_df.count()} rows!')
print(f'clean_viewing_df contains {clean_viewing_df.count()} rows!')
```

```
clean_reference_df contains 704172 rows!
clean_demo_df contains 357721 rows!
clean_daily_prog_df contains 13194849 rows!
clean_viewing_df contains 8436332 rows!
```

Applying the Brainwash Detection Conditions

Global Approach

To evaluate whether a program can be considered malicious, we define several conditions based on two types of data:

- **Program metadata** from the daily program table (`daily_prog_df`)
- **Household and demographic characteristics** from the demographic and reference data

Our approach proceeds in two phases:

1. Local Condition Flag Construction

We define individual boolean indicators (`condX_flag`) for conditions, using only the relevant table for that condition:

- Conditions based on program properties (e.g., duration, genre, airing date) are defined within the `daily_prog_df`.
- Conditions based on household information (e.g., number of devices, vehicle type, adult age gap) are defined within the `demo_df`, sometimes after joins with the reference data.

These `condX_flag` columns represent whether each row satisfies a given maliciousness criterion.

2. Aggregation and Combination

After constructing the condition flags:

- Program-level conditions are aggregated directly per `prog_code`.
- Household-level conditions are propagated to the programs viewed by those households, using the link between `device_id` (in viewing data) and `household_id` (in reference data). For each `prog_code`, we compute how many household-level conditions are satisfied at least once (We rely on the fact that for each prog_code, it is sufficient for one of the household-level conditions (2, 3, or 5) to be satisfied at least once by any of the associated households for the condition to be considered true for all these prog_code.)

Finally, we classify a progcode as *malicious* if its total conditions score is more than 4.

This modular and structured approach allows for flexibility, clarity, and scalability of condition definitions.

%md **Condition 2 – Vehicle Make is Toyota (`vehicle_make == "91"`)**
Condition 3 – Two Adults with Small Age Difference

We add two new columns to the `clean_demo_df` DataFrame: `cond2_flag` and `cond3_flag`.

- `cond2_flag` is used to identify households where the primary vehicle is a Toyota, encoded as `vehicle_make == "91"`.
  If the condition is satisfied, the flag is set to `1`; otherwise, it is set to `0`.

- `cond3_flag` marks households with exactly two adults (`num_adults == 2`) and a small age difference between them.
  Specifically, it checks that both `age_individual` and `age_2` are not null and their absolute difference is less than or equal to 6 years.
  When the condition is satisfied, the flag is set to `1`; otherwise, it is set to `0`.

In [0]:
```
# Add the cond3_flag column for condition 3
clean_demo_df = clean_demo_df.withColumn(
    "cond3_flag",
    when(
        (col("num_adults") == 2) &
        col("age_individual").isNotNull() &
        col("age_2").isNotNull() &
        (abs(col("age_individual") - col("age_2")) <= 6),
        1
    ).otherwise(0)
)

# Add the cond2_flag column for condition 2
clean_demo_df = clean_demo_df \
    .withColumn("cond2_flag", when((col("vehicle_make") == "91") & col("vehi
```

Condition 5: Households with More Than 3 Devices and Below-Average Income

To determine whether a program was watched by a household meeting this condition, we perform the following steps:

- Count the number of unique devices per household using the `clean_reference_df` table.
- Convert the household income codes (0–9, A–D) from the `clean_demo_df` table into numerical values, using a mapping where:

- Digits 0–9 are mapped to 0–9.
- Letters A–D are mapped to 10–13.

- Compute the average household income, **excluding null values**, to avoid skewing the result.
- Join the device count with the demographic data.
- Create the `cond5_flag` column, which is set to `1` **only if**:
  - The household has more than 3 devices.
  - The household's income is below the average.
  - The income field is not null.
- Print this average value and optionally display the result to assist with debugging or validating the calculation.

In [0]:
```
# Step 1. Count how many devices are associated with each household
device_counts = clean_reference_df.groupBy("household_id").agg(
    count("device_id").alias("num_devices")
)

# Step 2. Map income categories to numeric values (0–9 for digits, 10–13 for
income_mapping = {
    "0": 0, "1": 1, "2": 2, "3": 3, "4": 4,
    "5": 5, "6": 6, "7": 7, "8": 8, "9": 9,
    "A": 10, "B": 11, "C": 12, "D": 13
}
income_map_expr = create_map([lit(k) for pair in income_mapping.items() for

# Step 3. Add income_numeric column using the mapping
clean_demo_df = clean_demo_df.withColumn(
    "income_numeric",
    income_map_expr[col("income")]
)

# Step 4. Calculate the average income using only non-null values
avg_income = clean_demo_df.filter(col("income_numeric").isNotNull()) \
                                .select(avg("income_numeric").alias("avg_inc
                                .first()["avg_income"]

# Step 5. Join the device count data to the demographic dataframe
# Cast household_id to int to match the type of device_counts
clean_demo_df = clean_demo_df.withColumn(
    "household_id",
    col("household_id").cast("int")
)
clean_demo_df = clean_demo_df.join(device_counts, on="household_id", how="le

# Step 6. Add cond5_flag = 1 if household has more than 3 devices and income
clean_demo_df = clean_demo_df.withColumn(
    "cond5_flag",
    when(
        (col("num_devices").isNotNull()) &
        (col("num_devices") > 3) &
        (col("income_numeric").isNotNull()) &
        (col("income_numeric") < avg_income),
```

```
        1
    ).otherwise(0)
)
```

In [0]:
```
print(f"Average income: {avg_income}")
display(clean_demo_df.limit(20))
```

```
Average income: 6.715162771873656
```

household_id	vehicle_make	income	num_adults	age_individual	age_2	con
40	null	5	2	68	null	
111	null	4	2	72	68	
26	null	null	null	null	null	
117	null	null	null	null	null	
85	null	null	1	74	null	
99	null	null	null	null	null	
15	null	4	2	60	null	
61	null	8	2	68	58	
48	null	null	null	null	null	
126	37	null	2	60	86	

Condition 4 – Detecting Programs Aired on Friday the 13th

This block identifies programs that aired on a Friday the 13th or started on Thursday the 12th and ended on Friday the 13th. These are flagged under condition 4.

We start by creating a copy version of `clean_daily_prog_df` for creating timestamps, keeping only rows where `air_time`, `air_date`, and `Duration` are not null. Then, we construct a timestamp `start_datetime` by concatenating `air_date` (in `yyyyMMdd` format) with `air_time` (in `HHmmss` format) and parsing it accordingly.

To compute the program's end time, we add the `Duration` expressed in seconds to `start_datetime`. This is safe because although `Duration` is stored as a float, all values are actually integers (we verified). Thus, no rounding or overflow issues are expected.

From both `start_datetime` and `end_datetime`, we extract the calendar day and weekday name. We then define two conditions because it was verified that the maximum duration in the dataset is around 20 hours:

- One for programs starting on Friday the 13th.
- Another for programs starting on Thursday the 12th and ending on Friday the 13th.

We flag all rows satisfying at least one of these conditions by setting a new column `is_f13` to 1. We extract the distinct `prog_code`s that satisfy this and attach a `cond4_flag = 1` to them.

Finally, we left join this flagged subset with the original `clean_daily_prog_df` on `prog_code` and replace any missing values in `cond4_flag` (i.e., programs that did not meet the condition) with 0. This process ensures that the `cond4_flag` is accurately and safely incorporated into the main dataset.

We added two display calls on timestamp_clean_daily_prog_df:one to show rows from Friday the 13th, and another with arbitrary rows, in order to visualize that the condition is well applied.

In [0]:
```
# Convert Duration to Integer type since all values were verified to be whol
# Using integers avoids floating-point precision issues and simplifies furth
clean_daily_prog_df = clean_daily_prog_df.withColumn(
    "Duration",
    col("Duration").cast("int")
)
max_duration = clean_daily_prog_df.select(max("Duration").alias("max_duratio
                                  .first()["max_duration"]
print(f"The maximum duration in the dataset is: {max_duration} minutes")
```

```
The maximum duration in the dataset is: 1210 minutes
```

In [0]:
```
# Step 1: Create a clean dataframe to compute timestamp
timestamp_clean_daily_prog_df = clean_daily_prog_df.filter(
    col("air_time").isNotNull() &
    col("air_date").isNotNull() &
    col("Duration").isNotNull()
)

# Step 2: Create start datetime from air_date and air_time
timestamp_clean_daily_prog_df = timestamp_clean_daily_prog_df.withColumn(
    "start_datetime",
    to_timestamp(
        concat_ws("", col("air_date"), col("air_time")),
        "yyyyMMddHHmmss"
    )
)

# Step 3: Compute end datetime using Duration
timestamp_clean_daily_prog_df = timestamp_clean_daily_prog_df.withColumn(
    "end_datetime",
    (col("start_datetime").cast("long") + col("Duration") * 60).cast("timest
)

# Step 4: Extract day and weekday info from start and end timestamps
timestamp_clean_daily_prog_df = timestamp_clean_daily_prog_df \
    .withColumn("start_day", date_format("start_datetime", "d")) \
    .withColumn("start_weekday", date_format("start_datetime", "E")) \
    .withColumn("end_day", date_format("end_datetime", "d")) \
    .withColumn("end_weekday", date_format("end_datetime", "E"))
```

```
# Step 5: Define Friday 13th conditions
# Starts on Friday the 13th
cond1 = (col("start_day") == "13") & (col("start_weekday") == "Fri")
# Starts Thursday the 12th, ends Friday the 13th
cond2 = (
    (col("start_day") == "12") & (col("start_weekday") == "Thu") &
    (col("end_day") == "13") & (col("end_weekday") == "Fri")
)

# Step 6: Flag rows in clean df that satisfy Friday the 13th condition
timestamp_clean_daily_prog_df = timestamp_clean_daily_prog_df.withColumn(
    "is_f13",
    when(cond1 | cond2, 1).otherwise(0)
)

# Step 7: Get list of program codes that were aired at least once on a Frida
f13_progcodes = timestamp_clean_daily_prog_df.filter(col("is_f13") == 1).sel

# Step 8: Add cond4_flag = 1 to that list
f13_progcodes = f13_progcodes.withColumn("cond4_flag", lit(1))

# Step 9: Join back to original dataframe on prog_code
clean_daily_prog_df = clean_daily_prog_df.join(
    f13_progcodes,
    on="prog_code",
    how="left"
)

# Step 10: Replace nulls (for non-matching prog_codes) with 0
clean_daily_prog_df = clean_daily_prog_df.withColumn(
    "cond4_flag",
    when(col("cond4_flag").isNull(), 0).otherwise(col("cond4_flag"))
)
```

In [0]:
```
display(timestamp_clean_daily_prog_df.filter(col("is_f13") == 1).limit(5))
display(timestamp_clean_daily_prog_df.limit(5))
```

prog_code	Duration	air_date	air_time	genre	title	sta
EP000365112742	90	20151113	233000	Sports event,Volleyball	High School Volleyball	1
EP000369550133	31	20151113	233000	Sitcom	Martin	1
EP002309632678	60	20151113	233000	Reality,Law	The People's Court	1
EP004941440430	120	20151113	230000	Music	Top 20 Country Countdown	1
EP005544725036	180	20151113	233000	Sports event,Golf	PGA Tour Golf	1

prog_code	Duration	air_date	air_time	genre	title	start_datetim
EP000000250035	60	20151219	050000	Crime drama	21 Jump Street	2015-12-19T05:00:00
EP000000250035	60	20151219	110000	Crime drama	21 Jump Street	2015-12-19T11:00:00
EP000000250063	60	20151219	180000	Crime drama	21 Jump Street	2015-12-19T18:00:00
EP000000510007	30	20151219	100000	Sitcom	A Different World	2015-12-19T10:00:00
EP000000510008	30	20151219	103000	Sitcom	A Different World	2015-12-19T10:30:00

Constructing count from conditions (1, 4, 6, 7) in clean_daily_prog_df

We define a column `cond_count_dailyprog` that accumulates the number of local suspicious conditions satisfied by each program. This is done efficiently in a single `withColumn` statement, without introducing intermediate flag columns.

The conditions evaluated are:

- Condition 1: The program's `Duration` exceeds the average duration of all programs (previously computed).
- Condition 4: The program is flagged with `cond4_flag = 1` if it was aired on a Friday the 13th (computed earlier).
- Condition 6: The `genre` field contains at least one **case-sensitive exact match** of the following terms:
  `['Collectibles', 'Art', 'Snowmobile', 'Public affairs', 'Animated', 'Music']`.
  This is evaluated using a regular expression with `\b` to ensure full-word boundaries.
- Condition 7: The `title` contains at least **two** of the following words: `['better', 'girls', 'the', 'call']`, matched **case-insensitively** and **as full words only**.

Details on **Condition 7** implementation:

- For each word, we use a regex expression like `rlike("(?i)\bword\b")` to ensure:
  - **(?i)**: case-insensitive matching
  - **\b**: word boundary to avoid partial matches (e.g. “calling” won't match “call”)
- Each match returns a boolean, which we cast to `int` (`1` if matched, `0` otherwise).

- The sum of these 4 booleans is compared to `2` or more to determine satisfaction of this condition.

This logic ensures we precisely and compactly identify suspicious programs without polluting the dataframe with temporary columns.

To better understand and validate the condition logic, we added three display() calls: one showing the top 5 rows with the highest cond_count_dailyprog, one showing 5 rows matching Condition 6 (malicious genres), and one showing 5 rows matching Condition 7 (titles with at least two malicious words).

In [0]:
```
# Step 1: Compute average duration
avg_duration = clean_daily_prog_df.filter(col("Duration").isNotNull()) \
                                  .select(avg("Duration").alias("avg_duratic
                                  .first()["avg_duration"]

# Step 2: Define genre and title malicious sets
malicious_genres = ['Collectibles', 'Art', 'Snowmobile', 'Public affairs', '
malicious_words = ['better', 'girls', 'the', 'call']

# Step 3: Create cond_count directly without intermediate condX_flag columns
clean_daily_prog_df = clean_daily_prog_df.withColumn(
    "cond_count_dailyprog",
    # Condition 1: Duration > average
    when(col("Duration") > avg_duration, 1).otherwise(0)
    +
    # Condition 4: already in cond4_flag
    when(col("cond4_flag") == 1, 1).otherwise(0)
    +
    # Condition 6: Genre contains a malicious genre (case-sensitive)
    when(
    col("genre").rlike(r"\bCollectibles\b|\bArt\b|\bSnowmobile\b|\bPublic af
    1
    ).otherwise(0)
    +
    # Condition 7: title contains at least 2 malicious words (case insensiti
    when(
        (
            col("title").rlike(r"(?i)\bbetter\b").cast("int") +
            col("title").rlike(r"(?i)\bgirls\b").cast("int") +
            col("title").rlike(r"(?i)\bthe\b").cast("int") +
            col("title").rlike(r"(?i)\bcall\b").cast("int")
        ) >= 2,
        1
    ).otherwise(0)
)
```

In [0]:
```
print(f"Average Duration: {avg_duration}")
print(f"----------------------------------------- BEST COUND COUNT---------
display(clean_daily_prog_df.drop("air_date", "air_time").orderBy(col("cond_c
print(f"----------------------------------------- COUND 6------------------
display(
    clean_daily_prog_df.drop("air_date", "air_time").filter(
```

```
        col("genre").rlike(r"\bCollectibles\b|\bArt\b|\bSnowmobile\b|\bPubli
    ).limit(5)
)
print(f"---------------------------------------- COUND 7------------------
display(
    clean_daily_prog_df.drop("air_date", "air_time").filter(
        (
            col("title").rlike(r"(?i)\bbetter\b").cast("int") +
            col("title").rlike(r"(?i)\bgirls\b").cast("int") +
            col("title").rlike(r"(?i)\bthe\b").cast("int") +
            col("title").rlike(r"(?i)\bcall\b").cast("int")
        ) >= 2
    ).limit(5)
)
```

```
Average Duration: 60.425457161351375
---------------------------------------- BEST COUND COUNT------------------
-------------------------
```

prog_code	Duration	genre	title	cond4_flag	cond_count_da
SH020449080000	360	Music	Today's Country: NASH	1	
SH006615910000	120	Special,Music	Magic Moments: The Best of 50s Pop	1	
SH006615910000	120	Special,Music	Magic Moments: The Best of 50s Pop	1	
SH020449080000	180	Music	Today's Country:	1	

```
---------------------------------------- COUND 6--------------------------
----------------
```

prog_code	Duration	genre	title	cond4_flag	cond_c
EP000005361169	30	How-to,Art	The Best of the Joy of Painting	0	
EP002971100037	30	Sitcom,Animated	The PJs	0	
EP002971100037	30	Sitcom,Animated	The PJs	0	
EP008051970156	60	Music	Song of the Mountains	1	
EP010604980340	30	Newsmagazine,Public affairs	Global 3000	0	

```
---------------------------------------- COUND 7--------------------------
----------------
```

prog_code	Duration	genre	title	cond4_flag	cond_count_dailyprog
EP000174760142	30	Sitcom	The Golden Girls	0	1
EP000174760142	30	Sitcom	The Golden Girls	0	1
EP000174760142	30	Sitcom	The Golden Girls	0	1
EP000174760142	30	Sitcom	The Golden Girls	0	1
EP015685700002	60	Drama	Call the Midwife	0	1

Combining Daily Program-Based and Demographic-Based Malicious Conditions

In this section, we determine how many of the demographic-related malicious conditions (specifically conditions 2, 3, and 5) are satisfied at least once for each program. The idea is not to count how many households satisfy a condition overall, but rather to evaluate for each program whether at least one household that viewed it satisfies each condition. This results in a per-program score indicating how many demographic risk signals are associated with it.

To achieve this, we first join the program viewing data (`clean_viewing_df`) with the household mapping (`clean_reference_df`) to identify which household watched each program. We then join this result with the demographic dataset (`clean_demo_df`) to enrich the records with household information such as vehicle make, number of adults, age difference, income, and number of devices.

The three conditions are represented by binary flags:

- `cond2_flag`: household owns a Toyota (vehicle make code "91")
- `cond3_flag`: household has exactly two adults with an age difference of 6 years or less
- `cond5_flag`: household owns more than 3 devices and has income below the average

These flags are aggregated per program using the max function to determine if each condition was satisfied at least once for each prog_code. This is appropriate because the flags are binary (0 or 1), so the max will be 1 if at least one household that watched the program met the condition — which is exactly the behavior we want. Alternatively, we could have used sum and checked whether the result is greater than 0. The sum of these three flags gives us a `cond_count_demo` column that counts how many of the three conditions were satisfied at least once by any household that viewed the program.

This demographic-based count is then joined back with the program metadata (`clean_daily_prog_df`). Programs that were not matched with any viewer household receive a default value of 0 for the condition count. The final malicious score per airing, `cond_count_total`, is computed by summing the demographic-based `cond_count_demo` with the previously calculated content-based `cond_count_dailyprog`.

Finally, we extract the columns of interest (`title` and `cond_count_total`) into a new DataFrame `count_df`, which will serve as the basis for malicious title detection in the next steps.

We also added a display(progcode_cond_df) to visually verify that the condition aggregation per program (prog_code) behaves as expected and correctly captures the presence of each demographic flag.

In [0]:

```
# Step 1: Join reference_df with viewing_df on 'device_id'
ref_view_df = clean_reference_df.join(
    clean_viewing_df,
    on="device_id",
    how="inner"
)

# Step 2: Join the result with clean_demo_df on 'household_id'
ref_view_demo_df = ref_view_df.join(
    clean_demo_df,
    on="household_id",
    how="inner"
)

# Step 3: Group by prog_code and aggregate each condition with max
# This checks if the condition happened at least once for the program
progcode_cond_df = ref_view_demo_df.groupBy("prog_code").agg(
    max("cond2_flag").alias("cond2_seen"),
    max("cond3_flag").alias("cond3_seen"),
    max("cond5_flag").alias("cond5_seen")
)

# Step 4: Count how many of the conditions were met at least once per prog_c
progcode_cond_df = progcode_cond_df.withColumn(
    "cond_count_demo",
    col("cond2_seen") + col("cond3_seen") + col("cond5_seen")
)
progcode_cond_demo_df = progcode_cond_df.select("prog_code", "cond_count_dem

# Step 5: Join this result with a copy of daily_program_data on 'prog_code'
full_daily_prog_df = clean_daily_prog_df.join(
    progcode_cond_demo_df,
```

```
    on="prog_code",
    how="left"
)

# Step 6: Replace null cond_count_demo with 0
full_daily_prog_df = full_daily_prog_df.withColumn(
    "cond_count_demo",
    when(col("cond_count_demo").isNull(), 0).otherwise(col("cond_count_demo"
)

# Step 7: Compute total condition count
full_daily_prog_df = full_daily_prog_df.withColumn(
    "cond_count_total",
    col("cond_count_demo") + col("cond_count_dailyprog")
)

# Step 8: Keep only the final columns of interest
count_df = full_daily_prog_df.select( "title", "cond_count_total")
```

In [0]:
```
display(progcode_cond_df.limit(20))
```

prog_code	cond2_seen	cond3_seen	cond5_seen	cond_count_demo
EP006819110182	1	1	1	3
MV002754530000	1	1	1	3
EP020959710005	1	1	1	3
EP007263640277	1	1	0	2
SH021195330000	1	1	1	3
SP003098600000	1	1	1	3
SH015840390000	1	1	1	3
SH015815970000	1	1	1	3
EP006609610456	1	1	1	3
EP000191552705	1	1	1	3

Final Step: Identify and Display Malicious Titles

In this final step, we identify "malicious" program titles based on the total number of malicious records associated with each title.

- We first added a new boolean column `is_malicious` to the `daily_prog_df`, where a program is considered malicious if it satisfies at least 4 out of the 7 predefined conditions.
- We then grouped the data by `title` to calculate both the total number of records and the number of malicious records per title.

- From this, we computed the `malicious_ratio` for each title, representing the proportion of its records labeled as malicious.
- We filtered the titles to keep only those for which **more than 40% of the records** are malicious (`malicious_ratio > 0.4`).
- Finally, we selected and displayed the **top 20 titles** with the highest malicious ratios, ordered in descending order as required .

In [0]:
```
# Step 1: Add boolean column to mark if record is malicious (cond_count_tota
malicious_df = count_df.withColumn(
    "is_malicious",
    when(col("cond_count_total") >= 4, 1).otherwise(0)
)

# Step 2: Group by title to get total records and malicious records
title_stats_df = malicious_df.groupBy("title").agg(
    count("*").alias("total_count"),
    sum("is_malicious").alias("malicious_count")
)

# Step 3: Compute ratio of malicious records
title_stats_df = title_stats_df.withColumn(
    "malicious_ratio",
    (col("malicious_count") / col("total_count"))
)

# Step 4: Filter titles where more than 40% of records are malicious
malicious_titles_df = title_stats_df.filter(
    col("malicious_ratio") > 0.4
)

# Final result: titles and their malicious percentage (top 20 by ratio)
malicious_titles_df = malicious_titles_df.select("title", "malicious_ratio")
                                         .orderBy(col("malicious_ratio").des

display(malicious_titles_df.limit(20))
```

title	malicious_ratio
Noticiero Telemundo KTMO	1.0
KING 5 News at 9	1.0
Battle of the Year	1.0
TV Star Confesses to Helping	1.0
She Hate Me	1.0
KSPR News at 11	1.0
Documentary	1.0
Arkansas Alive	1.0
WWE Raw En Español	1.0
(A)Sexual	1.0