Project 1 - Part 1

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 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 - 'bitw
ise'
                    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	adult_range	marital_status	race_code	presence_ch
00000015	2	2	1	0000000000000100000000	S	B	
00000024	2	2	1	0000000001000000000000	null	W	
00000026	null	null	null	0000000000000000000000	null	null	
00000028	3	2	2	0000001100000000000000	S	W	
00000035	1	1	1	0000000001000000000000	null	W	
00000036	null	null	null	0000000000000000000000	null	null	

```
CPU times: user 80.8 ms, sys: 10.1 ms, total: 91 ms
Wall time: 22.9 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 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!
```

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 13 ms, sys: 2.99 ms, total: 16 ms
Wall time: 9.34 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_code
01540	0000000050f3	20150222	193802	61812	EP009279780033
01540	0000000050f3	20150222	195314	31709	EP021056430002
01540	0000000050f3	20150222	200151	61812	EP009279780033
01540	000000005518	20150222	111139	46784	EP004891370013
01540	000000005518	20150222	190000	14771	EP012124070127
01540	000000005518	20150222	200000	14771	EP010237320166

```
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!
```

Part 1.1

Data preprocessing - adding helpful columns for easier implementation later. Filtering out unneeded columns.

In [0]:

```
# Filter records by value - drop duplicates
demo_df_new = demo_df.distinct()
daily_prog_df_new = daily_prog_df.distinct()
viewing10m_df_new = viewing10m_df.distinct()
ref_data_new = ref_data.distinct()


# Filter relevant columns only and cast household_id to have a matching type in different dataframes
demo_df_new = demo_df_new.select(['household_id', 'income', 'num_adults', 'age_individual', 'age_2','num_vehicles', 'vehicle_make']).withColumn("household_id", col("household_id").cast("int"))
daily_prog_df_new = daily_prog_df_new.select(['prog_code', 'title', 'genre', 'air_date', 'air_time', 'Duration'])
viewing10m_df_new = viewing10m_df_new.select(['prog_code','device_id', 'event_date', 'event_time'])
ref_data_new = ref_data_new.select(['device_id', 'household_id', 'dma', 'dma_code'])

# Casting A-D to 10-13
demo_df_new = demo_df_new.withColumn(
    "income",
    when(col("income") == "A", 10)
    .when(col("income") == "B", 11)
    .when(col("income") == "C", 12)
    .when(col("income") == "D", 13)
    .when(col("income").cast("int").isNotNull(), col("income").cast("int"))
)

# Extract averages for checking conditions later
avg_prog_duration_df = daily_prog_df_new.groupBy('prog_code').avg('Duration')
overall_avg_duration = avg_prog_duration_df.select(avg("avg(Duration)")).first()[0]
avg_household_income_df = demo_df_new.groupBy('household_id').avg('income')
avg_income = avg_household_income_df.select(avg("avg(income)")).first()[0]


# Columns we added:
# 1.daily_prog_df - day_of_week
# 2.demo_df - age_diff_if_two
```

```
# 3.ref_data - amount of distinct devices per household

#Convert 'air date' from string (YYYYMMDD) to DateType
daily_prog_df_new = daily_prog_df_new.withColumn('air_date',to_date(col('air_date'), 'yy
yyMMdd'))

#Add day of week
daily_prog_df_new = daily_prog_df_new.withColumn('day',dayofweek(col('air_date')))

#Add age difference for 2-adult households
demo_df_new = demo_df_new.withColumn(
    "age_diff_if_two",
    when(
        col("num_adults") == 2,
        abs(col("age_individual") - col("age_2"))
    ).otherwise(None)
)

#Add devices count column
device_counts = ref_data_new.groupBy('household_id').agg(countDistinct('device_id').alias
('device_count'))
ref_data_new = ref_data_new.join(device_counts, on='household_id', how='left')

# Displaying columns we added:
display(demo_df_new.limit(6))
display(daily_prog_df_new.limit(6))
display(ref_data_new.limit(6))
```

household_id	income	num_adults	age_individual	age_2	num_vehicles	vehicle_make	age_diff_if_two
36	null	null	null	null	null	null	null
24	7	2	46	null	null	null	null
35	null	1	50	null	null	null	null
15	4	2	60	null	null	null	null
26	null	null	null	null	null	null	null
28	7	2	38	34	null	null	4

prog_code	title	genre	air_date	air_time	Duration	day
EP000009760041	Coach	Sitcom	2015-12-19	190000	30.0	7
EP000012570105	The Dick Van Dyke Show	Sitcom	2015-12-19	223000	30.0	7
EP000024540029	Keeping Up Appearances	Sitcom	2015-12-20	080000	30.0	1
EP000028000206	Market to Market	Newsmagazine	2015-12-20	130000	30.0	1
EP000029601399	MotorWeek	How-to,Auto,Consumer	2015-12-19	223000	30.0	7
EP000037900771	Sewing With Nancy	Educational,How-to	2015-12-20	000000	30.0	1

household_id	device_id	dma	dma_code	device_count
1963765	0000003c83aa	Little Rock-Pine Bluff	693	2
1969296	000000428da4	Alexandria, LA	644	2
1969958	00000042d861	Tulsa	671	1
1314105	00000043ab9d	Sioux Falls(Mitchell)	725	1
1971832	00000043bc7d	Shreveport	612	2
1517615	0000005d6d7a	Toledo	547	4

Part 1.2:

An airing is considered malicious if it satisfies at least 4 out of 7 conditions. We observed that 4 conditions are based on the program (prog_code), while the remaining 3 are based on household attributes.

This insight allows us to filter the dataset: any program that does not meet any of the 4 program-dependent conditions can be dropped before evaluating household-related conditions, improving performance.

conditions can be dropped before evaluating household-related conditions, improving performance.

In [0]:

```
# Checking prog_code dependent conditions
# Condition 1
daily_prog_df_new = daily_prog_df_new.withColumn("cond_1", col("duration") > overall_avg
_duration)
```

For Condition 4 we used the help of ChatGPT to implement a check that verifies if any portion of the program aired during Friday 13th (instead of just checking if it started on one).

In [0]:

```
# Condition 4

# help from chat GPT - making sure that every program that was aired in friday 13th in *a
ny* part of it will be marked as true
# help with adding helper columns: air_datetime, end_datetime, end_time, end_date, day_of
_week_of_end_date, date_range, array_of_all_dates.

# Step 1: Create full datetime from air_date (DateType) and air_time (HHMMSS string)
# This combines the date and time into a proper timestamp
daily_prog_df_new = daily_prog_df_new.withColumn(
    "air_datetime",
    to_timestamp(expr(
        "concat(date_format(air_date, 'yyyy-MM-dd'), ' ', " +
        "substr(air_time, 1, 2), ':', substr(air_time, 3, 2), ':', substr(air_time, 5, 2)
)"
    ))
)

# Step 2: Add duration (in seconds) to air_datetime to compute end_datetime
daily_prog_df_new = daily_prog_df_new.withColumn(
    "end_datetime",
    expr("from_unixtime(unix_timestamp(air_datetime) + int(Duration * 60))")
)

# Step 3: Extract end_time (as HHMMSS string) and end_date (as DateType)
daily_prog_df_new = daily_prog_df_new.withColumn(
    "end_time",
    date_format(col("end_datetime"), "HHmmss")
).withColumn(
    "end_date",
    date_format(col("end_datetime"), "yyyy-MM-dd").cast("date")
)

# Step 4: Compute day of week for the end_date
daily_prog_df_new = daily_prog_df_new.withColumn(
    "day_of_week_of_end_date",
    dayofweek(col("end_date"))
)

# Step 5: Generate a sequence of dates between air_date and end_date
daily_prog_df_new = daily_prog_df_new.withColumn(
    "date_range",
    sequence(col("air_date"), col("end_date"))
)

# Step 6: Convert each date in the range to a tuple (day of month, day of week)
daily_prog_df_new = daily_prog_df_new.withColumn(
    "array_of_all_dates",
    transform(
        col("date_range"),
        lambda d: struct(dayofmonth(d).alias("day"), dayofweek(d).alias("weekday"))
    )
)

# Step 7: Identify prog_codes where day=13, weekday=6 (which means Friday the 13th exists
in the date array)
friday_13_prog_codes = daily_prog_df_new.filter(
```

```
    expr("exists(array_of_all_dates, x -> x.day = 13 AND x.weekday = 6)")
).select("prog_code").distinct()

# Step 8: Mark those prog_codes with cond_4 = True in the original DataFrame, else False
daily_prog_df_new = daily_prog_df_new.join(
    friday_13_prog_codes.withColumn("cond_4", lit(True)),
    on="prog_code",
    how="left"
).withColumn(
    "cond_4",
    col("cond_4").isNotNull()
)
```

We can see in the table displayed below the columns added for condition 4 and how they have been used.

In [0]:

```
# Printing all relavent and helper columns for condition 4 so we can see how it was done
display(daily_prog_df_new.select(
    "prog_code",
    "Duration",
    "air_datetime",
    "end_datetime",
    "end_time",
    "end_date",
    "day_of_week_of_end_date",
    "date_range",
    "array_of_all_dates","cond_4"
).limit(10))

# Droping helper columns
daily_prog_df_new = daily_prog_df_new.drop(
    "air_datetime",
    "end_datetime",
    "end_time",
    "end_date",
    "day_of_week_of_end_date",
    "date_range",
    "array_of_all_dates"
)
```

prog_code	Duration	air_datetime	end_datetime	end_time	end_date	day_of_week_of_end_date	date_range	array_
EP000009760041	30.0	2015-12-19T19:00:00Z	2015-12-19 19:30:00	193000	2015-12-19	7	List(2015-12-19)	Lis
EP000012570105	30.0	2015-12-19T22:30:00Z	2015-12-19 23:00:00	230000	2015-12-19	7	List(2015-12-19)	Lis
EP000024540029	30.0	2015-12-20T08:00:00Z	2015-12-20 08:30:00	083000	2015-12-20	1	List(2015-12-20)	Lis
EP000028000206	30.0	2015-12-20T13:00:00Z	2015-12-20 13:30:00	133000	2015-12-20	1	List(2015-12-20)	Lis
EP000029601399	30.0	2015-12-19T22:30:00Z	2015-12-19 23:00:00	230000	2015-12-19	7	List(2015-12-19)	Lis
EP000037900771	30.0	2015-12-20T00:00:00Z	2015-12-20 00:30:00	003000	2015-12-20	1	List(2015-12-20)	Lis

moving on to conditons 6+7 and filtering:

In [0]:

```
# Condition 6 - program contains at least one of the genres in list. (case sensitive)
daily_prog_df_new = daily_prog_df_new.withColumn("genre_array", split(col("genre"), ",")
)
daily_prog_df_new = daily_prog_df_new.withColumn(
    "cond_6",
    (array_contains(col("genre_array"), "Collectibles") |
```

```
    array_contains(col("genre_array"), "Art") |
    array_contains(col("genre_array"), "Snowmobile") |
    array_contains(col("genre_array"), "Public affairs") |
    array_contains(col("genre_array"), "Animated") |
    array_contains(col("genre_array"), "Music")) &
    col("genre").isNotNull()
)

# Condition 7 -  programs with titles containing at least two of the words in the list (c
ase insensitive)
bad_words = ['better', 'girls', 'the', 'call']
daily_prog_df_new = daily_prog_df_new.withColumn("title_lower", lower(col("title")))
bw1 = col("title_lower").contains("better").cast("int")
bw2 = col("title_lower").contains("girls").cast("int")
bw3 = col("title_lower").contains("the").cast("int")
bw4 = col("title_lower").contains("call").cast("int")
bad_words_count = bw1 + bw2 + bw3 + bw4
daily_prog_df_new = daily_prog_df_new.withColumn( "cond_7", bad_words_count >= 2).drop('
title_lower', 'genre_array')


#Filter out programs that fail all program-dependent conditions
daily_prog_df_new = daily_prog_df_new.filter(
    col("cond_1") | col("cond_4") | col("cond_6") | col("cond_7")
)
```

We can see below that the filtering by the first 4 cconditions reduced significantly the size of the table(6,542,562 instead of 13,194,849 records!). now we can determine the household-dependent conditions more efficiently. we can also see the amount of airings matching each condition.

In [0]:

```
#Printing how many rows are left after filtering
print(f'daily prog now contains {daily_prog_df_new.count()} rows!')

#Displaying the amount of airings matching each condition
daily_prog_df_new.select(
    sum(col("cond_1").cast("int")).alias("sum_cond_1"),
    sum(col("cond_4").cast("int")).alias("sum_cond_4"),
    sum(col("cond_6").cast("int")).alias("sum_cond_6"),
    sum(col("cond_7").cast("int")).alias("sum_cond_7")
).show()
```

```
daily prog now contains 6542562 rows!
+----------+----------+----------+----------+
|sum_cond_1|sum_cond_4|sum_cond_6|sum_cond_7|
+----------+----------+----------+----------+
|   3718572|   4150198|   1325663|     15088|
+----------+----------+----------+----------+
```

Now with the filtered data we got, we move on to the next 3 conditions:

In condition 5 - help from chatGPT with joining - (broadcasting ,adding column 'match_prog_code')

In [0]:

```
#Condition 2
# Identify households with a Toyota (vehicle_make == '91')
toyota_households = demo_df_new.filter(col("vehicle_make") == "91").select("household_id
").distinct()

#Map devices to these households
device_household_df = ref_data_new.select("device_id", "household_id")
view_with_household = viewing10m_df_new.join(device_household_df, on="device_id")

#Extract unique program codes watched by those households and set cond2 = True for them
toyota_prog_codes = view_with_household.join(toyota_households, on="household_id").selec
t("prog_code").distinct()
```

```
daily_prog_df_new = daily_prog_df_new.join(
    toyota_prog_codes.withColumn("cond_2", lit(True)),
    on="prog_code",
    how="left"
).withColumn("cond_2", when(col("cond_2").isNull(), False).otherwise(True))


# Condition 3
#Find devices of households matching the condition
cond3_households = demo_df_new.filter(col("age_diff_if_two").isNotNull() & (col("age_dif
f_if_two") <= 6)).select("household_id").distinct()
cond3_devices = ref_data_new.join(cond3_households, on="household_id", how = "inner").se
lect("device_id").distinct()

#Find programs watched by these devices and set cond3 = True for them
cond3_prog_codes = viewing10m_df_new.join(cond3_devices, on="device_id", how="inner").se
lect("prog_code").distinct().withColumn("cond_3", lit(True))
daily_prog_df_new = daily_prog_df_new.join(cond3_prog_codes, on="prog_code", how="left")
.fillna({"cond_3": False}).fillna({"cond_3": False})


# Condition 5
#Join relevant tables and filter according to the condition
joined_df = viewing10m_df_new.join(ref_data_new, on='device_id', how='inner') \
                             .join(demo_df_new, on='household_id', how='inner')
joined_df = joined_df.filter((col('device_count') > 3) & (col('income') < avg_income))

# Find relevant prog_codes for cond 5
relevant_prog_codes_cond5 = joined_df.select('prog_code').distinct()

# Set cond5 = True for them relevant prog_codes - help from chatGPT with joining - (broad
casting ,adding column 'match_prog_code')
daily_prog_df_new = daily_prog_df_new.join(
    broadcast(relevant_prog_codes_cond5.withColumnRenamed('prog_code', 'match_prog_code'
)),
    daily_prog_df_new['prog_code'] == col('match_prog_code'),
    how='left'
)
daily_prog_df_new = daily_prog_df_new.withColumn(
    'cond_5',
    when(col('match_prog_code').isNotNull(), True).otherwise(False)
).drop('match_prog_code')
```

We can see below how the dataframe looks like with a column marking true or false for each condition. We can also see the amount of airings matching each of the last 3 conditions we added.

In [0]:

```
#Print amounts of rows that matched each condition and display the dataframe with the con
dition columns
daily_prog_df_new.select(
    sum(col("cond_2").cast("int")).alias("sum_cond_2"),
    sum(col("cond_3").cast("int")).alias("sum_cond_3"),
    sum(col("cond_5").cast("int")).alias("sum_cond_5")
).show()

display(daily_prog_df_new.limit(20))
```

```
+----------+----------+----------+
|sum_cond_2|sum_cond_3|sum_cond_5|
+----------+----------+----------+
|   2979479|   3484728|   3065708|
+----------+----------+----------+
```

prog_code	title	genre	air_date	air_time	Duration	day	cond_1	cond_4	co
EP000029340035	Mod Squad	Crime drama	2015-08-16	060000	60.0	1	true	false	
EP000174760038	The Golden Girls	Sitcom	2015-08-17	033000	30.0	2	false	false	

prog_code	title	genre	air_date	air_time	Duration	day	cond_1	cond_4	co
EP002954050038	Crashbox	Children,Game show	2015-08-16	173000	30.0	1	false	true	
EP003034830150	Futurama	Sitcom,Science fiction,Animated	2015-08-16	033000	30.0	1	false	false	
EP003077660005	SpongeBob SquarePants	Children,Comedy,Fantasy,Animated	2015-08-17	050000	30.0	2	false	false	
EP003954600859	Cheaters	Reality	2015-08-16	230000	60.0	1	true	false	

Now we perform the final filtering based on condition counts and passing ratios for each title:

In [0]:

```
# Final Filtering Based on Condition Count
# Count how many conditions (cond_1 to cond_7) each row satisfies, then keep titles with
> 40% pass rate


# Count how many of the 7 conditions are True for each row
daily_prog_df_new = daily_prog_df_new.withColumn(
    "cond_count",
    col("cond_1").cast("int") +
    col("cond_2").cast("int") +
    col("cond_3").cast("int") +
    col("cond_4").cast("int") +
    col("cond_5").cast("int") +
    col("cond_6").cast("int") +
    col("cond_7").cast("int")
)

# Mark whether a row passed (4 or more conditions are met)
daily_prog_df_new = daily_prog_df_new.withColumn("cond_pass", col("cond_count") >= 4)

# Calculate the pass ratio per title (what % of its rows passed)
title_pass_ratio = daily_prog_df_new.groupBy("title").agg(
    avg(col("cond_pass").cast("double")).alias("pass_ratio")
)

# Keep only titles where more than 40% of rows passed
malicious_titles = title_pass_ratio.filter(col("pass_ratio") > 0.4).select("title", "pas
s_ratio")

# Final result: 20 titles with top pass ratio
Top_20_titles = malicious_titles.orderBy(col("pass_ratio").desc()).limit(20)
display(Top_20_titles)
```

title	pass_ratio
On Demand	1.0
Zola Levitt Presents	1.0
The Karate Kid Part III	1.0
Shall We Dance?	1.0
Angels Sing, Libera in America	1.0
21	1.0
February Sharathon	1.0
Young Guns II	1.0
KOMO 4 News 4:30am	1.0
Mindhunters	1.0

As we can see below by filtering the titles with 100% pass ratio, there are 5907 contenders for the top 20 titles. Therefore, the top 20 we displayed above were chosen by spark's default tie-breaking methods.

In [0]:

```
contenders_count = malicious_titles.filter(col("pass_ratio") == 1.0).count()
print(f"there are {contenders_count} titles with 100% pass rate")
```

```
there are 5907 titles with 100% pass rate
```