In memory Evaluation

This notebook includes trial runs for setvis vs upstetplot using functions in the accompanying `utils.py` module. It can be executed outside the browser.

In [1]:

```
import sys
import time
import pandas as pd
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
from IPython.display import clear_output

from utils import (generate_data,
                   plot_data,
                   compute)
```

In [2]:

```
def format_row(row):
    if row >= 1000000:
        return f"{row / 1000000}m"
    elif row >= 1000:
        return f"{row / 1000}k"
    else:
        return f"{row}"

def factors_of(x):
    # % of # of rows
    # min 1/1000
    # max
    # list
    # min 1000, these not allowed for 100k & 500 col
    # 0.0001, 0.0005, 0.001, 0.005,
    d2 = [0.001, 0.005, 0.01, 0.05, 0.1]
    return [int(i * x) for i in d2]

PM = 'planned missing'
GM = 'general missing'
SET = 'sets'
PATTERNS = [SET, GM, PM]
SETVIS = 'setvis'
UPSET = 'upset'
PACKAGES = [UPSET, SETVIS]
# upset not possible with 100000 rows & 500 cols
UPSET_COL_LIMIT = 500
UPSET_ROW_LIMIT = 100000
# given these
# unique combinations (GM_INTS): [100, 500, 1000, 5000, 10000]
# ROWS = [10000, 25000,50000, 100000, 500000]
# upsetplot set mode crashes 44gb memory with
# 10k rows & 5000 combs
# also crashes with 500k rows & 1000 combs
# For 500k rows & 500 combs it takes 80% of the memory
UPSET_COMB_LIMIT = UPSET_COL_LIMIT

# setvis not possible with planned missing 500000 x 1000 44gb machine
GM_ROW = 100000
GM_COL = 50
```

```
GM_INTS = factors_of(GM_ROW)
# setvis crashes 44GB machine in PM 1M rows & 700 cols
ROWS = [10000, 25000,50000, GM_ROW, 500000]
COLS = [10, GM_COL, 100, 500, 700]
COMPUTE = 'COMPUTE'
VIS = 'VISUALISE'
FIGURE_NUMBER = 0
```

In [3]:

```
# PATTERNS = [SET]
# GM_ROW = 100000
# GM_COL = 50
# GM_INTS = [10, 100]
# ROWS = [100, 1000]
# COLS = [GM_COL]
```

In [4]:

```
GM_INTS
```

Out[4]:

```
[100, 500, 1000, 5000, 10000]
```

Evaluate

calls the `compute` and `plot_data` functions. Each call is timed only as the function is called.

In [5]:

```
rams = []
times = []

def evaluate(df, row, col, set_mode, package, pattern):
    """Evaluates the performance of the "compute" and "plot_data" functions
    for a given dataframe and number of rows and columns.
    """
    # 1. capture compute time
    t = time.time()
    data = compute(df, package, set_mode)
    t = time.time() - t
    # we cannot have shapes of output yet so
    output_shape = [0,0]
    times.append((row,
                  2 if set_mode else col,
                  round(t, 3), pattern, package, COMPUTE, num_int,
                  output_shape[0], output_shape[1]))
    print(f'{t:.2f} secs')
    # 2. capture memory
    m = sys.getsizeof(data)
    m_col = m_row = 0
    if package == SETVIS:
        m_col = sys.getsizeof(data._intersection_id_to_columns)
        m_row = sys.getsizeof(data._intersection_id_to_records)
        m = m_col + m_row
        output_shape = data._intersection_id_to_columns.shape
    else:
        output_shape = data.shape
    m_df = sys.getsizeof(df)
    # order like times
    rams.append((row,
                 2 if set_mode else col,
                 m, pattern, package, m_df, num_int,
                 m_col, m_row, output_shape[0], output_shape[1]))

    # 3. capture plotting
    t = 0
    # skip upsetplot in set mode
    if (package != UPSET) | (pattern != SET):
        t = time.time()
        plot_data(df, data, package)
        t = time.time() - t

    times.append((row,
                  2 if set_mode else col,
                  round(t, 3), pattern, package, VIS, num_int,
                  output_shape[0], output_shape[1]))
```

Main

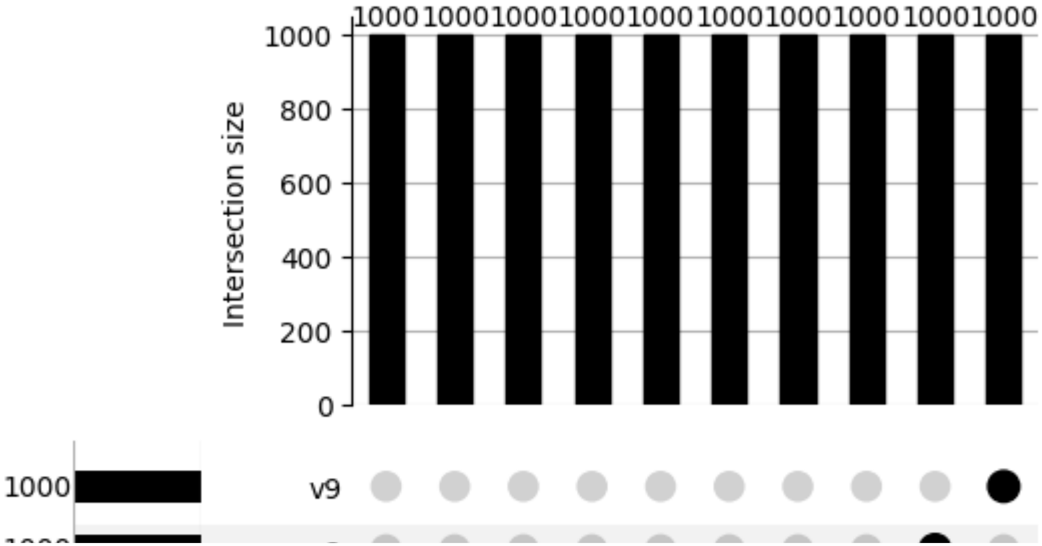
Current rows and columns are set with fast execution in mind. The comment in the code (constants cell) shows where upset and setvis start to overload memory. Likewise, the exectuion is only called once for the same reason.

In [6]:

```
start_time = time.time()
for c in COLS:
    for r in ROWS:
        for p in PATTERNS:
            for l in PACKAGES:
                if (l == UPSET) & (c >= UPSET_COL_LIMIT) & (r >= UPSET_ROW_LIMIT):
                    continue
                num_int = None
                set_mode = False
                if (p == GM) & (r == GM_ROW) & (c == GM_COL):
                    for i in GM_INTS:
                        num_int = i
                        print(l, "\t:", p, "\t", "\t combs: ", num_int)
                        df = generate_data(p, GM_ROW, GM_COL, num_int)
                        evaluate(df, GM_ROW, GM_COL, set_mode, l, p)
                elif (p == SET) & (c == GM_COL): # as c is ignored in generate_data for set mode
                    set_mode = True
                    for i in GM_INTS:
                        num_int = i
                        if (l == UPSET) & (num_int > UPSET_COMB_LIMIT):
                            print("Skipping upset with combinations: ", num_int)
                            continue
                        print(l, "\t:", p, "\t", "\t combs: ", [num_int, num_int/10])
                        print(f"Rows: {r}, Cols: {c}")
                        df = generate_data(p, r, c, [num_int, int(num_int/10)])
                        evaluate(df, r, c, set_mode, l, p)
                else:
                    if (p == GM) | (p == SET):
                        continue
                    else:
                        print(l, "\t:", p, "\t", r, "\t", c)
                        df = generate_data(p, r, c, num_int)
                        evaluate(df, r, c, set_mode, l, p)

# clear_output()
print(f"Done. Total time ({time.time() - start_time:.2f}s). Runs:",
      len(times))
```

```
upset   : planned missing        10000   10
0.03 secs
```


1000
1000
1000
1000
1000
1000
1000
1000
1000
1000
Intersection size
0
200
400
600
800
1000
1000
v9

In [7]:

```
def round_to_mb(m):
    return round(m/1024/1024, 5)

times_df = pd.DataFrame([
    {
        "rows": r,
        "columns": c,
        "colxrow": r * c if p != GM else num_int,
        "seconds": t,
        "pattern": p,
        "library": l,
        "compute": cp == COMPUTE,
        "combinations": num_int if p == SET else 0,
        "output_rows": out_row,
        "output_cols": out_col
    }
    for r, c, t, p, l, cp, num_int, out_row, out_col in times
])
mem_df = pd.DataFrame([
    {
        "rows": r,
        "columns": c,
        "colxrow": r * c if p != GM else num_int,
        "memory": round_to_mb(m),
        "pattern": p,
        "library": l,
        "memory_df": round_to_mb(mdf),
        "memory_col": round_to_mb(mcol),
        "memory_row": round_to_mb(mrow),
        "combinations": num_int if p == SET else 0,
        "output_rows": out_row,
        "output_cols": out_col
    }
    for r, c, m, p, l, mdf, num_int, mcol, mrow, out_row, out_col in rams
])
```

In [8]:

```
# write them to csv file
file = '-'.join(map(str, COLS)) + 'X' + '-'.join(map(str, ROWS))
times_df.to_csv(file + "-times.csv", index=False)
mem_df.to_csv(file + "-mems.csv", index=False)
```

Visualization

See Results notebook

Machine profile

In [9]:

```
import platform
import sys
import psutil
import pandas as pd

info = {
    "Total Memory": [psutil.virtual_memory().total / (1024**3)],
    "System": [platform.system()],
    "Machine": [platform.machine()],
    "Architecture": [platform.architecture()[0]],
    "Processor": [platform.processor()],
    "Release": [platform.release()],
    "Version": [platform.version()],
    "Python Version": [sys.version],
    "Python Version Info": [sys.version_info]
}

df = pd.DataFrame(info)

disk_io_counters = psutil.disk_io_counters()
if "nvme" in df["Machine"][0]:
    disk_type = "SSD"
else:
    disk_type = "HDD"

df["Disk Type"] = [disk_type]
display(df)
```

	Total Memory	System	Machine	Architecture	Processor	Release	Version	Python Version	Python Version Info	Disk Type
0	42.084702	Linux	x86_64	64bit	x86_64	5.4.0-139-generic	#156-Ubuntu SMP Fri Jan 20 17:27:18 UTC 2023	3.8.12 (default, Oct 12 2021, 13:49:34) \n[GCC...	(3, 8, 12, final, 0)	HDD