Introduction

This is the ***fifth*** of a series where I look at big datasets, and in each case I'm using a different tool to carry out the same analysis on the same dataset.

This time I'm using **PostgreSQl**, an open source relational database, together with its admin tool **pgAdmin**. You can find each notebook in the series in my Github repo, including:

1. Pandas chunksize
2. Dask library
3. PySpark
4. Talend Open Studio
5. PostgreSQL

There is a little more explanation in the first notebook (Pandas chunksize) on the overall approach to the analysis. In the other notebooks I focus more on the elements specific to the tool being used.

Dataset description

Throughout the series we'll use the SmartMeter Energy Consumption Data in London Households dataset, which according to the website contains:

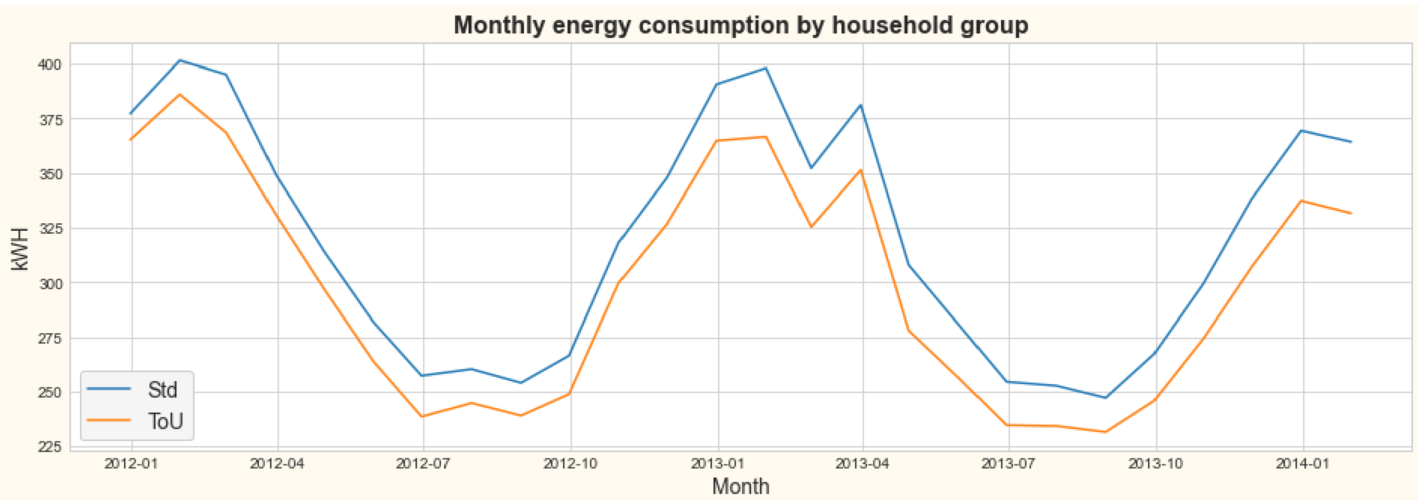
> Energy consumption readings for a sample of 5,567 London Households that took part in the UK Power Networks led Low Carbon London project between November 2011 and February 2014.

The households were divided into two groups:

- Those who were sent Dynamic Time of Use (dToU) energy prices (labelled "High", "Medium", or "Low") a day in advance of the price being applied.
- Those who were subject to the Standard tariff.

One aim of the study was to see if pricing knowledge would affect energy consumption behaviour.

Results



The results show the expected seasonal variation with a clear difference between the two groups, suggesting that energy price knowledge does indeed help reduce energy consumption.

The rest of the notebook shows how this chart was produced from the raw data.

Accessing the data

The data is downloadable as a single zip file which contains a csv file of 167 million rows. If the `curl` command doesn't work (and it will take a while as it's a file of 800MB), you can download the file here and put it in the folder `data` which is in the folder where this notebook is saved.

In [ ]:
```
!curl "https://data.london.gov.uk/download/smartmeter-energy-use-data-in-london-households/3527bf39-d93e-4071-8451-df2ade1ea4f2/LCL-FullData.zip" --location --create-dirs -o "data/LCL-FullData.zip"
```

First we unzip the data. This may take a while! Alternatively you can unzip it manually using whatever unzip utility you have. Just make sure the extracted file is in a folder called `data` within the folder where your notebook is saved.

In [ ]:
```
!unzip "data/LCL-FullData.zip" -d "data"
```

Examining the data

First we use pandas to create a little test file of 1,000,000 rows.

In [1]:
```
import pandas as pd
from IPython.display import HTML
```

In [2]:
```
chunks = pd.read_csv('data/CC_LCL-FullData.csv', chunksize=1000000)
type(chunks)
```

Out[2]:
```
pandas.io.parsers.readers.TextFileReader
```

In [3]:
```
table_style = [{
    'selector' : 'caption',
    'props' : [
        ('font-size', '16px'),
        ('color', 'black'),
        ('font-weight', 'bold'),
        ('text-align', 'left')
    ]
}]

for chunk in chunks:

    display(
        chunk.describe(include='all')
        .style.set_caption('Describe')
        .set_table_styles(table_style)
    )

    display(
        chunk.head()
        .style.set_caption('Head')
        .set_table_styles(table_style)
    )
```

```python
    display(HTML('<br><span style="font-weight: bold; font-size: 16px">Info</span>'))
    display(chunk.info())

    test_data = chunk

    break # Just the first chunk
```

Describe

	LCLid	stdorToU	DateTime	KWH/hh (per half hour)
count	1000000	1000000	1000000	1000000
unique	30	1	39102	4801
top	MAC000018	Std	2012-11-20 00:00:00.0000000	0
freq	39082	1000000	58	45538

Head

	LCLid	stdorToU	DateTime	KWH/hh (per half hour)
0	MAC000002	Std	2012-10-12 00:30:00.0000000	0
1	MAC000002	Std	2012-10-12 01:00:00.0000000	0
2	MAC000002	Std	2012-10-12 01:30:00.0000000	0
3	MAC000002	Std	2012-10-12 02:00:00.0000000	0
4	MAC000002	Std	2012-10-12 02:30:00.0000000	0

Info

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000000 entries, 0 to 999999
Data columns (total 4 columns):
 #   Column                  Non-Null Count    Dtype 
---  ------                  --------------    ----- 
 0   LCLid                   1000000 non-null  object
 1   stdorToU                1000000 non-null  object
 2   DateTime                1000000 non-null  object
 3   KWH/hh (per half hour)  1000000 non-null  object
dtypes: object(4)
memory usage: 30.5+ MB
None
```

The column `KWH/hh (per half hour)` is of type `object` and not `float` which is surprising, so that probably means there are some non-numeric values we'll need to deal with.

In [4]:
```python
test_data
```

Out[4]:

	LCLid	stdorToU	DateTime	KWH/hh (per half hour)
0	MAC000002	Std	2012-10-12 00:30:00.0000000	0
1	MAC000002	Std	2012-10-12 01:00:00.0000000	0
2	MAC000002	Std	2012-10-12 01:30:00.0000000	0
3	MAC000002	Std	2012-10-12 02:00:00.0000000	0
4	MAC000002	Std	2012-10-12 02:30:00.0000000	0
...	...	...	...	...
999995	MAC000036	Std	2012-11-08 08:00:00.0000000	0.228
999996	MAC000036	Std	2012-11-08 08:30:00.0000000	0.042
999997	MAC000036	Std	2012-11-08 09:00:00.0000000	0.076
999998	MAC000036	Std	2012-11-08 09:30:00.0000000	0.07
999999	MAC000036	Std	2012-11-08 10:00:00.0000000	0.005

1000000 rows × 4 columns

We save our test data as a csv file so we can load into PostgreSQL.

In [5]:
```
test_data.to_csv("data/sql-test-data.csv", index=False)
```

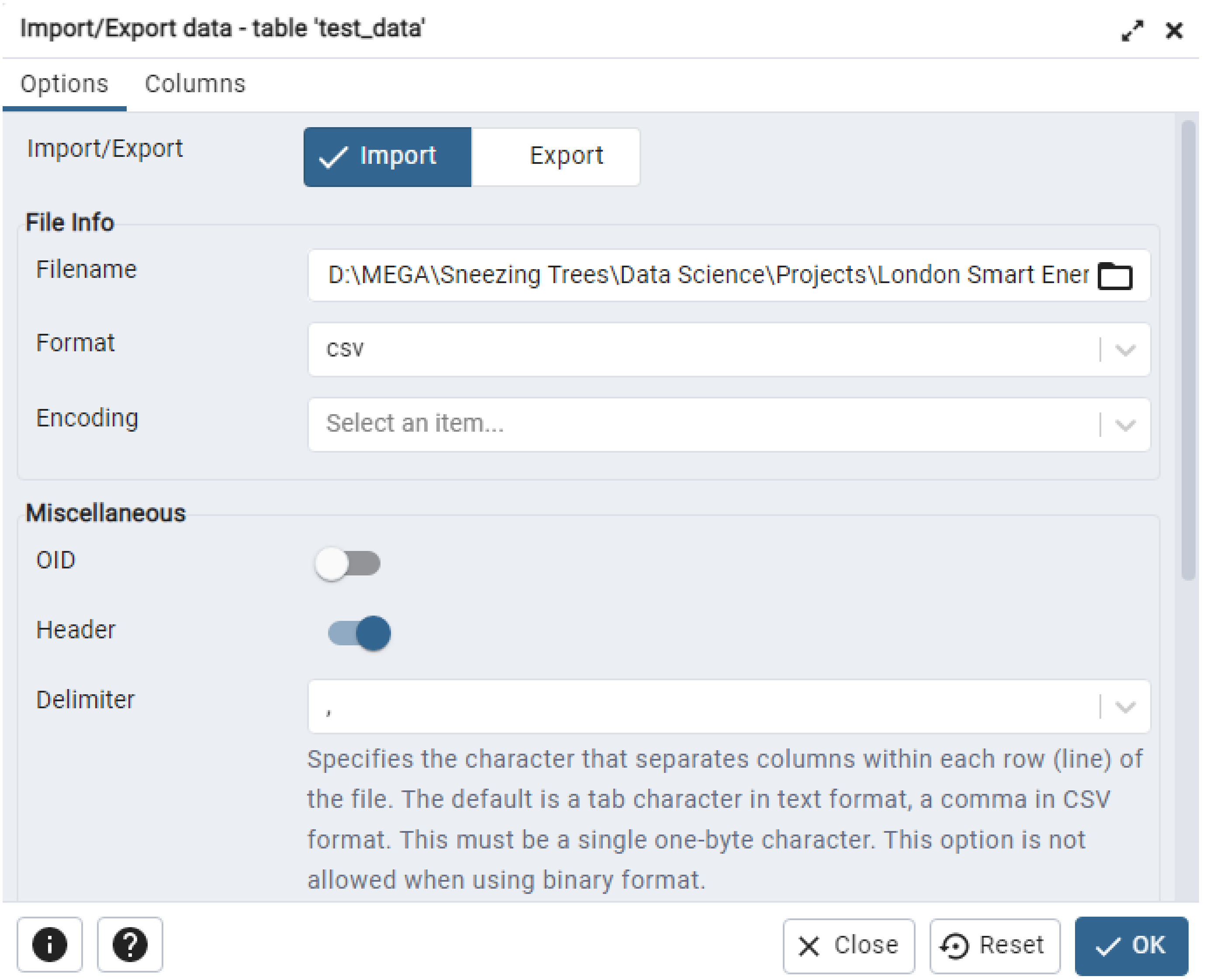
Importing the data

First we create a table to import into. We add an auto-increment `id` column (which will come in handy later). Note the type of `kWh_raw_data` is `TEXT` because we suspect we have some text values to deal with.

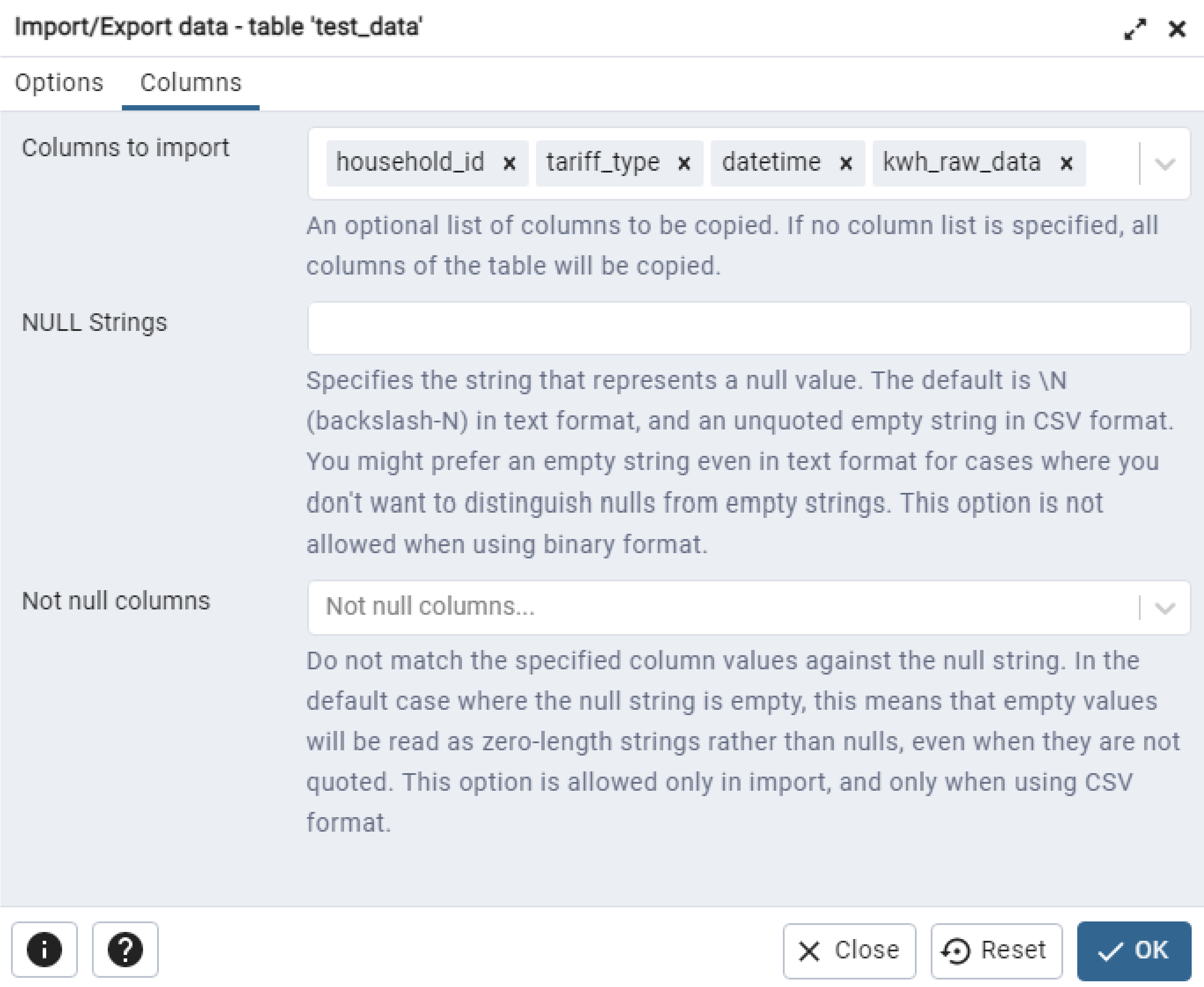
```
CREATE TABLE test_data
   (
        id SERIAL PRIMARY KEY,
        Household_ID TEXT,
        Tariff_Type TEXT,
        Datetime TEXT,
        kWh_raw_data TEXT
   );
```

Then we use the pgAdmin import tool to load the data from the test file. The tool is accessed by right-clicking on the table we want to fill and choosing Import/Export Data...

In the Import tab, we just specify the csv file we're importing from, the delimiter and the fact that we have a header in the file.



In the Columns tab, we just need to remove the id column (as we aren't importing that) to leave us with the four columns of data we want.



We can view the imported data by right-clicking on the table and choosing View/Edit Data. (I chose the option to view the first 100 rows.)

Data output Messages Notifications

	id [PK] integer	household_id text	tariff_type text	datetime text	kwh_raw_data text
1	1	MAC000002	Std	2012-10-12 00:30:00.0000000	0
2	2	MAC000002	Std	2012-10-12 01:00:00.0000000	0
3	3	MAC000002	Std	2012-10-12 01:30:00.0000000	0
4	4	MAC000002	Std	2012-10-12 02:00:00.0000000	0
5	5	MAC000002	Std	2012-10-12 02:30:00.0000000	0
6	6	MAC000002	Std	2012-10-12 03:00:00.0000000	0
7	7	MAC000002	Std	2012-10-12 03:30:00.0000000	0
8	8	MAC000002	Std	2012-10-12 04:00:00.0000000	0
9	9	MAC000002	Std	2012-10-12 04:30:00.0000000	0
10	10	MAC000002	Std	2012-10-12 05:00:00.0000000	0
11	11	MAC000002	Std	2012-10-12 05:30:00.0000000	0
12	12	MAC000002	Std	2012-10-12 06:00:00.0000000	0
13	13	MAC000002	Std	2012-10-12 06:30:00.0000000	0
14	14	MAC000002	Std	2012-10-12 07:00:00.0000000	0
15	15	MAC000002	Std	2012-10-12 07:30:00.0000000	0
16	16	MAC000002	Std	2012-10-12 08:00:00.0000000	0
17	17	MAC000002	Std	2012-10-12 08:30:00.0000000	0
18	18	MAC000002	Std	2012-10-12 09:00:00.0000000	0
19	19	MAC000002	Std	2012-10-12 09:30:00.0000000	0
20	20	MAC000002	Std	2012-10-12 10:00:00.0000000	0
21	21	MAC000002	Std	2012-10-12 10:30:00.0000000	0
22	22	MAC000002	Std	2012-10-12 11:30:00.0000000	0.143
23	23	MAC000002	Std	2012-10-12 12:00:00.0000000	0.663
24	24	MAC000002	Std	2012-10-12 12:30:00.0000000	0.256
25	25	MAC000002	Std	2012-10-12 13:00:00.0000000	0.155
26	26	MAC000002	Std	2012-10-12 13:30:00.0000000	0.199
27	27	MAC000002	Std	2012-10-12 14:00:00.0000000	0.125
28	28	MAC000002	Std	2012-10-12 14:30:00.0000000	0.165
29	29	MAC000002	Std	2012-10-12 15:00:00.0000000	0.14

Total rows: 100 of 100 Query complete 00:00:00.160

We can try to convert our `kwh_raw_data` to numeric in a new column:

```
ALTER TABLE test_data ADD COLUMN kwh NUMERIC;
UPDATE test_data SET kwh = CAST(kWh_raw_data AS NUMERIC);
```

But we get an error on the update:

```
ERROR:  invalid input syntax for type numeric: "Null"
```

It looks like we have some `"Null"` values in the `kwh_raw_data` column. Let's check:

```
SELECT * FROM test_data WHERE kwh_raw_data = 'Null';
```

We see we have 29 rows with a `"Null"` value in our test data.

Data output   Messages   Notifications

	id [PK] integer	household_id text	tariff_type text	datetime text	kwh_raw_data text	kwh numeric
1	3241	MAC000002	Std	2012-12-...	Null	[null]
2	38711	MAC000003	Std	2012-12-...	Null	[null]
3	70387	MAC000004	Std	2012-12-...	Null	[null]
4	106847	MAC000006	Std	2012-12-...	Null	[null]
5	131898	MAC000007	Std	2012-12-...	Null	[null]
6	183153	MAC000009	Std	2012-12-...	Null	[null]
7	163720	MAC000008	Std	2012-12-...	Null	[null]
8	208193	MAC000010	Std	2012-12-...	Null	[null]
9	618214	MAC000025	Std	2012-12-...	Null	[null]
10	231900	MAC000011	Std	2012-12-...	Null	[null]
11	256570	MAC000012	Std	2012-12-...	Null	[null]
12	344745	MAC000018	Std	2012-12-...	Null	[null]
13	422897	MAC000020	Std	2012-12-...	Null	[null]
14	461975	MAC000021	Std	2012-12-...	Null	[null]

Total rows: 29 of 29   Query complete 00:00:00.178

We can delete those easily enough:

```
DELETE FROM test_data WHERE kwh_raw_data = 'Null';
```

And now we should be able to convert our kwh data to numeric:

```
UPDATE test_data SET kwh = CAST(kWh_raw_data AS NUMERIC);
```

It works - but it's slow. 11 seconds for just the test data. We'll look at that when we work on the full data.

Let's also check for duplicates. This is where the `id` column we created is useful.

```
SELECT * FROM test_data
WHERE id IN (
    SELECT id
    FROM (
        SELECT id,
        ROW_NUMBER() OVER(
            PARTITION BY Household_ID, Tariff_Type, Datetime
            ORDER BY  id
        ) AS row_num
        FROM test_data
    ) t
```

```
        WHERE t.row_num > 1
    );
```

We find we have 688 duplicates in our test data.

Data output   Messages   Notifications

	id [PK] integer	household_id text	tariff_type text	datetime text	kwh_raw_data text	kwh numeric
1	7780	MAC000002	Std	2013-03-24 00:00:00.0000000	0.486	0.486
2	44738	MAC000003	Std	2013-04-24 00:00:00.0000000	1.424	1.424
3	65991	MAC000004	Std	2012-09-19 00:00:00.0000000	0	0
4	134947	MAC000007	Std	2013-02-21 00:00:00.0000000	0.179	0.179
5	160815	MAC000008	Std	2012-10-20 00:00:00.0000000	0.267	0.267
6	172726	MAC000008	Std	2013-06-25 00:00:00.0000000	0.281	0.281
7	180246	MAC000009	Std	2012-10-20 00:00:00.0000000	0.041	0.041
8	186203	MAC000009	Std	2013-02-21 00:00:00.0000000	0.098	0.098
9	189181	MAC000009	Std	2013-04-24 00:00:00.0000000	0.051	0.051
10	190670	MAC000009	Std	2013-05-25 00:00:00.0000000	0.112	0.112
11	204062	MAC000009	Std	2014-02-28 00:00:00.0000000	0.05	0.05
12	267064	MAC000012	Std	2013-07-26 00:00:00.0000000	0.05	0.05
13	298166	MAC000013	Std	2013-08-26 00:00:00.0000000	0.063	0.063
14	367551	MAC000019	Std	2012-01-15 00:00:00.0000000	0.063	0.063
15	376485	MAC000019	Std	2012-07-19 00:00:00.0000000	0.046	0.046
16	411099	MAC000020	Std	2012-04-17 00:00:00.0000000	0.056	0.056
17	448689	MAC000021	Std	2012-03-17 00:00:00.0000000	0.47	0.47
18	457623	MAC000021	Std	2012-09-19 00:00:00.0000000	0.314	0.314

Total rows: 688 of 688   Query complete 00:00:04.056

And we can remove those easily too by replacing `SELECT *` with `DELETE`:

```
    DELETE FROM test_data
    WHERE id IN (
        SELECT id
        FROM (
            SELECT id,
            ROW_NUMBER() OVER(
                PARTITION BY Household_ID, Tariff_Type, Datetime
                ORDER BY  id
            ) AS row_num
            FROM test_data
        ) t
        WHERE t.row_num > 1
    );
```

Aggregating the test data

The goal here is to ***reduce*** the data by aggregating it in some way. Since we know that we have data in half-hour intervals, we'll aggregate it to daily data by summing over each 24-hour period. That should reduce the number of rows by a factor of about 48.

First we need to create a `Date` column.

```
ALTER TABLE test_data ADD COLUMN Date TEXT;
UPDATE test_data SET Date = LEFT(Datetime, 10);

SELECT * FROM test_data;
```

Data output Messages Notifications

	id [PK] integer	household_id text	tariff_type text	datetime text	kwh_raw_data text	kwh numeric	date text
1	5945	MAC000002	Std	2013-02-13 19:30:00.0000000	0.256	0.256	2013-02-13
2	5946	MAC000002	Std	2013-02-13 20:00:00.0000000	0.272	0.272	2013-02-13
3	5947	MAC000002	Std	2013-02-13 20:30:00.0000000	0.838	0.838	2013-02-13
4	5948	MAC000002	Std	2013-02-13 21:00:00.0000000	0.248	0.248	2013-02-13
5	5949	MAC000002	Std	2013-02-13 21:30:00.0000000	0.214	0.214	2013-02-13
6	5950	MAC000002	Std	2013-02-13 22:00:00.0000000	0.275	0.275	2013-02-13
7	5951	MAC000002	Std	2013-02-13 22:30:00.0000000	0.247	0.247	2013-02-13
8	5952	MAC000002	Std	2013-02-13 23:00:00.0000000	0.258	0.258	2013-02-13
9	5953	MAC000002	Std	2013-02-13 23:30:00.0000000	0.258	0.258	2013-02-13
10	5954	MAC000002	Std	2013-02-14 00:00:00.0000000	0.212	0.212	2013-02-14
11	5955	MAC000002	Std	2013-02-14 00:30:00.0000000	0.252	0.252	2013-02-14
12	5956	MAC000002	Std	2013-02-14 01:00:00.0000000	0.225	0.225	2013-02-14
13	5957	MAC000002	Std	2013-02-14 01:30:00.0000000	0.255	0.255	2013-02-14
14	5958	MAC000002	Std	2013-02-14 02:00:00.0000000	0.234	0.234	2013-02-14

Total rows: 1000 of 999971 Query complete 00:00:01.086

And now we can aggregate:

```
SELECT Household_ID, Tariff_Type, Date, SUM(kwh)
FROM lse_test_data
GROUP BY Household_ID, Tariff_Type, Date;
```

Data output Messages Notifications

	household_id text	tariff_type text	date text	sum numeric
1	MAC000002	Std	2012-10-12	7.098
2	MAC000002	Std	2012-10-13	11.087
3	MAC000002	Std	2012-10-14	13.223
4	MAC000002	Std	2012-10-15	10.257
5	MAC000002	Std	2012-10-16	9.769
6	MAC000002	Std	2012-10-17	10.885
7	MAC000002	Std	2012-10-18	10.751
8	MAC000002	Std	2012-10-19	8.431
9	MAC000002	Std	2012-10-20	17.5779999
10	MAC000002	Std	2012-10-21	24.4900001
11	MAC000002	Std	2012-10-22	18.885
12	MAC000002	Std	2012-10-23	10.485
13	MAC000002	Std	2012-10-24	15.5370001
14	MAC000002	Std	2012-10-25	13.128
15	MAC000002	Std	2012-10-26	15.065
16	MAC000002	Std	2012-10-27	16.886
17	MAC000002	Std	2012-10-28	19.6289999
18	MAC000002	Std	2012-10-29	12.779

Total rows: 1000 of 20870 Query complete 00:00:00.428

Processing the full data

We'll apply the same principles to the full data but with a small modification.

Importing

First we create a new table.

```
CREATE TABLE full_data
   (
    id SERIAL PRIMARY KEY,
    Household_ID TEXT,
    Tariff_Type TEXT,
    Datetime TEXT,
    kWh_raw_data TEXT
   );
```

And then we import in exactly the same way as we did for the test data. This takes a while: 7 - 10 minutes on my laptop depending on what else is running.

Deduplication

We'll start by removing the duplicates This takes 25 - 30 minutes on my laptop.

```
DELETE FROM full_data
WHERE id IN (
    SELECT id
    FROM (
        SELECT id,
        ROW_NUMBER() OVER(
            PARTITION BY Household_ID, Tariff_Type, Datetime
            ORDER BY id
        ) AS row_num
        FROM full_data
    ) t
    WHERE t.row_num > 1
);
```

Conversion

Now we'll convert the `kwh` data to numeric. But rather than removing the `'Null'` rows and then using `UPDATE`, we'll create another table from a `SELECT` query as it's a faster method. And we may as well create the `Date` column at the same time. This takes 8 - 10 minutes to execute on my laptop.

```
CREATE TABLE lse_data AS
SELECT
CAST(Household_ID AS TEXT) Household_ID,
CAST(Tariff_Type AS TEXT) Tariff_Type,
CAST(LEFT(Datetime, 10) AS TEXT) Date,
CAST(kWh_raw_data AS NUMERIC) kWh
FROM full_data
WHERE kWh_raw_data != 'Null';
```

Then we can drop the original table so as not to take up space unnecessarily.

```
DROP TABLE full_data;
```

Aggregation

Finally we can aggregate. And we'll again create a table so we don't need to rerun the query to access the results.

```
CREATE TABLE lse_agg_data AS
SELECT Household_ID, Tariff_Type, Date, SUM(kwh) kwh
FROM lse_data
GROUP BY Household_ID, Tariff_Type, Date;
```

The last step is to save the results to a csv using the pgdmin Export Data tool (by right-clicking on the table and selecting Import/Export Data...

I save it to a file called "daily-summary-data-postgresql.csv" in the "data" folder.

Viewing the results

We can access the resulting data in a Pandas dataframe.

In [6]:
```
daily_summary = (
    pd.read_csv("data/daily-summary-data-postgresql.csv")
)
```

In [7]:
```
daily_summary
```

Out[7]:

	household_id	tariff_type	date	kwh
0	MAC000002	Std	2012-10-12	7.098
1	MAC000002	Std	2012-10-13	11.087
2	MAC000002	Std	2012-10-14	13.223
3	MAC000002	Std	2012-10-15	10.257
4	MAC000002	Std	2012-10-16	9.769
...	...	...	...	...
3510398	MAC005567	Std	2014-02-24	4.107
3510399	MAC005567	Std	2014-02-25	5.762
3510400	MAC005567	Std	2014-02-26	5.066
3510401	MAC005567	Std	2014-02-27	3.217
3510402	MAC005567	Std	2014-02-28	0.183

3510403 rows × 4 columns

In [8]:
```
daily_summary = (
    pd.read_csv("data/daily-summary-data-postgresql.csv").sort_values(
        ['household_id', 'tariff_type', 'date']
    )
    .rename(columns = {
        'household_id' : 'Household ID',
        'tariff_type' : 'Tariff Type',
        'date' : 'Date',
        'kwh' : 'kWh'
    })
)
```

In [9]:
```
daily_summary
```

Out[9]:

	Household ID	Tariff Type	Date	kWh
0	MAC000002	Std	2012-10-12	7.098
1	MAC000002	Std	2012-10-13	11.087
2	MAC000002	Std	2012-10-14	13.223
3	MAC000002	Std	2012-10-15	10.257
4	MAC000002	Std	2012-10-16	9.769
...	...	...	...	...
3510398	MAC005567	Std	2014-02-24	4.107
3510399	MAC005567	Std	2014-02-25	5.762
3510400	MAC005567	Std	2014-02-26	5.066
3510401	MAC005567	Std	2014-02-27	3.217
3510402	MAC005567	Std	2014-02-28	0.183

3510403 rows × 4 columns

Saving aggregated data

Now that we have reduced the data down to about 3 million rows it should be managable in a single dataframe. It's useful to save the data so that we don't have to re-run the aggregation every time we want to work on the aggregated data.

We'll save it in a compressed gz format - pandas automatically recognizes the filetype we specify.

In [10]:
```
daily_summary.to_csv("data/daily-summary-data.gz", index=False)
```

The rest of this notebook is now essentially the same processing as applied in all the other notebooks in the series.

Analysing the data

In [11]:
```
saved_daily_summary = pd.read_csv("data/daily-summary-data.gz")
```

In [12]:
```
saved_daily_summary
```

Out[12]:

	Household ID	Tariff Type	Date	kWh
0	MAC000002	Std	2012-10-12	7.098
1	MAC000002	Std	2012-10-13	11.087
2	MAC000002	Std	2012-10-14	13.223
3	MAC000002	Std	2012-10-15	10.257
4	MAC000002	Std	2012-10-16	9.769
...	...	...	...	...
3510398	MAC005567	Std	2014-02-24	4.107
3510399	MAC005567	Std	2014-02-25	5.762
3510400	MAC005567	Std	2014-02-26	5.066
3510401	MAC005567	Std	2014-02-27	3.217
3510402	MAC005567	Std	2014-02-28	0.183

3510403 rows × 4 columns

Out of interest let's see what sort of data coverage we have. First we re-organize so that we have households as columns and dates as rows.

In [13]:

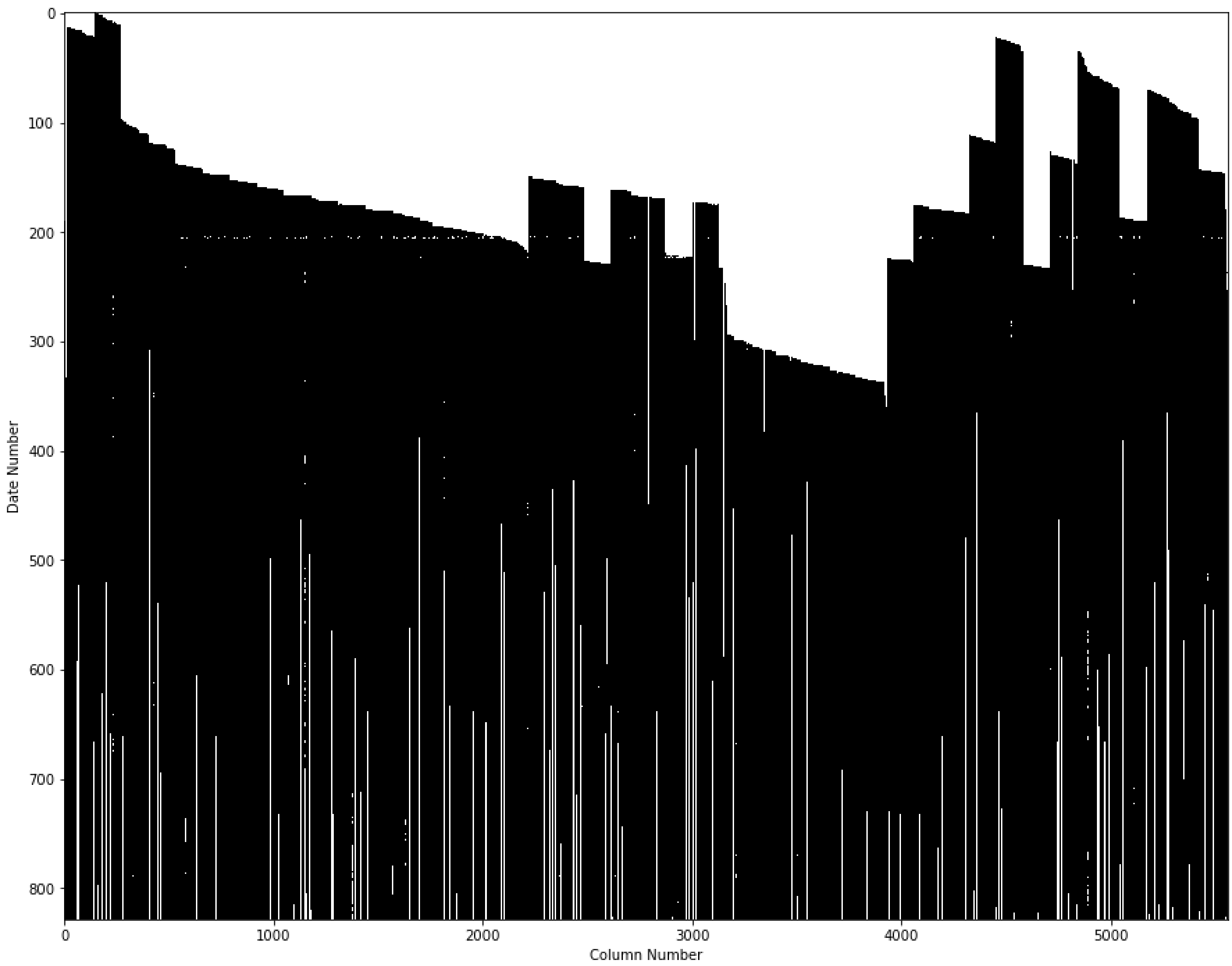
```
summary_table = saved_daily_summary.pivot_table(
    'kWh',
    index='Date',
    columns='Household ID',
    aggfunc='sum'
)
```

Then we can plot where we have data (black) and where we don't (white).

In [14]:

```
import matplotlib.pyplot as plt

plt.figure(figsize=(15, 12))
plt.imshow(summary_table.isna(), aspect="auto", interpolation="nearest", cmap="gray")
plt.xlabel("Column Number")
plt.ylabel("Date Number");
```



Despite a slightly patchy data coverage, averaging by tariff type across all households for each day should give us a useful comparison.

In [15]:
```
daily_mean_by_tariff_type = saved_daily_summary.pivot_table(
    'kWh',
    index='Date',
    columns='Tariff Type',
    aggfunc='mean'
)
daily_mean_by_tariff_type
```

Out[15]:

Tariff Type	Std	ToU
Date		
2011-11-23	7.430000	4.327500
2011-11-24	8.998333	6.111750
2011-11-25	10.102885	6.886333
2011-11-26	10.706257	7.709500
2011-11-27	11.371486	7.813500
...	...	...
2014-02-24	10.580187	9.759439
2014-02-25	10.453365	9.683862
2014-02-26	10.329026	9.716652
2014-02-27	10.506416	9.776561
2014-02-28	0.218075	0.173949

829 rows × 2 columns

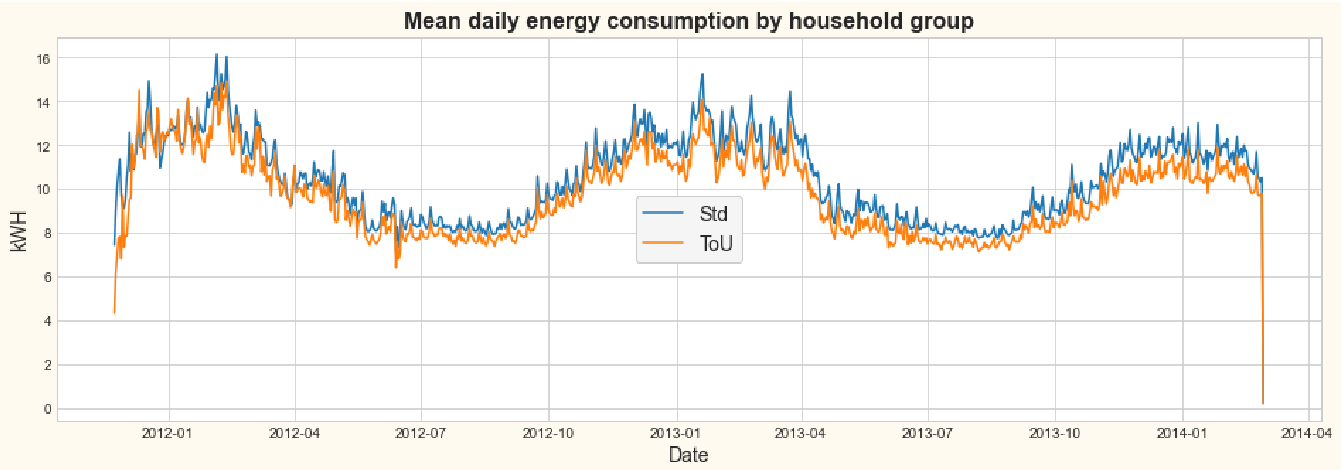
Finally we can plot the two sets of data. The plotting works better if we convert the date from type `string` to type `datetime`.

In [16]:
```
daily_mean_by_tariff_type.index = pd.to_datetime(daily_mean_by_tariff_type.index)
```

In [17]:
```
plt.style.use('seaborn-whitegrid')

plt.figure(figsize=(16, 5), facecolor='floralwhite')
for tariff in daily_mean_by_tariff_type.columns.to_list():
    plt.plot(
        daily_mean_by_tariff_type.index.values,
        daily_mean_by_tariff_type[tariff],
        label = tariff
    )

plt.legend(loc='center', frameon=True, facecolor='whitesmoke', framealpha=1, fontsize=14)
plt.title(
    'Mean daily energy consumption by household group',
    fontdict = {'fontsize' : 16, 'fontweight' : 'bold'}
)
plt.xlabel('Date', fontsize = 14)
plt.ylabel('kWH', fontsize = 14)
plt.show()
```



The pattern looks seasonal which makes sense given heating energy demand.

It also looks like there's a difference between the two groups with the ToU group tending to consume less, but the display is too granular. Let's aggregate again into months.

In [18]:
```
daily_mean_by_tariff_type
```

Out[18]:

Tariff Type	Std	ToU
Date		
2011-11-23	7.430000	4.327500
2011-11-24	8.998333	6.111750
2011-11-25	10.102885	6.886333
2011-11-26	10.706257	7.709500
2011-11-27	11.371486	7.813500
...	...	...
2014-02-24	10.580187	9.759439
2014-02-25	10.453365	9.683862
2014-02-26	10.329026	9.716652
2014-02-27	10.506416	9.776561
2014-02-28	0.218075	0.173949

829 rows × 2 columns

We can see that the data starts partway through November 2011, so we'll start from 1 December. It looks like the data finishes perfectly at the end of February, but the last value looks suspiciously low compared to the others. It seems likely the data finished part way through the last day. This may be a problem elsewhere in the data too, but it shouldn't have an enormous effect as at worst it will reduce the month's energy consumption for that household by two days (one at the beginning and one at the end).

In [19]:
```
monthly_mean_by_tariff_type = daily_mean_by_tariff_type['2011-12-01' : '2014-01-31'].resample('M').sum()
monthly_mean_by_tariff_type
```

Out[19]:

Tariff Type Date	Std	ToU
2011-12-31	377.218580	365.145947
2012-01-31	401.511261	386.016403
2012-02-29	395.065321	368.475150
2012-03-31	349.153085	330.900633
2012-04-30	314.173857	296.903425
2012-05-31	281.666428	263.694338
2012-06-30	257.204029	238.417505
2012-07-31	260.231952	244.641359
2012-08-31	253.939017	238.904096
2012-09-30	266.392972	248.707929
2012-10-31	318.214026	299.714701
2012-11-30	347.818025	326.651435
2012-12-31	390.616106	364.754528
2013-01-31	398.004581	366.548143
2013-02-28	352.189818	325.298845
2013-03-31	381.191994	351.371278
2013-04-30	307.857771	277.856327
2013-05-31	280.762752	256.292247
2013-06-30	254.399013	234.481016
2013-07-31	252.609890	234.104814
2013-08-31	247.046087	231.347310
2013-09-30	267.024791	245.597424
2013-10-31	299.533302	274.332936
2013-11-30	338.082197	306.942424
2013-12-31	369.381371	337.331504
2014-01-31	364.225310	331.578243

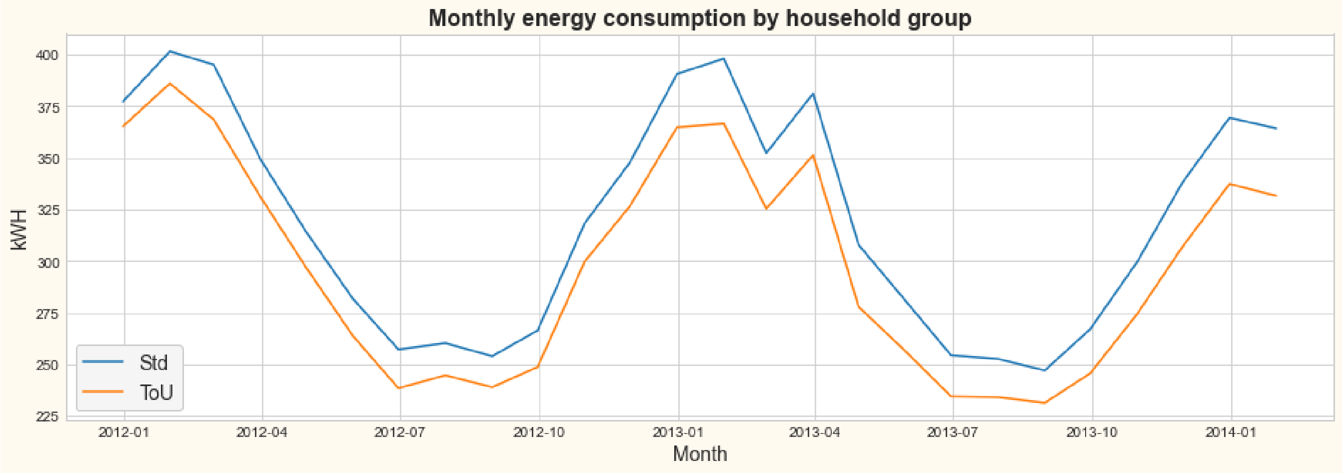
In [20]:

```
plt.figure(figsize=(16, 5), facecolor='floralwhite')
for tariff in daily_mean_by_tariff_type.columns.to_list():
    plt.plot(
        monthly_mean_by_tariff_type.index.values,
        monthly_mean_by_tariff_type[tariff],
        label = tariff
    )

plt.legend(loc='lower left', frameon=True, facecolor='whitesmoke', framealpha=1, fontsize=14)
plt.title(
    'Monthly energy consumption by household group',
    fontdict = {'fontsize' : 16, 'fontweight' : 'bold'}
)
plt.xlabel('Month', fontsize = 14)
plt.ylabel('kWH', fontsize = 14)
```

```
# Uncomment for a copy to display in results
# plt.savefig(fname='images/result1-no-dupes.png', bbox_inches='tight')

plt.show()
```



The pattern is much clearer and there is an obvious difference between the two groups of consumers.

Note that the chart does not show mean monthly energy consumption, but the sum over each month of the daily means. To calculate true monthly means we would need to drop the daily data for each household where the data was incomplete for a month. Our method should give a reasonable approximation.