Introduction

This is the ***second*** 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 the **Dask library** for parallel computing to manage a large file size. You can find each notebook in the series in my Github repo, including:

1. Pandas chunksize
2. Dask library

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:

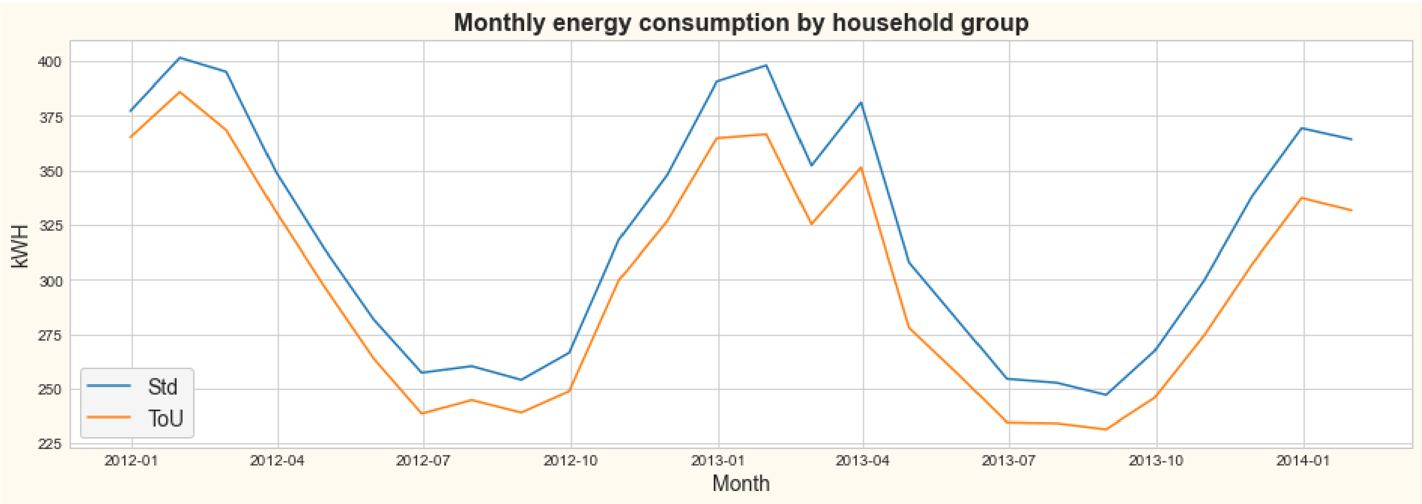
> Energy consumption readings for a sample of 5,567 London Households that took part in the UK Power Networks led Low Carbon London project between November 2011 and February 2014.

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.

Introduction to Dask

According to the docs:

> *Dask is a flexible library for parallel computing in Python*
>
> Dask is composed of two parts:
>
> - Dynamic task scheduling optimized for computation. This is similar to Airflow, Luigi, Celery, or Make, but optimized for interactive computational workloads.
> - “Big Data” collections like parallel arrays, dataframes, and lists that extend common interfaces like NumPy, Pandas, or Python iterators to larger-than-memory or distributed environments. These parallel collections run on top of dynamic task schedulers.

This means that not only can we process larger-than-memory files, but unlike the pandas chunksize approach, we can also make use of clusters - or multiple CPU cores when working on a single machine.

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

In [1]:
```
import pandas as pd
import numpy as np
from dask import dataframe as dd
```

Now let's load the data into a Dask dataframe.

In [2]:
```
raw_data_ddf = dd.read_csv('data/CC_LCL-FullData.csv')
raw_data_ddf
```

Out[2]: **Dask DataFrame Structure:**

	LCLid	stdorToU	DateTime	KWH/hh (per half hour)
npartitions=133				
	object	object	object	int64
	...	...	...	...
...	...	...	...	...
	...	...	...	...
	...	...	...	...

Dask Name: read-csv, 133 tasks

Viewing the dataframe shows that Dask has divided our data into 133 partitions. Dask has also "guessed" the data types by taking a sample of data. Leaving the dataframe as it is will eventually cause errors, because the kWh data is a mix of numbers and 'Null' string values.

As a first step we can specify the kWh data type, using `object` to handle strings.

In [3]:
```
raw_data_ddf = dd.read_csv(
    'data/CC_LCL-FullData.csv',
    dtype={'KWH/hh (per half hour) ': 'object'}
)
raw_data_ddf
```

Out[3]: **Dask DataFrame Structure:**

	LCLid	stdorToU	DateTime	KWH/hh (per half hour)
npartitions=133				
	object	object	object	object
	...	...	...	...
...	...	...	...	...
	...	...	...	...
	...	...	...	...

Dask Name: read-csv, 133 tasks

We rename the columns to make them more readable.

In [4]:
```
col_renaming = {
    'LCLid' : 'Household ID',
    'stdorToU' : 'Tariff Type',
    'KWH/hh (per half hour) ' : 'kWh'
}
full_data_ddf = raw_data_ddf.rename(columns=col_renaming)
```

Let's work on a small subset of the data (1,000,000 rows) to develop each processing step.

In [5]:
```
test_data = full_data_ddf.head(1000000)
test_data
```

Out[5]:

	Household ID	Tariff Type	DateTime	kWh
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 need to convert this data back into a Dask dataframe. We'll split it into 2 partitions so we know we are testing across partitions.

In [6]:

```
test_data_ddf = dd.from_pandas(test_data, npartitions=2)
test_data_ddf
```

Out[6]: **Dask DataFrame Structure:**

	Household ID	Tariff Type	DateTime	kWh
npartitions=2				
0	object	object	object	object
500000	...	...	...	...
999999	...	...	...	...

Dask Name: from_pandas, 2 tasks

Cleaning the data

We can see there `"Null"` string values in the kWH data.

In [7]:

```
test_nulls = test_data[test_data['kWh'] == 'Null']
test_nulls
```

Out[7]:

	Household ID	Tariff Type	DateTime	kWh
3240	MAC000002	Std	2012-12-19 12:37:27.0000000	Null
38710	MAC000003	Std	2012-12-19 12:37:26.0000000	Null
70386	MAC000004	Std	2012-12-19 12:32:40.0000000	Null
106846	MAC000006	Std	2012-12-19 12:37:26.0000000	Null
131897	MAC000007	Std	2012-12-19 12:37:27.0000000	Null
163719	MAC000008	Std	2012-12-19 12:37:27.0000000	Null
183152	MAC000009	Std	2012-12-19 12:37:27.0000000	Null
208192	MAC000010	Std	2012-12-19 12:37:27.0000000	Null
231899	MAC000011	Std	2012-12-19 12:37:28.0000000	Null
256569	MAC000012	Std	2012-12-19 12:37:28.0000000	Null
286182	MAC000013	Std	2012-12-19 12:37:27.0000000	Null
325260	MAC000016	Std	2012-12-18 15:13:40.0000000	Null
344744	MAC000018	Std	2012-12-18 15:13:41.0000000	Null
383815	MAC000019	Std	2012-12-18 15:13:41.0000000	Null
422896	MAC000020	Std	2012-12-18 15:13:41.0000000	Null
461974	MAC000021	Std	2012-12-18 15:13:41.0000000	Null
501052	MAC000022	Std	2012-12-18 15:13:41.0000000	Null
540115	MAC000023	Std	2012-12-18 15:13:41.0000000	Null
579142	MAC000024	Std	2012-12-18 15:13:41.0000000	Null
618213	MAC000025	Std	2012-12-18 15:13:41.0000000	Null
657284	MAC000026	Std	2012-12-18 15:13:41.0000000	Null
696349	MAC000027	Std	2012-12-18 15:13:41.0000000	Null
735416	MAC000028	Std	2012-12-18 15:13:42.0000000	Null
767574	MAC000029	Std	2012-12-18 15:13:42.0000000	Null
806639	MAC000030	Std	2012-12-18 15:13:42.0000000	Null
845699	MAC000032	Std	2012-12-18 15:13:42.0000000	Null
884771	MAC000033	Std	2012-12-18 15:13:42.0000000	Null
923843	MAC000034	Std	2012-12-18 15:13:42.0000000	Null
962865	MAC000035	Std	2012-12-18 15:13:42.0000000	Null

Let's remove those `"Null"` values.

In [8]:
```
def remove_nulls(df):
    output = df.copy()
    output.loc[:, 'kWh'] = pd.to_numeric(output['kWh'], errors='coerce')
    return output.dropna(subset=['kWh'])
```

In [9]:
```
test_data_no_nulls_ddf = test_data_ddf.map_partitions(remove_nulls)
```

Notice that nothing has happened yet. Dask methods are generally "lazy" in that they only run when needed. To execute we need to call `compute`. This means we can chain together lots of methods, and then run them all at once.

In [10]:
```
test_data_no_nulls = test_data_no_nulls_ddf.compute()
test_data_no_nulls
```

Out[10]:

	Household ID	Tariff Type	DateTime	kWh
0	MAC000002	Std	2012-10-12 00:30:00.0000000	0.000
1	MAC000002	Std	2012-10-12 01:00:00.0000000	0.000
2	MAC000002	Std	2012-10-12 01:30:00.0000000	0.000
3	MAC000002	Std	2012-10-12 02:00:00.0000000	0.000
4	MAC000002	Std	2012-10-12 02:30:00.0000000	0.000
...	...	...	...	...
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	0.070
999999	MAC000036	Std	2012-11-08 10:00:00.0000000	0.005

999971 rows × 4 columns

That's worked as we now have one less row in our test data (9,999).

We also need to remove duplicates.

In [11]:
```
test_data_cleaned_ddf = test_data_no_nulls_ddf.drop_duplicates(
    subset=['Household ID', 'Tariff Type', 'DateTime']
)
test_data_cleaned_ddf.compute()
```

Out[11]:

	Household ID	Tariff Type	DateTime	kWh
0	MAC000002	Std	2012-10-12 00:30:00.0000000	0.000
1	MAC000002	Std	2012-10-12 01:00:00.0000000	0.000
2	MAC000002	Std	2012-10-12 01:30:00.0000000	0.000
3	MAC000002	Std	2012-10-12 02:00:00.0000000	0.000
4	MAC000002	Std	2012-10-12 02:30:00.0000000	0.000
...	...	...	...	...
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	0.070
999999	MAC000036	Std	2012-11-08 10:00:00.0000000	0.005

999283 rows × 4 columns

Some duplicates have been removed - we have fewer rows in the data now.

Aggregating the 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.

Aggregation is simple when using Dask, as the `groupby` function works across the partitions. However first we need to convert the timestamp data into date format so that we can group by date. To do this we use the Dask `map_partitions` method, which is similar to Pandas `map` but is applied across all partitions. One important difference though is that we need to specify the output types using the `meta` parameter.

In [12]:
```
def timestamp_to_date(df):
    df.loc[:, 'DateTime'] = pd.to_datetime(df['DateTime']).dt.date
    return df
```

In [13]:
```
meta = {
    'Household ID' : object,
    'Tariff Type' : object,
    'DateTime' : object,
    'kWh' : float
}
```

In [14]:
```
test_data_by_date_ddf = (
    test_data_cleaned_ddf.map_partitions(timestamp_to_date, meta=meta)
    .rename(columns={'DateTime' : 'Date'})
)
```

In [15]:
```
test_data_by_date = test_data_by_date_ddf.compute()
test_data_by_date
```

Out[15]:

	Household ID	Tariff Type	Date	kWh
0	MAC000002	Std	2012-10-12	0.000
1	MAC000002	Std	2012-10-12	0.000
2	MAC000002	Std	2012-10-12	0.000
3	MAC000002	Std	2012-10-12	0.000
4	MAC000002	Std	2012-10-12	0.000
...	...	...	...	...
999995	MAC000036	Std	2012-11-08	0.228
999996	MAC000036	Std	2012-11-08	0.042
999997	MAC000036	Std	2012-11-08	0.076
999998	MAC000036	Std	2012-11-08	0.070
999999	MAC000036	Std	2012-11-08	0.005

999283 rows × 4 columns

Now we can aggregate by day.

In [16]:
```
test_summary_daily_ddf = test_data_by_date_ddf.groupby(['Household ID', 'Tariff Type',
'Date']).sum()
```

In [17]:
```
test_summary_daily = test_summary_daily_ddf.compute()
test_summary_daily
```

Out[17]:

Household ID	Tariff Type	Date	kWh
MAC000002	**Std**	**2012-10-12**	7.098
		2012-10-13	11.087
		2012-10-14	13.223
		2012-10-15	10.257
		2012-10-16	9.769
...	**...**	**...**	...
MAC000036	**Std**	**2012-11-04**	2.401
		2012-11-05	2.379
		2012-11-06	2.352
		2012-11-07	2.599
		2012-11-08	0.689

20870 rows × 1 columns

Let's tidy this all up into a single function.

In [18]:
```
def process_data(ddf):

    data_no_nulls_ddf = ddf.map_partitions(remove_nulls)

    data_deduped_ddf = data_no_nulls_ddf.drop_duplicates(
        subset=['Household ID', 'Tariff Type', 'DateTime']
    )

    data_by_date_ddf = (
        data_deduped_ddf.map_partitions(timestamp_to_date, meta=meta)
        .rename(columns={'DateTime' : 'Date'})
    )

    data_summary_daily_ddf = data_by_date_ddf.groupby(['Household ID', 'Tariff Type',
'Date']).sum()

    return data_summary_daily_ddf
```

In [19]:
```
test_summary_daily_ddf = process_data(test_data_ddf)
test_summary_daily_ddf.compute()
```

Out[19]:

			kWh
Household ID	**Tariff Type**	**Date**	
MAC000002	**Std**	**2012-10-12**	7.098
		2012-10-13	11.087
		2012-10-14	13.223
		2012-10-15	10.257
		2012-10-16	9.769
...	**...**	**...**	...
MAC000036	**Std**	**2012-11-04**	2.401
		2012-11-05	2.379
		2012-11-06	2.352
		2012-11-07	2.599
		2012-11-08	0.689

20870 rows × 1 columns

Reducing memory load

The idea now would be to process the whole data like so:

```
daily_summary = process_data(full_data_ddf)
daily_summary.compute()
```

But that doesn't work, because even with Dask distributing the load across the 4 CPUs of my laptops, we hit Out of memory errors during the deduplication.

My solution was to split the data into chunks and process them in turn, combining the aggregate results at the end. However for deduplication to work the data has to be divided so that there cannot be duplicates between chunks, only within each chunk. So I decided to split by groups of household ID.

First we decide how many chunks we want, then we get the highest Household ID and use that to work out the breakpoints.

In [20]:
```
num_divisions = 2
```

In [21]:
```
max_household_id = test_data_ddf['Household ID'].str[3:].astype('int64').max().compute()
```

In [22]:
```
max_household_id
```

Out[22]: 36

In [23]:
```
def get_splits(max_household_id, num_divisions):
    interval = max_household_id // num_divisions
    splits = np.array(range(num_divisions)) * interval
    return np.append(splits, max_household_id)
```

In [24]:
```
splits = get_splits(max_household_id, num_divisions)
splits
```

Out[24]:
```
array([ 0, 18, 36])
```

In [25]:
```
def batch_process_data(splits, data_ddf):

    # Start our progress indicator.
    print(f"Splits processed of {len(splits) - 1}: ", end="")

    # Loop through each chunk.
    for i in range(1, len(splits)):

        # Extract all data corresponding to the household IDs in this chunk.
        # The .str[3:] removes the 'MAC' part of the household ID, then .astype('int64')
converts to an integer.
        data_partition_ddf = data_ddf[
            (data_ddf['Household ID'].str[3:].astype('int64') > splits[i - 1]) &
            (data_ddf['Household ID'].str[3:].astype('int64') <= splits[i])
        ]

        # Calculate the summary daily totals for the chunk.
        summary_daily_partition_ddf = process_data(data_partition_ddf)
        summary_daily_partition = summary_daily_partition_ddf.compute()

        # Combine the summary data for this chunk with the summary data for all the preceding
chunks.
        if i == 1:
            output = summary_daily_partition
        else:
            output = pd.concat([output, summary_daily_partition])

        # Update the progress indicator.
        print(i, end=", ")

    return output
```

In [26]:
```
combined_test_summary_daily_data = batch_process_data(splits, test_data_ddf)
```

```
Splits processed of 2: 1, 2,
```

In [27]:
```
combined_test_summary_daily_data
```

Out[27]:

			kWh
Household ID	**Tariff Type**	**Date**	
MAC000002	**Std**	**2012-10-12**	7.098
		2012-10-13	11.087
		2012-10-14	13.223
		2012-10-15	10.257
		2012-10-16	9.769
...	**...**	**...**	...
MAC000036	**Std**	**2012-11-04**	2.401
		2012-11-05	2.379
		2012-11-06	2.352
		2012-11-07	2.599
		2012-11-08	0.689

20870 rows × 1 columns

That seems to work as we get the same results on the test data.

Operating on the full data

Dask client

We'll start a Dask Client which is generally used for interacting with a cluster, but it's also useful on a single machine as it shows progress during an operation.

In [28]:
```
from dask.distributed import Client
client = Client()
client
```

Out[28]:

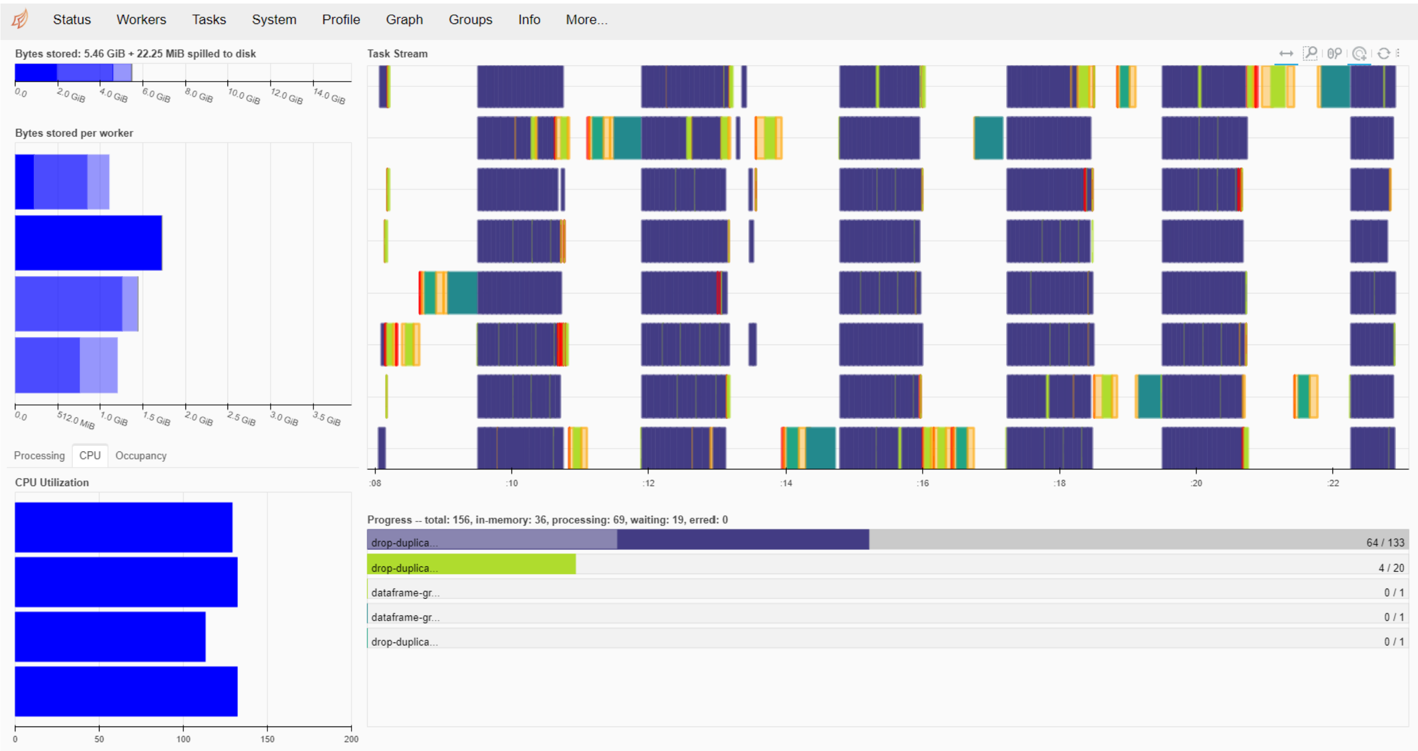
Client

Client-a90704e9-eb15-11ec-8d3c-e45e37a84c64

Connection method: Cluster object	**Cluster type:** distributed.LocalCluster
Dashboard: http://127.0.0.1:8787/status	

▸ **Cluster Info**

We can click on the dashboard link above and open as a new window. Then when a `compute` is executed we can watch the progress on the dashboard in the client window.



At the bottom right we can see the progress bar - very handy! We can also see at the bottom left that all 4 of my CPUs are in use, and the 8 task streams (upper right) represent the 8 logical CPUs (2 per physical CPU).

Obviously the tasks complete much more quickly than when using a non-distributed approach (like Pandas chunksize for example).

Full data execution

In testing I found that the fewer the chunks the faster the process, but I couldn't use less than about 10 chunks on my laptop without hitting memory errors.

In [29]:
```
num_divisions_full_data = 10
```

In [30]:
```
max_household_id_full_data = full_data_ddf['Household
ID'].str[3:].astype('int64').max().compute()
```

In [31]:
```
max_household_id_full_data
```

Out[31]:
```
5567
```

In [32]:
```
splits_full_data = get_splits(max_household_id_full_data, num_divisions_full_data)
splits_full_data
```

Out[32]:
```
array([   0,  556, 1112, 1668, 2224, 2780, 3336, 3892, 4448, 5004, 5567])
```

In [33]:
```
daily_summary = batch_process_data(splits_full_data, full_data_ddf)
```

```
Splits processed of 10: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

In [34]:
```
daily_summary
```

Out[34]:

			kWh
Household ID	**Tariff Type**	**Date**	
MAC000002	**Std**	**2012-10-12**	7.098
		2012-10-13	11.087
		2012-10-14	13.223
		2012-10-15	10.257
		2012-10-16	9.769
...	**...**	**...**	...
MAC005567	**Std**	**2014-02-24**	4.107
		2014-02-25	5.762
		2014-02-26	5.066
		2014-02-27	3.217
		2014-02-28	0.183

3510403 rows × 1 columns

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

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 [35]:
```
daily_summary.to_csv("data/daily-summary-data.gz")
```

Analysing the data

In [36]:
```
saved_daily_summary = pd.read_csv("data/daily-summary-data.gz")
```

In [37]:
```
saved_daily_summary
```

Out[37]:

	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

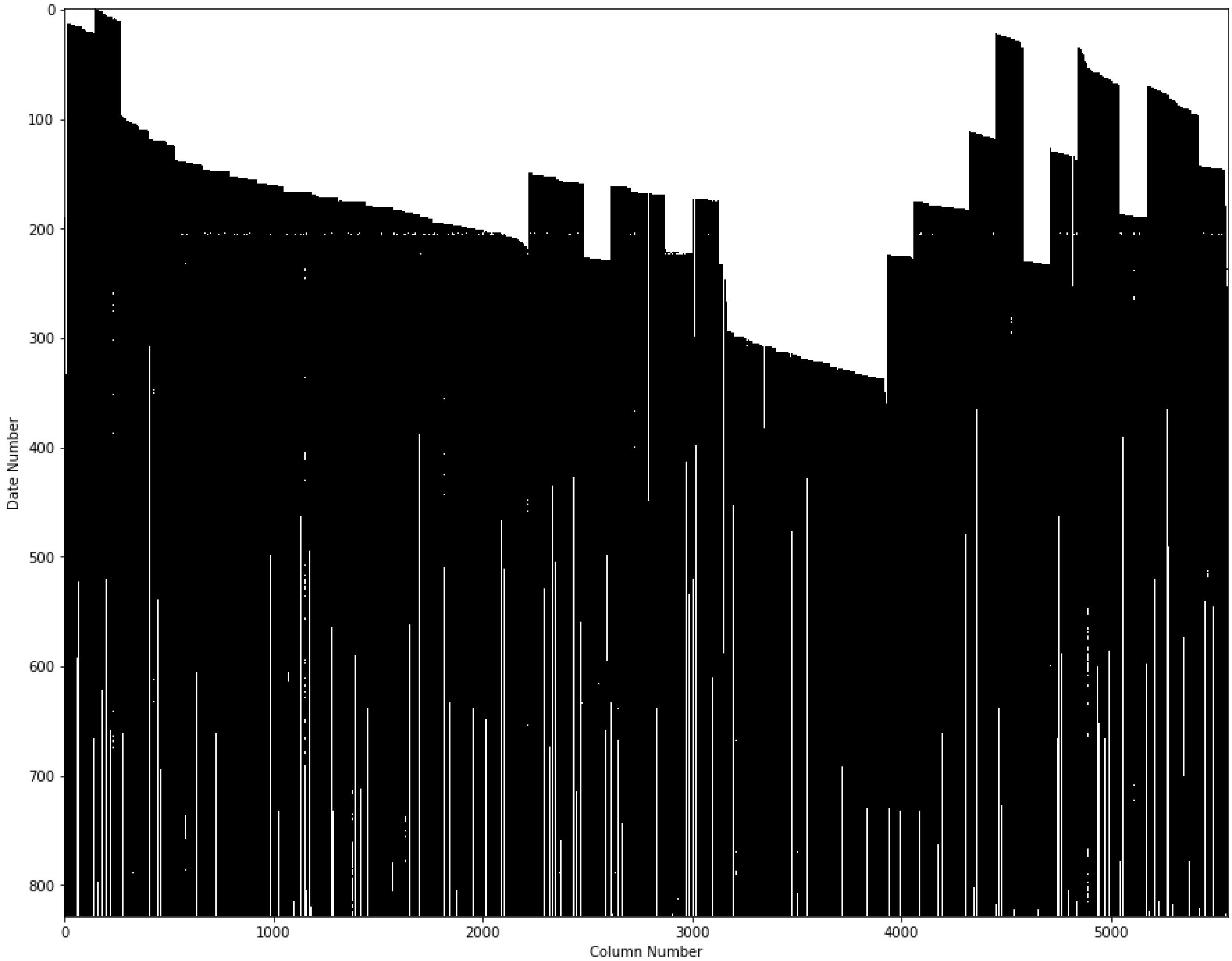
Out of interest let's see what sort of data coverage we have. First we re-organize so that we have households as columns and dates as rows.

In [38]:
```
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 [39]:
```
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 [40]:
```
daily_mean_by_tariff_type = saved_daily_summary.pivot_table(
    'kWh',
    index='Date',
    columns='Tariff Type',
    aggfunc='mean'
)
daily_mean_by_tariff_type
```

Out[40]:

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

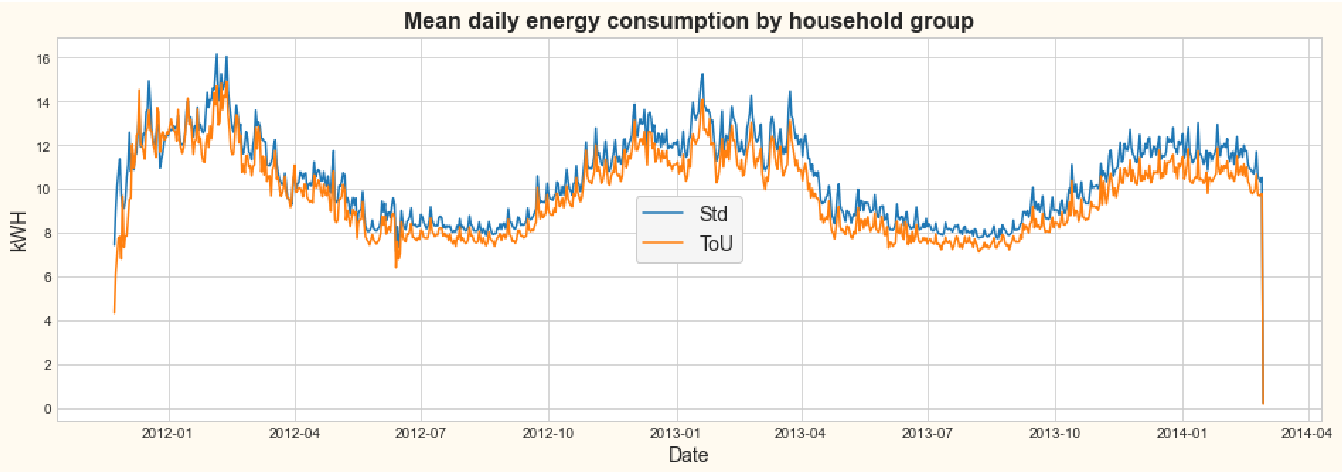
Finally we can plot the two sets of data. The plotting works better if we convert the date from type `string` to type `datetime`.

In [41]:
```
daily_mean_by_tariff_type.index = pd.to_datetime(daily_mean_by_tariff_type.index)
```

In [42]:
```
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 [43]:
```
daily_mean_by_tariff_type
```

Out[43]:

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 [44]:
```
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[44]:

Tariff Type	Std	ToU
Date		
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

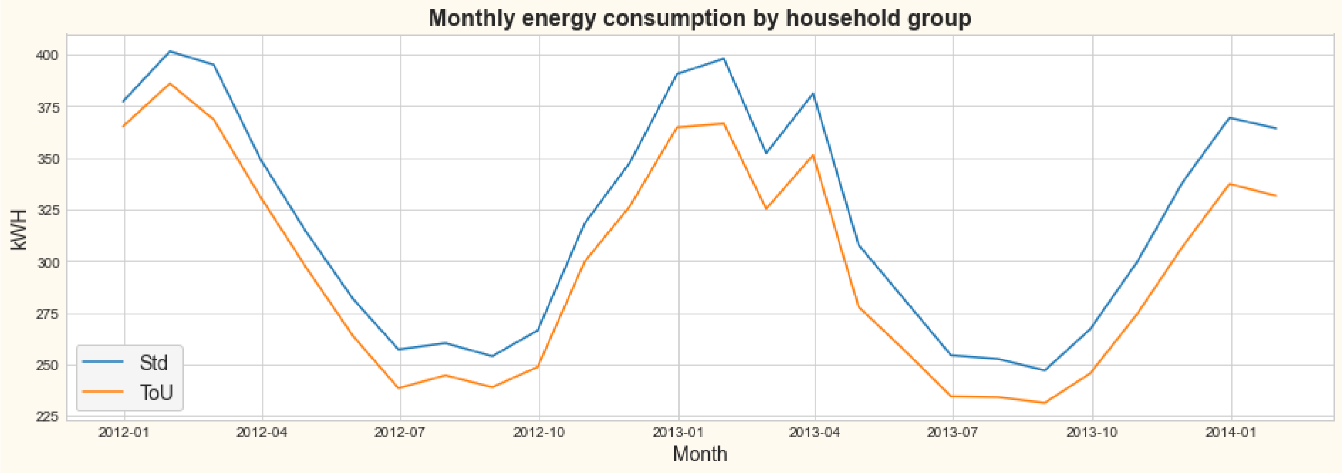
In [45]:

```
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.

Lastly we close the Dask client although it will automatically close when our Python session ends.

In [46]:
```
client.close()
```