

Introduction

Ce notebook est le **deuxième** d'une série où je regarde des grands jeux de données, et dans chaque cas j'utilise un outil différent pour effectuer la même analyse sur le même jeu de données.

Cette fois-ci j'utilise la **bibliothèque Dask** qui est adaptée au traitement parallèle pour traiter un fichier de grande taille. On peut trouver chaque notebook dans la série dans mon [répertoire Github](#), y compris:

1. Pandas chunksize
2. Bibliothèque Dask

Il y a un peu plus d'explication dans le premier notebook (Pandas chunksize) par rapport à l'approche générale de l'analyse. Dans les autres notebooks je me concentre plus sur les éléments spécifiques à l'outil que j'utilise.

Description du jeu de données

On se servira du jeu de données des [Données de consommation d'énergie des résidences muni de SmartMeter à Londres](#), qui contient, selon le site web:

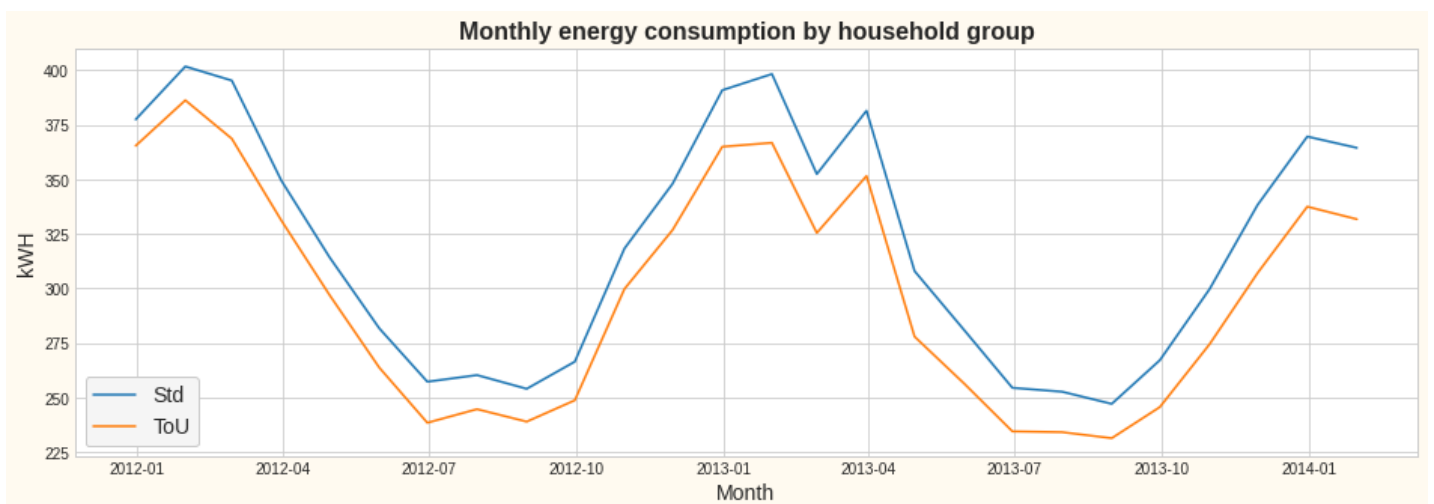
Des relevés de consommation d'énergie pour un échantillon de 5 567 résidences à Londres qui ont participé au projet de Low Carbon London (géré par UK Power Networks) entre novembre 2011 et février 2014.

Les résidences étaient divisées en deux groupes:

- Celles qui ont reçu des tarifs d'énergie Dynamic Time of Use (dTou) (décrit "Haut", "Moyen", ou "Bas") la veille du jour où le prix allait être appliqué.
- Celles qui étaient soumises au tarif Standard.

Un but du projet était d'évaluer si la connaissance du prix de l'énergie changerait le comportement par rapport à la consommation d'énergie.

Résultats



Les résultats montrent la variation saisonnière attendue et une différence nette entre les deux groupes, qui suggère qu'une connaissance du prix d'énergie aide à réduire la consommation de l'énergie.

Le reste du notebook montre comment le diagramme était produit des données brutes.

Introduction à Dask

Selon la documentation:

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.

Cela veut dire que non seulement on peut traiter des fichiers de taille plus grande que la mémoire, mais contrairement à l'approche de pandas chunksize, on peut aussi se servir d'un cluster de serveurs - ou de plusieurs cœurs en utilisant un seul ordinateur.

Accéder les données

On peut télécharger les données sous forme de fichier zip qui contient un fichier csv de 167 million lignes. Si la commande `curl` ne fonctionne pas (il faudra un certains temps puisque c'est un fichier de 800MB), vous pouvez télécharger le fichier [ici](#) et le mettre dans le dossier `data` qui se trouve dans le dossier où ce notebook est sauvegardé.

```
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"
```

Ensuite on décompresse les données. Il faudra peut-être un certain temps! Vous pouvez également le décompresser manuellement en utilisant un autre logiciel de décompression. Assurez-vous simplement que vous mettez le fichier décompressé dans un dossier qui s'appelle `data` dans le dossier où votre notebook est sauvegardé.

```
In [1]: !unzip "data/LCL-FullData.zip" -d "data"
Archive:  data/LCL-FullData.zip
  inflating: data/CC_LCL-FullData.csv
```

Examiner les données

```
In [2]: import pandas as pd
        from dask import dataframe as dd
```

Maintenant chargeons les données dans un dataframe Dask.

```
In [3]: raw_data_ddf = dd.read_csv('data/CC_LCL-FullData.csv')
raw_data_ddf
```

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

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

...

Dask Name: read-csv, 133 tasks

On peut voir que Dask a divisé nos données en 133 partitions. Dask a aussi "deviné" les types des données en examinant un échantillon des données. Laisser le dataframe tel quel entraînera des erreurs, parce que les données de kWh consiste en un mélange de valeurs numériques et de valeurs 'Null' de type chaîne.

Comme première étape on peut préciser le type de données kWh, en utilisant `object` pour permettre les chaînes.

```
In [4]: raw_data_ddf = dd.read_csv(
        'data/CC_LCL-FullData.csv',
        dtype={'KWH/hh (per half hour)': 'object'})
raw_data_ddf
```

Out[4]: **Dask DataFrame Structure:**

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

...

Dask Name: read-csv, 133 tasks

On renomme les colonnes pour les rendre plus lisibles.

```
In [5]: col_renaming = {
        'LCLid' : 'Household ID',
        'stdorToU' : 'Tariff Type',
        'KWH/hh (per half hour)' : 'kWh'
    }
full_data_ddf = raw_data_ddf.rename(columns=col_renaming)
```

Travillons sur une petite partie du jeu de données (10 000 lignes) pour créer et tester chaque étape de traitement.

```
In [6]: test_data = full_data_ddf.head(10000)
test_data
```

```
Out[6]:
```

	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
...
9995	MAC000002	Std	2013-05-09 03:30:00.0000000	0.09
9996	MAC000002	Std	2013-05-09 04:00:00.0000000	0.114
9997	MAC000002	Std	2013-05-09 04:30:00.0000000	0.105
9998	MAC000002	Std	2013-05-09 05:00:00.0000000	0.099
9999	MAC000002	Std	2013-05-09 05:30:00.0000000	0.117

10000 rows × 4 columns

Il faut reconverter ces données en dataframe Dask. On divise le dataframe en 2 partitions pour être sûr qu'on teste sur plus qu'une seule partition.

```
In [7]: test_data_ddf = dd.from_pandas(test_data, npartitions=2)
test_data_ddf
```

Out[7]: **Dask DataFrame Structure:**

	Household ID	Tariff Type	DateTime	kWh
npartitions=2				
0	object	object	object	object
5000
9999

Dask Name: from_pandas, 2 tasks

Nettoyer les données

On voit qu'il y a au moins une valeur "Null" dans les données kWh puisqu'il y a un exemple dans notre dataset test.

```
In [8]: test_nulls = test_data[test_data['kWh'] == 'Null']
test_nulls
```

```
Out[8]:
```

	Household ID	Tariff Type	DateTime	kWh
3240	MAC000002	Std	2012-12-19 12:37:27.0000000	Null

Supprimons ces valeurs "Null".

```
In [9]: def remove_nulls(df):
output = df.copy()
output.loc[:, 'kWh'] = pd.to_numeric(output['kWh'], errors='coerce')
return output.dropna(subset=['kWh'])
```

```
In [10]: test_data_no_nulls_ddf = test_data_ddf.map_partitions(remove_nulls)
```

Veuillez noter que rien ne s'est passé encore. Les méthodes de Dask sont "paresseuses" en général, qui veut dire qu'elles ne sont exécutées que lorsqu'il est nécessaire. Pour exécuter il faut appeler `compute`. Ca veut dire qu'on peut enchaîner plusieurs méthodes, et ensuite les exécuter toutes ensemble.

```
In [11]: test_data_no_nulls = test_data_no_nulls_ddf.compute()
test_data_no_nulls
```

```
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
...
9995	MAC000002	Std	2013-05-09 03:30:00.0000000	0.090
9996	MAC000002	Std	2013-05-09 04:00:00.0000000	0.114
9997	MAC000002	Std	2013-05-09 04:30:00.0000000	0.105
9998	MAC000002	Std	2013-05-09 05:00:00.0000000	0.099
9999	MAC000002	Std	2013-05-09 05:30:00.0000000	0.117

9999 rows × 4 columns

Notre traitement a marché car maintenant on a une ligne de moins dans notre jeu de données (9 999).

Réduire les données

Le but est de **réduire** les données en les agréant d'une manière ou d'une autre. Puisque nous savons que les données sont organisées par demi-heure, on va les agréger par jour en les additionnant sur chaque période de 24 heures. Cela devrait réduire le nombre de lignes par un facteur d' environ 48.

L'agrégation est simple en utilisant Dask, car la fonction `groupby` fonctionne sur toutes les partitions d'un seul coup. Pourtant il faut d'abord convertir les valeurs de type horodateur en format de date pour qu'on puisse les grouper par date. Pour faire ceci on utilise la méthode Dask `map_partitions`, qui est semblable à `map` de Pandas mais qui est appliquée à toutes les partitions. Une différence importante à noter pourtant - il faut préciser les types des données de sortie en utilisant le paramètre `meta`.

```
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_no_nulls_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
...
9995	MAC000002	Std	2013-05-09	0.090
9996	MAC000002	Std	2013-05-09	0.114
9997	MAC000002	Std	2013-05-09	0.105
9998	MAC000002	Std	2013-05-09	0.099
9999	MAC000002	Std	2013-05-09	0.117

9999 rows × 4 columns

Maintenant on peut agréger par jour.

```
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
		
			2013-05-05	8.826
			2013-05-06	7.418
			2013-05-07	7.607
			2013-05-08	8.576
			2013-05-09	1.567

210 rows × 1 columns

Traiter le jeu de données complet

Maintenant on peut appliquer les mêmes méthodes aux données complètes. Veuillez noter que rien ne va se passer jusqu'à ce qu'on appelle `compute`.


```
In [18]: full_data_no_nulls_ddf = full_data_ddf.map_partitions(remove_nulls)
```

```
In [19]: full_data_by_date_ddf = (  
    full_data_no_nulls_ddf.map_partitions(timestamp_to_date, meta=meta)  
    .rename(columns={'DateTime' : 'Date'})  
)
```

```
In [20]: full_summary_daily_ddf = full_data_by_date_ddf.groupby(['Household ID', 'Tariff  
Type', 'Date']).sum()
```

On va lancer un Client Dask qu'on utilise en général pour gérer un cluster, mais il est également utile sur un seul ordi parce qu'il affiche le progrès pendant une opération.

```
In [21]: from dask.distributed import Client  
client = Client()  
client
```

Out[21]:  **Client**
Client-e7973761-ddc4-11ec-bccd-4b8caf26a57e

Connection method: Cluster object

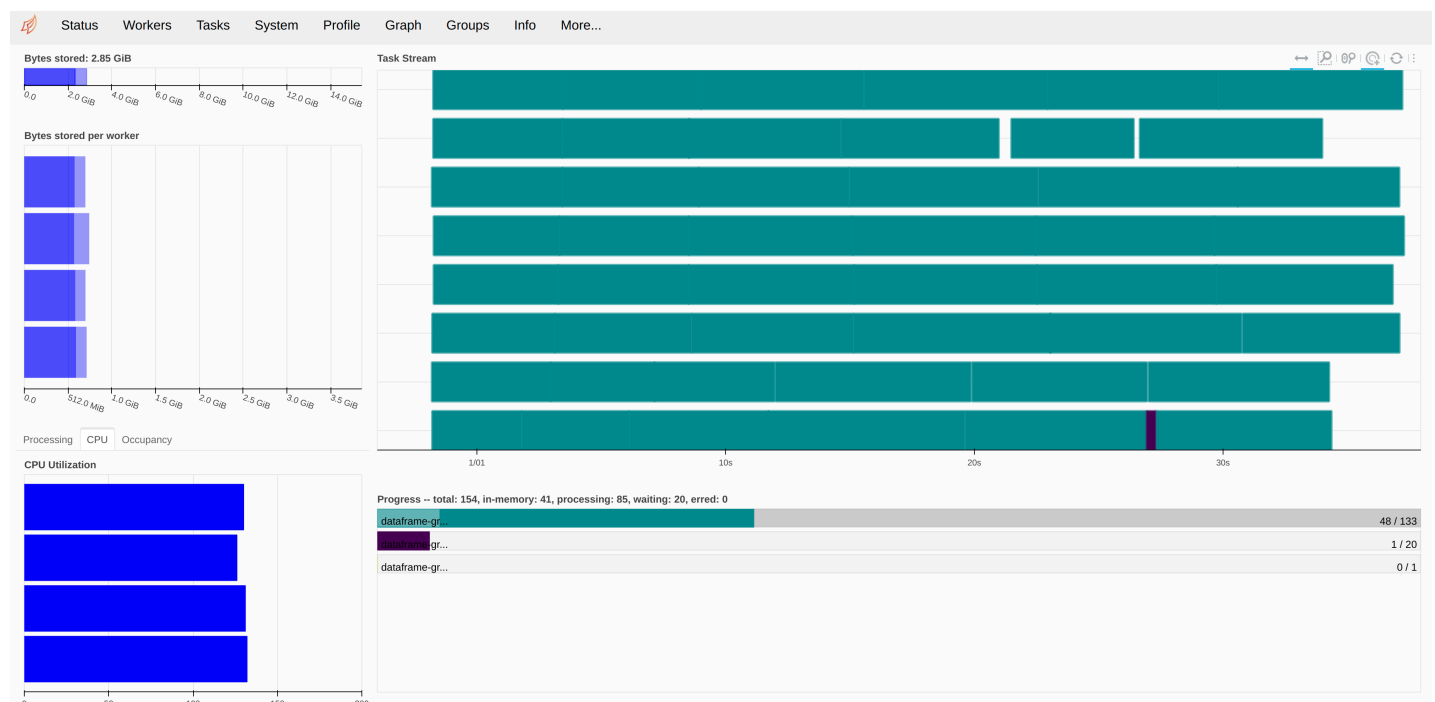
Cluster type: distributed.LocalCluster

Dashboard: <http://127.0.0.1:8787/status>

► Cluster Info

On peut cliquer sur le lien "Dashboard" ci-dessus qui va ouvrir une nouvelle fenêtre. Ensuite on peut exécuter `compute` et observer le progrès dans la fenêtre du client.

```
In [22]: daily_summary = full_summary_daily_ddf.compute()
```



En bas à droite on voit la barre de progrès - très utile! On voit aussi en bas à gauche que tous mes CPUs sont en usage, et les 8 task streams (en haut à droite) représentent les 8 CPUs logiques (2 par CPU physique).

Evidemment les opérations exécutent beaucoup plus vite qu'une approche qui n'utilise qu'un seul coeur (comme Pandas chunksize par exemple).

```
In [23]: daily_summary
```

```
Out[23]:
```

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
...
MAC005564	ToU	2014-02-26	3.431
		2014-02-27	4.235
		2014-02-28	0.122
MAC005565	ToU	2012-06-20	3.896
		2012-06-21	1.894

3510403 rows × 4 columns

A partir d'ici, le reste de ce notebook contient à peu près le même traitement que tous les autres notebooks dans la série.

Sauvegarder les données agrégées

Maintenant qu'on a ramené les données à environ 3 millions lignes on devrait pouvoir les contenir dans un seul dataframe. Il vaut mieux les sauvegarder pour qu'on n'ait pas besoin de réexécuter l'agrégation chaque fois qu'on veut traiter les données.

On va le sauvegarder comme fichier compressé gz - pandas reconnaît automatiquement le type de fichier quand on précise l'extension.

```
In [24]: daily_summary.to_csv("data/daily-summary-data.gz")
```

Analyser les données

```
In [25]: saved_daily_summary = pd.read_csv("data/daily-summary-data.gz")
saved_daily_summary
```


Out[25]:

	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	MAC005564	ToU	2014-02-26	3.431
3510399	MAC005564	ToU	2014-02-27	4.235
3510400	MAC005564	ToU	2014-02-28	0.122
3510401	MAC005565	ToU	2012-06-20	3.896
3510402	MAC005565	ToU	2012-06-21	1.894

3510403 rows × 4 columns

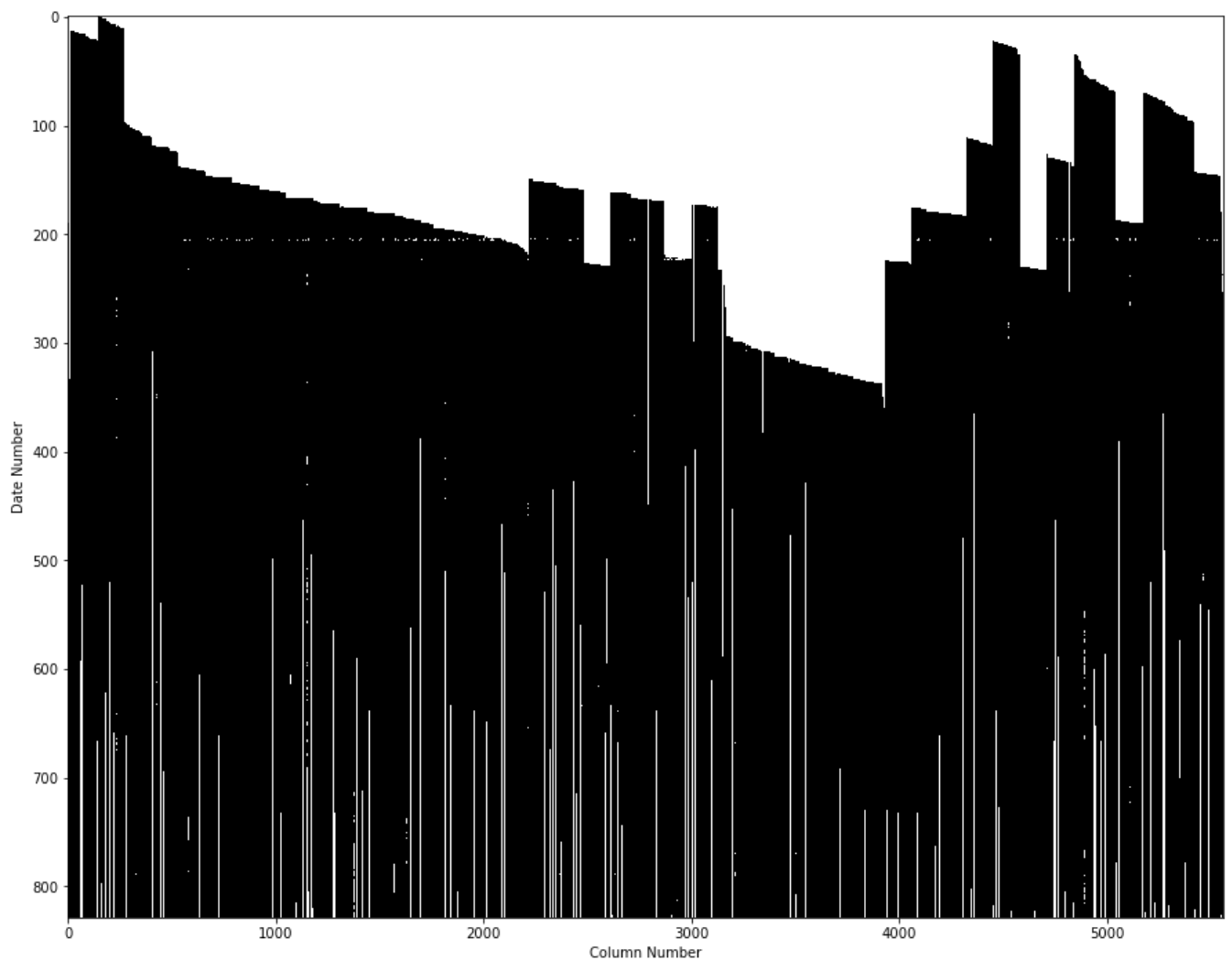
Par intérêt examinons la couverture des données. D'abord on réorganise pour avoir les résidences en colonne et les date en ligne.

```
In [26]: summary_table = saved_daily_summary.pivot_table(
    'kWh',
    index='Date',
    columns='Household ID',
    aggfunc='sum'
)
```

Ensuite on peut afficher où on a des données (noir) et où on n'en a pas (blanc).

```
In [27]: 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");
```



Malgré une couverture un peu lacunaire, calculer par tarif sur toutes les résidences par jour devrait nous donner une comparaison utile.

```
In [28]: daily_mean_by_tariff_type = saved_daily_summary.pivot_table(
    'kWh',
    index='Date',
    columns='Tariff Type',
    aggfunc='mean'
)
daily_mean_by_tariff_type.head()
```

```
Out[28]:
```

	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	

Finalement on peut tracer les deux groupes de données.

```
In [34]: plt.style.use('seaborn-whitegrid')

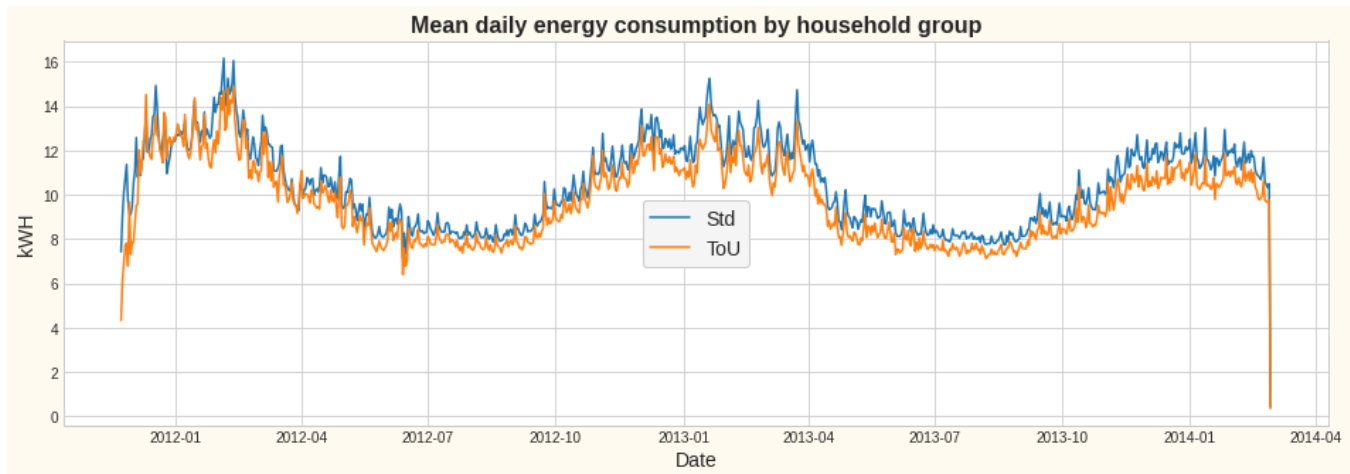
plt.figure(figsize=(16, 5), facecolor='floralwhite')
for tariff in daily_mean_by_tariff_type.columns.to_list():
```

```

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()

```



On dirait que la variation est saisonnière qui n'est pas étonnant vu la demande d'énergie de chauffage.

On dirait aussi qu'il y a une différence entre les deux groupes: le groupe ToU a l'air de consommer moins, mais l'affichage est trop granulaire pour voir bien. Agégeons encore une fois, cette fois-ci par mois.

```

In [30]: daily_mean_by_tariff_type.index = pd.to_datetime(daily_mean_by_tariff_type.index)
daily_mean_by_tariff_type.head()

```

```

Out[30]:

```

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

On voit que les données commencent au cours de novembre 2011, donc on commencera le 1 décembre. On dirait que les données terminent parfaitement à la fin de février, mais la dernière valeur est suspecte puisqu'elle est très basse comparé aux autres. Il paraît probable que les données ont terminé au cours de la dernière journée, donc on finira à la fin de janvier. Peut-être qu'on a le même problème ailleurs dans les données, mais l'effet ne devrait pas être énorme parce que dans le pire des cas la consommation mensuelle d'une résidence sera réduite par deux journées (une au début et une à la fin).

```

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

```

```
monthly_mean_by_tariff_type.head()
```

```
Out[31]:
```

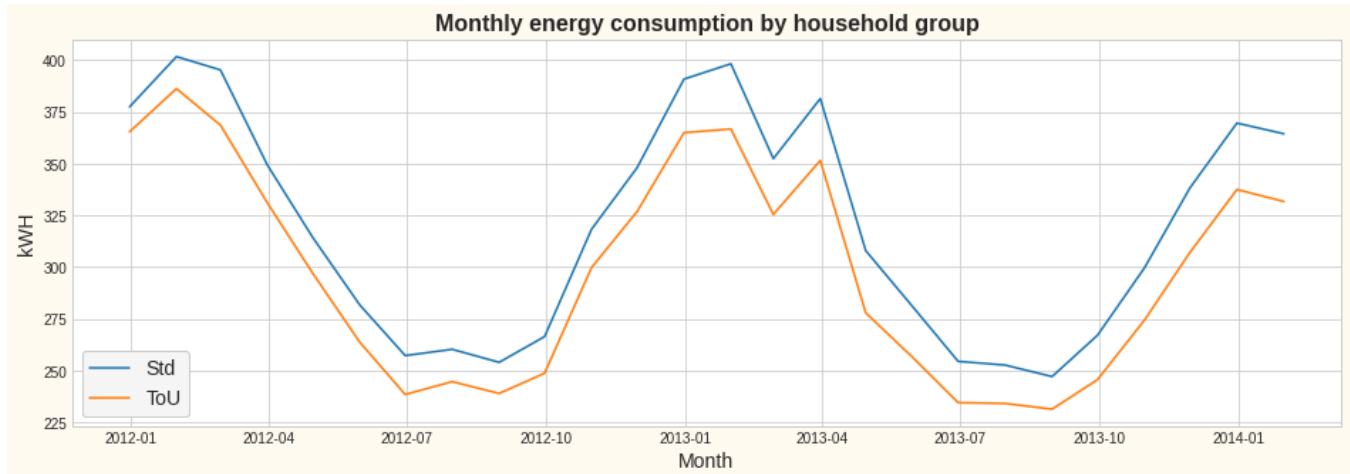
Tariff Type	Std	ToU
2011-12-31	377.443042	365.391597
2012-01-31	401.744672	386.253703
2012-02-29	395.294296	368.663764
2012-03-31	349.367317	331.095386
2012-04-30	314.323216	297.032370

```
In [32]: 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.png', bbox_inches='tight')

plt.show()
```



Le diagramme est plus clair et il y a une différence évidente entre les deux groupes.

Veuillez noter que le diagramme ne montre pas la consommation mensuelle moyenne. Il montre la somme des moyennes journalières pour chaque mois. Pour calculer les vraies moyennes mensuelles on aurait besoin d'exclure les données journalières pour chaque résidence pendant les mois où les données n'étaient pas complètes. Notre méthode plus simple devrait nous donner une bonne approximation.

Pour terminer on ferme le client Dask bien qu'il ferme automatiquement quand notre session de Python termine.

```
In [33]: client.close()
```