Introduction

Ce noteboook 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:

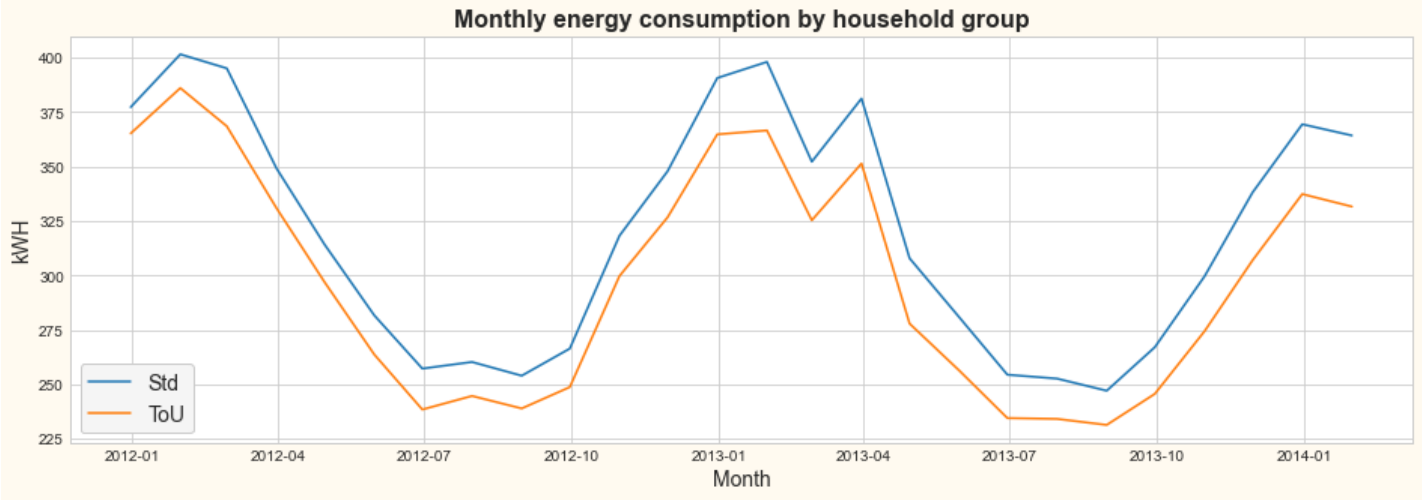
> Des relevés de consommation d'énergie pour un échantillon de 5 567 résidences à Londres qui ont participé au projet de Low Carbon London (géré par UK Power Networks) entre novembre 2011 et février 2014.

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 connaisance 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 memoire, mais contrairement à l'approche de pandas chunksize, on peut aussi se servir d'un cluster de serveurs - ou de plusieurs coeurs 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 [ ]:
```
!unzip "data/LCL-FullData.zip" -d "data"
```

Examiner les données

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

Maintenant chargeons les données dans un dataframe Dask.

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

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 valuers 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 [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

On renomme les colonnes pour les rendre plus lisibles.

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)
```

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

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

Il faut reconvertir 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 [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

Nettoyer les données

On voit qu'il y a des valeurs `"Null"` dans les données kWh.

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

Supprimons ces valeurs `"Null"`.

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)
```

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 [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

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

Nous devons aussi supprimer les doublons.

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

On a supprimé des doublons - il y a moins de lignes maintenant.

Réduire les données

Le but est de ***réduire*** les données en les agrégant 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_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

Maintenant on peur 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]:

			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

Rangeons un peu en mettant toutes les étapes dans une seule fonction.

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

Réduire la charge mémoire

Idéalement la prochaine étape serait traiter toutes les données comme ceci:

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

Mais cela ne marche pas, parce que même avec la répartition de charge parmi les 4 coeurs de mon laptop par Dask, on a des erreurs de mémoire insuffisante pendant la suppression de doublons.

Ma solution était de diviser les données en tranches, de les traiter une à une, et de combiner les résultats cumulés à la fin. Pourtant pour que la suppression de doublons fonctionne il faut diviser les données d'une manière qui garantit qu'il n'y a pas de doublons entre les tranches, seulement dedans chaque tranche. Donc j'ai décidé de diviser par des groupes de Household ID.

D'abord on décide le nombre de tranches qu'on veut, et ensuite on trouve le Household ID le plus grand et on l'utilise pour calculer les points de rupture.

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

On dirait que cela marche car on a les mêmes résultats qu'on a eu pour les données de test.

Traiter le jeu de données complet

Dask client

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

Out[28]:

Client

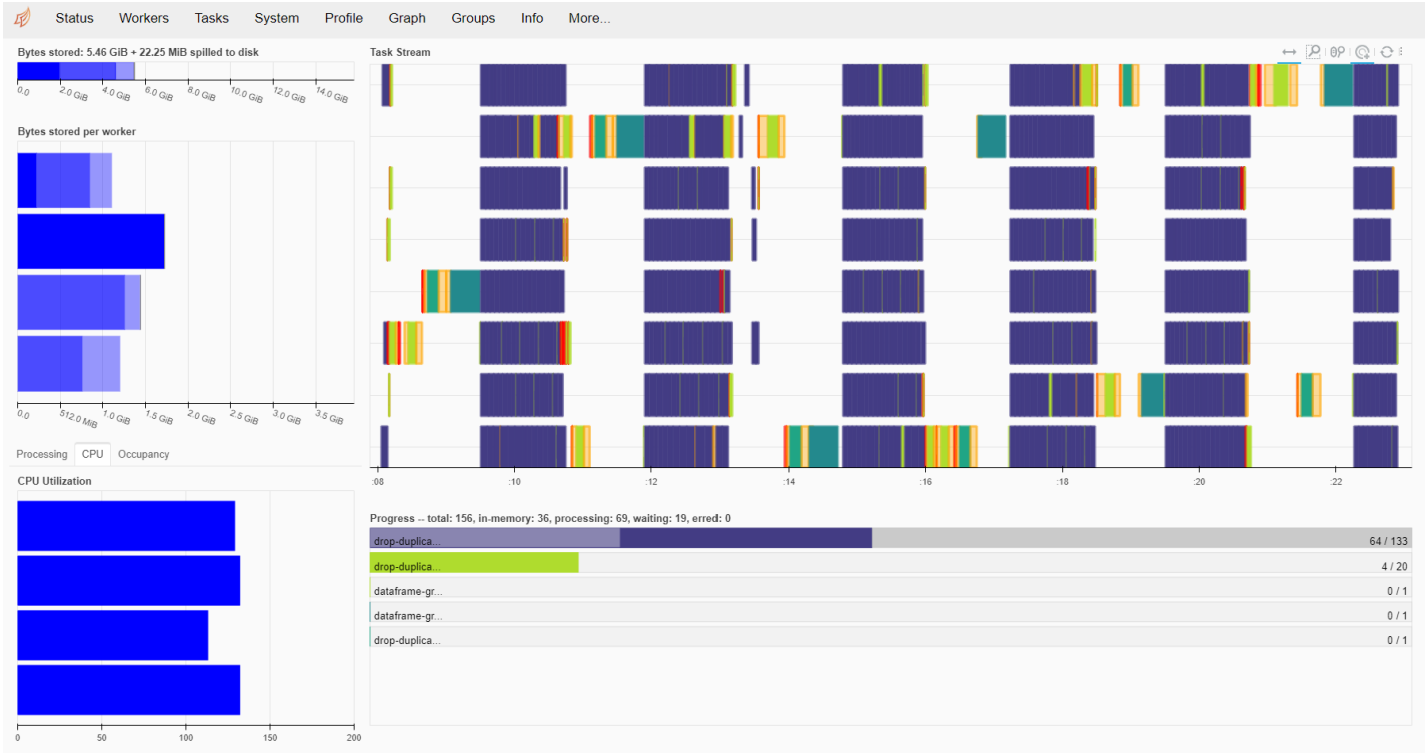
Client-a90704e9-eb15-11ec-8d3c-e45e37a84c64

Connection method: Cluster object **Cluster type:** distributed.LocalCluster

Dashboard: http://127.0.0.1:8787/status

▸ **Cluster Info**

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.



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 example).

Exécution

Quand j'ai fait des tests j'ai trouvé que le traitement était plus rapide avec moins de tranches, mais que je ne pouvais pas avoir moins de 10 tranches sans déclencher erreurs de mémoire insuffisante.

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

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

Maintentant 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 réexecuter l'agrégation chaque fois qu'on veut traiter les données.

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

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

Analyser les données

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

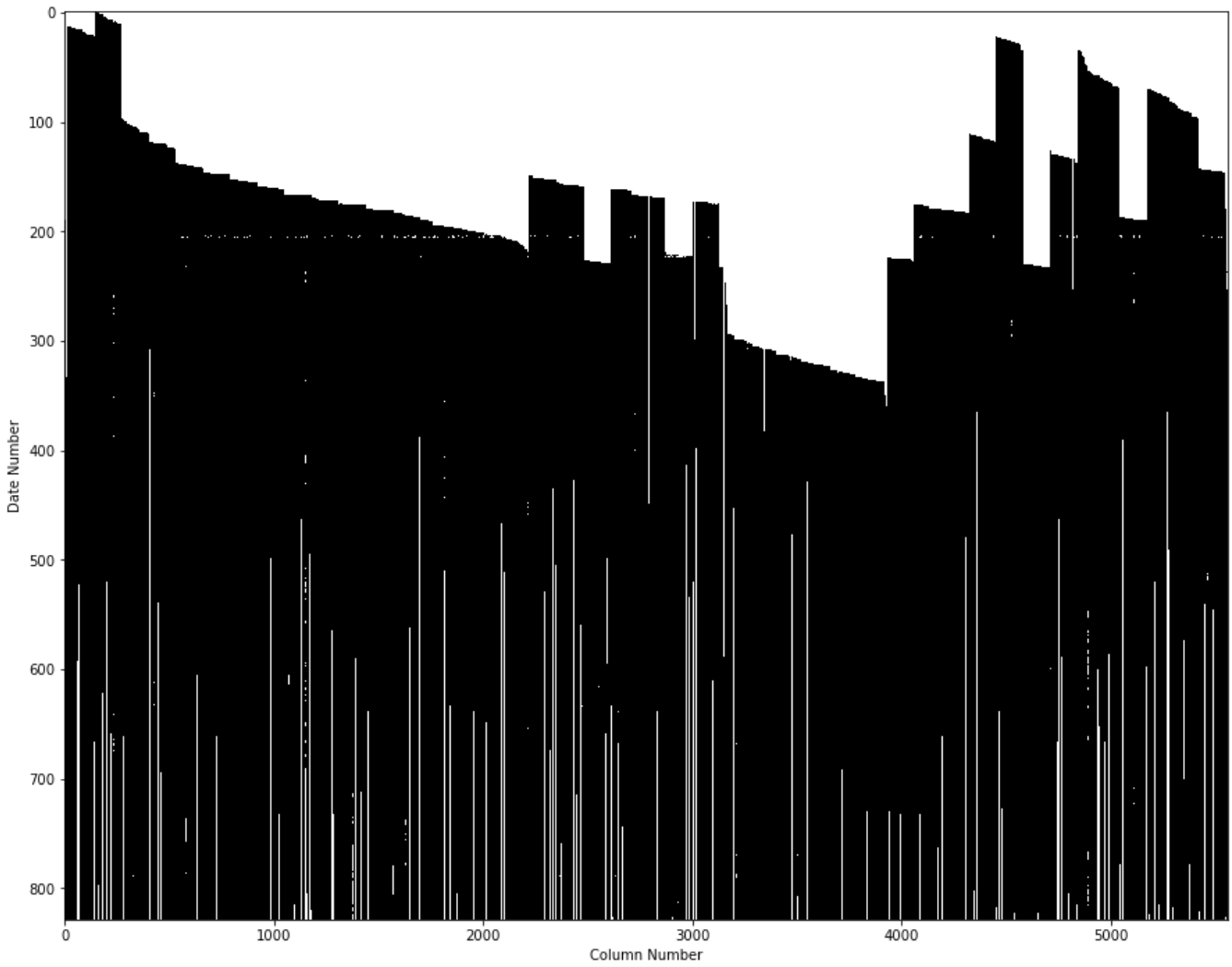
Par intérêt examinons la couverture des données. D'abord on réorganise pour avoir les résidences en colonne et les date en ligne.

In [38]:
```
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 [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");
```



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

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

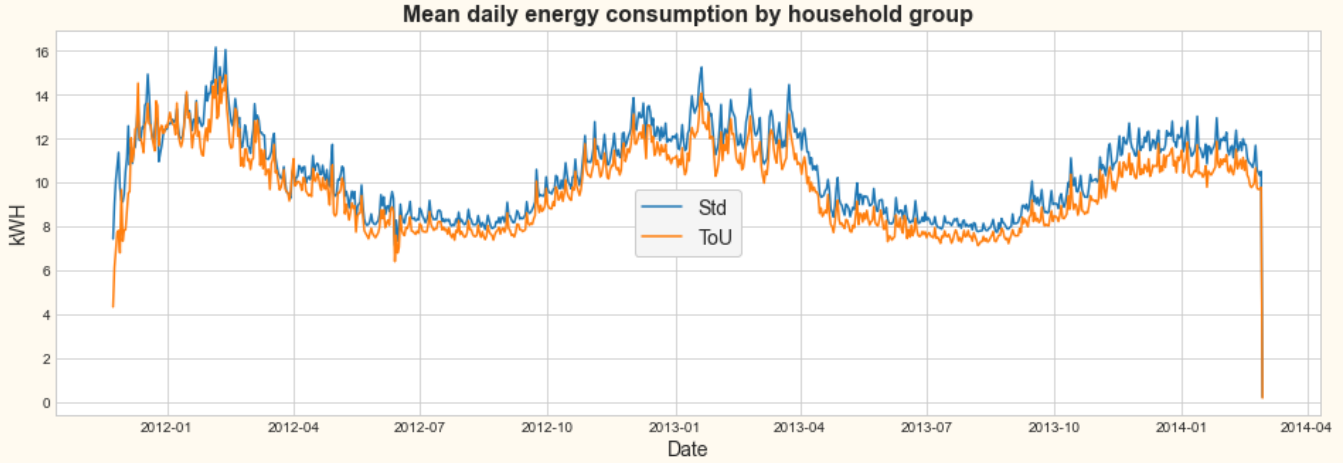
Finalement on peut tracer les deux groupes de données. Le traçage marche mieux si on convertit la date de type `string` en type `datetime`.

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()
```



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 [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

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 [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]:

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

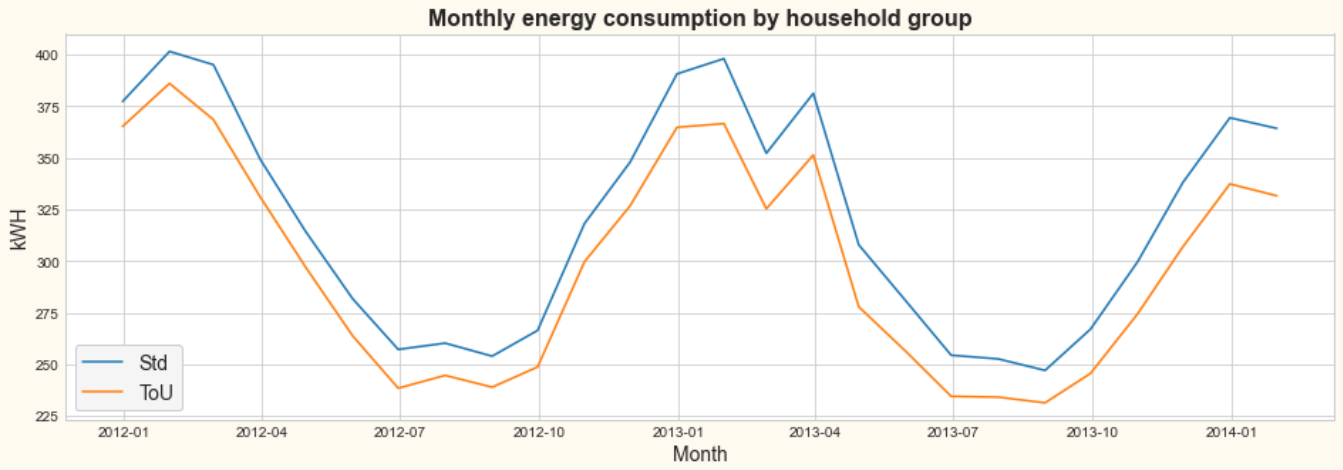
In [45]:

```
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()
```



Le diagramme est plus clair et il y a une différence evidente 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 [46]:
```
client.close()
```