

AirQuality

June 14, 2024

1 Proyecto Final: Estudio de la calidad del aire y sus efectos en la salud empleando herramientas de Big Data (HDFS, YARN, Spark)

Por Carlos Andrés González Bono

1.1 0. Presentación, contextualización e interés del caso

Según la Organización Mundial de la Salud, el estado de la atmósfera actual puede provocar, por simple acto de respirar, la muerte de alrededor de siete millones de personas al año (respiración de partículas finas), viéndose muchas más perjudicadas¹.













La contaminación del aire es una amenaza muy relevante, especialmente para las poblaciones urbanas, y mientras todas las personas están expuestas, las emisiones de contaminantes, los niveles de exposición y la vulnerabilidad de la población varía dependiendo del lugar. La exposición a contaminantes del aire han sido relacionadas con enfermedades cardiovasculares, cánceres y muertes prematuras. Estos indicadores nos dan una perspectiva a través del tiempo y de la geografía de Nueva York para poder visualizar la relación entre la calidad del aire y la salud de los ciudadanos de un entorno urbano².

Por otra parte, los sistemas de computación distribuida como HDFS permiten analizar grandes cantidades de datos distribuidos con hardware no especializado, mientras herramientas como Spark pueden sacarle partido a estos sistemas ofreciendo abstracciones de alto nivel para hacer rápidas operaciones en-memoria (in-memory). En este estudio exploraremos cómo realizar este tipo de análisis a través de este tipo de sistemas. Si bien los datos de origen de este estudio no requieren de estas capacidades (son unos datos pequeños ~2MB), la finalidad del uso de estas tecnologías es didáctica. Es decir, usaremos Hadoop/Spark para aprender a utilizar Hadoop/Spark.

1.2 1. Descripción de los datos

Los datos han sido recogidos entre 2005 y 2022 por el Departamento de Salud e Higiene Mental (DOHMH) de Nueva York, y se presentan en un archivo .csv, con [las siguientes columnas](#)³:

Columns (12)

| Column Name | Description | API Field Name | Data Type |
|--|--|----------------|------------------------------------|
|  Unique ID | Unique record identifier | unique_id | Text |
|  Indicator ID | Identifier of the type of measured value across time and space | indicator_id | Number |
|  Name | Name of the indicator | name | Text |
|  Measure | How the indicator is measured | measure | Text |
|  Measure Info | Information (such as units) about the measure | measure_info | Text |
|  Geo Type Name | Geography type; UHF' stands for United Hospital Fund neighborhoods; For instance, Citywide, Borough, and Community Districts are different geography types | geo_type_name | Text |
|  Geo Join ID | Identifier of the neighborhood geographic area, used for joining to mapping geography files to make thematic maps | geo_join_id | Text |
|  Geo Place Name | Neighborhood name | geo_place_name | Text |
|  Time Period | Description of the time that the data applies to ; Could be a year, range of years, or season for example | time_period | Text |
|  Start_Date | Date value for the start of the time_period; Always a date value; could be useful for plotting a time series | start_date | Floating Timestamp |
|  Data Value | The actual data value for this indicator, measure, place, and time | data_value | Number |
|  Message | notes that apply to the data value; For example, if an estimate is based on small numbers we will detail here | message | Text |

En nuestro caso, utilizaremos principalmente:

- Indicator ID: Cada ID corresponde a una combinación de Name, Measure y Measure Info, enlazando toda la información de la medida en una sola variable. Previamente tendré que identificar los IDs que correspondan a emisiones y presencia de químicos, y diferenciarlos de los IDs que indiquen incidencias clínicas.
- Geo Join ID: Cada ID corresponde a un área. Unas áreas pueden englobar a otras, y aunque a priori eso no tiene porqué generar un conflicto en los datos, es algo que tendré que identificar y considerar.
- Start_Date: Fecha de la medida
- Data Value: El valor de la medida.

Utilizaremos Spark para ver la relación entre distintos “Indicator ID” que indiquen la presencia/emisión de químicos y otros que indiquen incidencias clínicas. Haremos gráficos que reflejen la evolución de este tipo de variables en el tiempo, y compararemos las gráficas para ver relaciones entre distintos indicadores.

1.3 2. Configuración del clúster de Spark

AVISO: Debido a limitaciones en el tiempo, no he podido configurar el clúster, pero detallo toda la información que tengo sobre el proceso

Para comenzar, el stack de software que vamos a utilizar es: - Docker: Para desplegar contene-

dores que ejecuten los servicios que vamos a necesitar - Hadoop (HDFS): Para gestionar archivos distribuidos - YARN: Gestor de recursos del clúster - Spark: Para hacer operaciones en memoria sobre los datos en HDFS - Jupyter Lab: Para usar la API de PySpark para hacer el análisis y visualizar los datos

Se distribuirán los contenedores de la siguiente forma:

Ordenador principal (servicio: imagen):

- Jupyter Lab: quay.io/jupyter/base-notebook
- Name Node: bde2020/hadoop-namenode:2.0.0-hadoop2.7.4-java8
- Resource Manager: bde2020/hadoop-resourcemanager:2.0.0-hadoop2.7.4-java8
- Master Node: bde2020/spark-master:3.0.0-hadoop3.2

Ordenador secundario (servicio: imagen):

- Data Node: bde2020/hadoop-datanode:2.0.0-hadoop3.2.1-java8
- Node Manager: bde2020/hadoop-nodemanager:2.0.0-hadoop3.2.1-java8
- Worker Node: bde2020/spark-worker:3.0.0-hadoop3.2

Para permitir la comunicación entre contenedores de distintos ordenadores, se puede crear una red con Docker Swarm, o se pueden especificar las ips y puertos manualmente.

El archivo `docker-compose.yml` describe el despliegue de contenedores, y puede modificarse para añadir configuraciones a distintos servicios. En nuestro caso, al servicio de Jupyter Lab le podemos añadir un token para tener un mínimo de seguridad, y podemos especificar la ip para poder acceder desde otro ordenador si fuese necesario:

services:

jupyter-lab:

image: quay.io/jupyter/base-notebook

container_name: jupyter-lab

command: start-notebook.py --NotebookApp.token='mi-token-personalizado' --ip=mi_ip

ports:

- "8889:8888"

Dentro de este archivo se puede especificar un “volume” para especificar datos a los que tendrá acceso el contenedor. Alternativamente, se pueden enviar archivos a un contenedor usando `docker cp <ruta archivo> <id del namenode>:<ruta destino>`, y se pueden distribuir los archivos al sistema HDFS con `hdfs dfs -put <ruta local> <ruta destino hdfs>`.

Una vez el contenedor jupyter-lab esté conectado, antes de abrir un archivo, se abre una consola de jupyter lab, y se escribe `pip install pyspark findspark` y opcionalmente más módulos separados por espacios para instalar los módulos necesarios para nuestro trabajo.

Por último, se configura e inicia la `SparkSession` en una libreta, y se puede comenzar a trabajar.

```
[7]: import findspark
findspark.init()
import pyspark
from pyspark.sql import Row, SparkSession
import os
os.environ['PYARROW_IGNORE_TIMEZONE'] = '1'
```

```

os.environ['SPARK_LOCAL_IP'] = '192.168.1.19'
# Para evitar WARN. Sólo funcionará en mi ordenador en la red de mi casa.
# Comentar línea para ejecutar en otro ordenador.

spark = SparkSession.builder \
    .master('local[*]') \
    .appName('health_and_pollution') \
    .getOrCreate()

sc = spark.sparkContext

```

Setting default log level to "WARN".

To adjust logging level use `sc.setLogLevel(newLevel)`. For SparkR, use `setLogLevel(newLevel)`.

24/06/14 03:23:20 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable

1.4 3. Carga de datos y preparación de datos para el análisis

Podríamos permitir a Spark inferir las estructuras de datos, pero ya que tenemos la estructura de datos descrita en el apartado 1, haremos el schema manualmente, aunque sea un poco más laborioso, evitaremos errores.

```

[10]: from pyspark.sql.types import StructType, StructField, StringType, FloatType, \
      ↪ IntegerType

```

```

[11]: schema = StructType([
    StructField('Unique ID', StringType(), False ),
    StructField('Indicator ID', IntegerType(), False ),
    StructField('Name', StringType(), True ),
    StructField('Measure', StringType(), True ),
    StructField('Measure Info', StringType(), True ),
    StructField('Geo Type Name', StringType(), True ),
    StructField('Geo Join ID', IntegerType(), True ),
    StructField('Geo Place Name', StringType(), True ),
    StructField('Time Period', StringType(), True ),
    StructField('Start_Date', StringType(), False ),
    StructField('Data Value', FloatType(), False ),
    StructField('Message', StringType(), True )
])

# En principio 'Geo Join ID' no debería ser NULL pero hay valores NULL
# por algún motivo. Para 'Data Value' existe el DateType, pero aún
# especificando el dateFormat, me da problemas, buscaremos otra solución.

```

```

[12]: df_airquality = spark.read.csv('Air_Quality.csv', header=True, schema=schema)

```

Usando `DateType` y `dateFormat`, aunque si pude hacer select sobre los datos originales, causaba errores en acciones posteriores, por lo que he cargado la `'Start_Date'` como `StringType`. Ahora, para poder representar datos en gráficos de evolución sobre el tiempo, tendremos que convertir estos Strings en datos que pueda entender un motor de gráficos.

Ahora crearemos una columna extra con los datos convertidos a `DateType`, llamada `'Date'`. Posteriormente separaré los datos en dos tablas, químicos e incidencias, con las columnas indicadas en el apartado 1, sustituyendo `'Start_Date'`, por la nueva columna `'Date'`.

Por otra parte, para que tengan sentido nuestras gráficas, habrá que buscar el `'Time Period'` más conveniente para producir las mejores gráficas. Lo mejor, sería un `'Time Period'` lo más corto posible, pero también necesitamos datos abundantes para poder observar de manera precisa la “forma” de la evolución, y sacar las mejores conclusiones posibles. Si no hubiera suficientes datos en `'Time Period[s]'` cortos, habrá que utilizar más largos.

```
[14]: from pyspark.sql.functions import to_date, col
```

```
[15]: df_airquality = df_airquality.withColumn("Date", to_date(col("Start_Date"), "MM/
      ↪dd/yyyy")).cache()
df_airquality.select('Start_Date', 'Date').limit(5).show()
```

```
+-----+-----+
|Start_Date|      Date|
+-----+-----+
|01/01/2015|2015-01-01|
|01/01/2015|2015-01-01|
|12/01/2011|2011-12-01|
|12/01/2011|2011-12-01|
|06/01/2022|2022-06-01|
+-----+-----+
```

```
[16]: # Una vez vemos que las fechas se han convertido correctamente
      # podemos eliminar la columna original, y cachear el nuevo DataFrame
```

```
df_airquality = df_airquality.drop('Start_Date').cache()
```

```
[17]: def collect_column(source, column=None, *, sort=False, reverse=False, key=None):
      gen_error = 'An error occurred. Most likely, you didn\'t provide a valid_
      ↪source or column.' + \
          '\nValid source types are list[Row] and DataFrame.' + \
          '\nIf you provide more than one column, you have to specify the_
      ↪column parameter.' + \
          '\ncolumn must be int, str or None.'
      col_error = 'The specified column doesn\'t exist'

      try: n_col = source.columns.__len__() # N° columns
```

```

except: n_col = None
try: n_itm = source[0].__len__() # N° items
except: n_itm = None

if column == None: # If you didn't provide column, the function assumes it_
↳is the first one
    if (n_col >= 1 or n_itm >= 1): column = 0
    else: raise Exception(gen_error)

if isinstance(column, int): col_type = 'int'
elif isinstance(column, str): col_type = 'str'
else: raise Exception(gen_error) # column should be int, str or None

match (n_col, n_itm):
    case (None, _): src_type = 'list'
    case (_, None): src_type = 'df'
    case (_, _): raise Exception(gen_error)

try:
    match (col_type, src_type):
        case ( 'int', 'df' ): out = [ row[column] for row in source.
↳select(source.columns[column]).collect() ]
        case ( 'str', 'df' ): out = [ row[column] for row in source.
↳select(column).collect() ]
        case ( _ , 'list' ): out = [ row[column] for row in source ]
    except: raise Exception(col_error)

if sort: out.sort(reverse=reverse, key=key)
return out

```

```

[18]: periods_df = df_airquality.groupBy('Time Period').count()
periods = collect_column(periods_df)
periods.sort()
for item in periods: print(item)

```

2-Year Summer Average 2009-2010

2005

2005-2007

2009-2011

2010

2011

2012-2014

2013

2014

2015

2015-2017

2017-2019

2019
Annual Average 2009
Annual Average 2010
Annual Average 2011
Annual Average 2012
Annual Average 2013
Annual Average 2014
Annual Average 2015
Annual Average 2016
Annual Average 2017
Annual Average 2018
Annual Average 2019
Annual Average 2020
Annual Average 2021
Annual Average 2022
Summer 2009
Summer 2010
Summer 2011
Summer 2012
Summer 2013
Summer 2014
Summer 2015
Summer 2016
Summer 2017
Summer 2018
Summer 2019
Summer 2020
Summer 2021
Summer 2022
Winter 2008-09
Winter 2009-10
Winter 2010-11
Winter 2011-12
Winter 2012-13
Winter 2013-14
Winter 2014-15
Winter 2015-16
Winter 2016-17
Winter 2017-18
Winter 2018-19
Winter 2019-20
Winter 2020-21
Winter 2021-22

Podemos ver que hay 4 tipos de periodos: - Semestres: Summer|Winter yyyy[-yy] - Año: Annual Average yyyy | yyyy - Múltiples años: yyyy-yyyy - Otro: 2-Year Summer Average 2009-2010 (único caso)

Lo ideal sería hacer gráficas a partir de los datos de los semestres y/o años, sin embargo, al

desconocer la disponibilidad de estos datos en estos periodos, deberíamos como mínimo, extraer información de los “Múltiples años”, que sea más general y fácilmente manejable por el motor de Spark.

```
[20]: import re
from pyspark.sql.functions import udf, when, row_number
from pyspark.sql.window import Window
```

```
[21]: def period_type(x):
    if re.fullmatch(r'[swSW].{5} [\d]{4}.*', x ): return 'semester'
    elif re.fullmatch(r'([aA]nnual [aA]verage )0,1[\d]{4}', x ): return 'year'
    elif re.fullmatch(r'[\d]{4}-[\d]{4}', x ): return 'years'
    else: return 'other'

ptype = udf( period_type, StringType() )
```

```
[22]: period_types = periods_df.withColumn('Period Type', ptype('Time Period'))
period_types.show()
period_years = period_types.where(`Period Type` LIKE "years")
period_years.show()
```

24/06/14 03:23:37 WARN GarbageCollectionMetrics: To enable non-built-in garbage collector(s) List(G1 Concurrent GC), users should configure it(them) to spark.eventLog.gcMetrics.youngGenerationGarbageCollectors or spark.eventLog.gcMetrics.oldGenerationGarbageCollectors

| Time Period | count | Period Type |
|---------------------|-------|-------------|
| Winter 2021-22 | 282 | semester |
| 2009-2011 | 480 | years |
| Winter 2018-19 | 282 | semester |
| 2017-2019 | 489 | years |
| 2019 | 321 | year |
| 2014 | 96 | year |
| Winter 2015-16 | 282 | semester |
| 2013 | 144 | year |
| 2005 | 417 | year |
| Summer 2009 | 423 | semester |
| Annual Average 2013 | 282 | year |
| Annual Average 2022 | 282 | year |
| Annual Average 2021 | 282 | year |
| Winter 2008-09 | 282 | semester |
| Winter 2016-17 | 282 | semester |
| Summer 2022 | 423 | semester |
| Summer 2014 | 423 | semester |
| 2005-2007 | 480 | years |
| Summer 2010 | 423 | semester |


```
|      Summer 2018|  423|   semester|
+-----+-----+-----+
only showing top 20 rows
```

[Stage 9:>

(0 + 1) / 1]

```
+-----+-----+-----+
|Time Period|count|Period Type|
+-----+-----+-----+
|  2009-2011|  480|      years|
|  2017-2019|  489|      years|
|  2005-2007|  480|      years|
|  2012-2014|  480|      years|
|  2015-2017|  480|      years|
+-----+-----+-----+
```

```
[23]: def subtract_years(x):
      try:
          inicio, final = map( lambda x:int(x) , x.split('-') )
          return final - inicio
      except: return None

      years = udf( subtract_years , IntegerType() )
```

```
[24]: period_years.withColumn('Years elapsed', years('Time Period')).show()
```

[Stage 12:>

(0 + 1) / 1]

```
+-----+-----+-----+-----+
|Time Period|count|Period Type|Years elapsed|
+-----+-----+-----+-----+
|  2009-2011|  480|      years|          2|
|  2017-2019|  489|      years|          2|
|  2005-2007|  480|      years|          2|
|  2012-2014|  480|      years|          2|
|  2015-2017|  480|      years|          2|
+-----+-----+-----+-----+
```

Podemos ver que los intervalos de varios años, todos corresponden a periodos de dos años. Por lo que podemos fácilmente filtrar los datos por tipo de periodo (semester, year y years), y producir gráficas con datos de una frecuencia continua.

```
[26]: measures = df_airquality.groupBy('Indicator ID', 'Name', 'Measure', 'Measure_
↳Info')\
        .count().orderBy('Indicator ID', ascending = True).
        ↳cache()
measures.pandas_api()
```

```
[26]:
```

| | Indicator ID | | Name |
|--------------------------------------|--------------|----------------------|--|
| Measure | | Measure Info | count |
| 0 | 365 | | Fine particles (PM 2.5) |
| Mean | | mcg/m3 | 5922 |
| 1 | 375 | | Nitrogen dioxide (NO2) |
| Mean | | ppb | 5922 |
| 2 | 386 | | Ozone (O3) |
| Mean | | ppb | 2115 |
| 3 | 639 | | Deaths due to PM2.5 |
| Estimated annual rate (age 30+) | | per 100,000 adults | 240 |
| 4 | 640 | | Boiler Emissions- Total SO2 Emissions |
| Number per km2 | | number | 96 |
| 5 | 641 | | Boiler Emissions- Total PM2.5 Emissions |
| Number per km2 | | number | 96 |
| 6 | 642 | | Boiler Emissions- Total NOx Emissions |
| Number per km2 | | number | 96 |
| 7 | 643 | | Annual vehicle miles traveled |
| Million miles | | per square mile | 321 |
| 8 | 644 | | Annual vehicle miles traveled (cars) |
| Million miles | | per square mile | 321 |
| 9 | 645 | | Annual vehicle miles traveled (trucks) |
| Million miles | | per square mile | 321 |
| 10 | 646 | | Outdoor Air Toxics - Benzene |
| Annual average concentration | | Âµg/m3 | 203 |
| 11 | 647 | | Outdoor Air Toxics - Formaldehyde |
| Annual average concentration | | Âµg/m3 | 203 |
| 12 | 648 | | Asthma emergency department visits due to PM2.5 |
| Estimated annual rate (under age 18) | | per 100,000 children | 240 |
| 13 | 650 | | Respiratory hospitalizations due to PM2.5 (age 20+) |
| Estimated annual rate | | per 100,000 adults | 240 |
| 14 | 651 | | Cardiovascular hospitalizations due to PM2.5 (age 40+) |
| Estimated annual rate | | per 100,000 adults | 240 |
| 15 | 652 | | Cardiac and respiratory deaths due to Ozone |
| Estimated annual rate | | per 100,000 | 240 |
| 16 | 653 | | Asthma emergency departments visits due to Ozone |
| Estimated annual rate (under age 18) | | per 100,000 children | 245 |
| 17 | 655 | | Asthma hospitalizations due to Ozone |
| Estimated annual rate (under age 18) | | per 100,000 children | 244 |
| 18 | 657 | | Asthma emergency department visits due to PM2.5 |
| Estimated annual rate (age 18+) | | per 100,000 adults | 240 |
| 19 | 659 | | Asthma emergency departments visits due to Ozone |

| | | |
|---------------------------------|--------------------|--------------------------------------|
| Estimated annual rate (age 18+) | per 100,000 adults | 240 |
| 20 | 661 | Asthma hospitalizations due to Ozone |
| Estimated annual rate (age 18+) | per 100,000 adults | 240 |

En esta tabla podemos darnos cuentas de tres cosas: - Hay medidas (mayormente datos clínicos) que sólo están disponibles como “annual rate” (no todos los datos parecen tener la misma disponibilidad). - Los ‘Indicator ID’ están bastante ordenados. Exeptuando 639 (Deaths due to PM2.5), todos los datos sobre contaminación están entre 365 y 647, mientras que los datos clínicos están entre 648 y 661. - Hay muchas medidas que comparten ‘count’ similares, ésto junto a que se tratan de datos de categorías similares, podrían indicar que vienen de fuentes similares, con datos y metodologías similares.

Para simplificar la tarea de elegir qué registros ignorar para realizar los gráficos, identificaré los distintos tipos de períodos y separaré los datos en contaminación y datos clínicos.

```
[28]: # Listas de ID de datos clínicos, contaminación y ubicaciones

# Pollution ID List
poll_ID = collect_column( measures.select('Indicator ID').where('`Indicator_ID`<=647 AND `Indicator ID`!=639') \
                           .orderBy('Indicator ID') )

# Clinical Data ID List
clin_ID = collect_column( measures.select('Indicator ID').where('`Indicator_ID`>=648 OR `Indicator ID`=639') \
                           .orderBy('Indicator ID') )

# Locations ID list
locations = collect_column( df_airquality.groupBy('Geo Join ID').count().
                             collect(), 'Geo Join ID' )
```

```
[29]: df_airquality = df_airquality.withColumn('Period Type', ptype('Time Period'))
```

```
[30]: df_poll_complete = df_airquality.where( col('Indicator ID').isin(poll_ID) )
df_clin_complete = df_airquality.where( col('Indicator ID').isin(clin_ID) )
```

```
[31]: def most_frequent(df, id_col, to_count, show=False):
    df_temp = df.groupBy(id_col, to_count).count()
    window = Window.partitionBy(id_col).orderBy(col('count').desc())
    df_temp = df_temp.withColumn('row_number', row_number().over(window))
    df_temp = df_temp.filter(col('row_number') == 1).select(id_col, to_count)
    if show: df_temp.show()
    return df_temp
```

```
[32]: print('Pollution Data: most frequent Period Types by ID')
mf_poll = most_frequent(df_poll_complete, 'Indicator ID', 'Period Type', True)
print('Clinic Data: most frequent Period Types by ID')
mf_clin = most_frequent(df_clin_complete, 'Indicator ID', 'Period Type', True)
```

Pollution Data: most frequent Period Types by ID

| Indicator ID | Period Type |
|--------------|-------------|
| 365 | semester |
| 375 | semester |
| 386 | semester |
| 640 | year |
| 641 | year |
| 642 | year |
| 643 | year |
| 644 | year |
| 645 | year |
| 646 | year |
| 647 | year |

Clinic Data: most frequent Period Types by ID

| Indicator ID | Period Type |
|--------------|-------------|
| 639 | years |
| 648 | years |
| 650 | years |
| 651 | years |
| 652 | years |
| 653 | years |
| 655 | years |
| 657 | years |
| 659 | years |
| 661 | years |

Las listas “poll_ID” y “clin_ID” son bastante cortas, por lo que podemos ver de un vistazo todos los datos de “mf_poll” y “mf_clin”. Se puede observar que los periodos de tiempo de la tabla de contaminantes son bastante fáciles de manejar; semestres y años. Los datos clínicos parecen más complicados porque son todos de tipo “years”, pero como ya hemos visto, todos los datos en esta categoría se tratan de periodos de 2 años.

```
[34]: for i, item in enumerate(df_airquality.columns):  
      i_str = str(i)  
      if len(i_str) == 1 : i_str = ' '+i_str  
      print(i_str, ' ', item)
```

0 Unique ID

```

1  Indicator ID
2  Name
3  Measure
4  Measure Info
5  Geo Type Name
6  Geo Join ID
7  Geo Place Name
8  Time Period
9  Data Value
10 Message
11 Date
12 Period Type

```

```

[35]: # Indicator ID, Geo Join ID, Data Value, Date, Period Type
clean_select = list( df_airquality.columns[i] for i in (1, 6, 9, 11, 12) )
clean_where = ' '.join( [ *('AND `'+item+'` IS NOT NULL' for item in_
↪clean_select) ] ) [4:]

# tablaReducida = tabla.select('co','lum','nas').orderBy('ID').where('No haya_
↪valores NULL')

df_poll = df_poll_complete.select(clean_select).orderBy('Indicator ID').
↪where(clean_where) \
        .cache()
df_clin = df_clin_complete.select(clean_select).orderBy('Indicator ID').
↪where(clean_where) \
        .cache()

```

1.5 4. Análisis de los datos

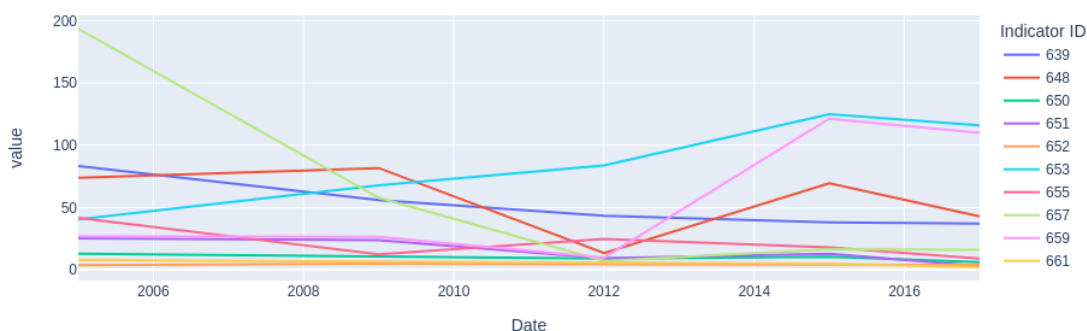
La forma más sencilla que veo de relacionar estos datos, se trata de representarlos en gráficos. Primero podemos obtener una visión general mostrando una de nuestras tablas reducidas en un gráfico. Como todos los datos clínicos son bianuales, la tabla de datos clínicos no nos dará problemas para representarla.

1.5.1 Datos clínicos

```

[37]: df_clin.orderBy('Date').pandas_api()\
        .pivot(index='Date', values='Data Value', columns='Indicator ID')\
        .plot(kind='line')

```



```
[38]: measures.where(`Indicator ID` LIKE 653 OR `Indicator ID` LIKE 659 OR
↪ `Indicator ID` LIKE 648`) \
      .select('Name', 'Measure').collect()
```

```
[38]: [Row(Name='Asthma emergency department visits due to PM2.5', Measure='Estimated
annual rate (under age 18)'),
      Row(Name='Asthma emergency departments visits due to Ozone', Measure='Estimated
annual rate (under age 18)'),
      Row(Name='Asthma emergency departments visits due to Ozone', Measure='Estimated
annual rate (age 18+)')]
```

Podemos observar en 2015 un pico en los indicadores 653, 659 (Visitas a emergencias por asma a causa de Ozono, de menores y mayores de edad respectivamente) y 648 (Visitas a emergencias por asma a causa de PM2.5 de menores de edad).

Los nombres de los indicadores ya nos hacen *spoiler* de cuál va a ser la causa de éstos problemas, así que lo más probable es que haya un pico en la producción/detección de ozono y PM2.5 (partículas de menos de 2,5 micras).

Las medidas cubren el periodo *a partir* de la fecha ('Date' en nuestra tabla, 'Start_Date' en la tabla original), por lo que las medidas representan más el año siguiente que el que se muestra en la tabla original. Con esta información, podemos entender el pico de casos de 2015, ocurrió en realidad en 2016.

```
[40]: df_airquality.where( col('Indicator ID').isin(cclin_ID) ).groupBy('Time Period',
↪ 'Date').count().show()
```

```
+-----+-----+-----+
|Time Period|      Date|count|
+-----+-----+-----+
| 2005-2007|2005-01-01|  480|
| 2009-2011|2009-01-01|  480|
| 2017-2019|2017-01-01|  489|
| 2015-2017|2015-01-01|  480|
```

```
| 2012-2014|2012-01-02| 480|
+-----+-----+-----+
```

```
[41]: # 365 pm25, 386 ozono, 641 pm25
print('365 pm25 presente')
df_poll.where("`Indicator ID` LIKE 365").groupBy('Period Type').count().show()
print('386 ozono presente')
df_poll.where("`Indicator ID` LIKE 386").groupBy('Period Type').count().show()
print('641 pm25 emitido')
df_poll.where("`Indicator ID` LIKE 641").groupBy('Period Type').count().show()
```

365 pm25 presente

```
+-----+-----+
|Period Type|count|
+-----+-----+
| semester| 3948|
| year| 1974|
+-----+-----+
```

386 ozono presente

```
+-----+-----+
|Period Type|count|
+-----+-----+
| semester| 1974|
| other| 141|
+-----+-----+
```

641 pm25 emitido

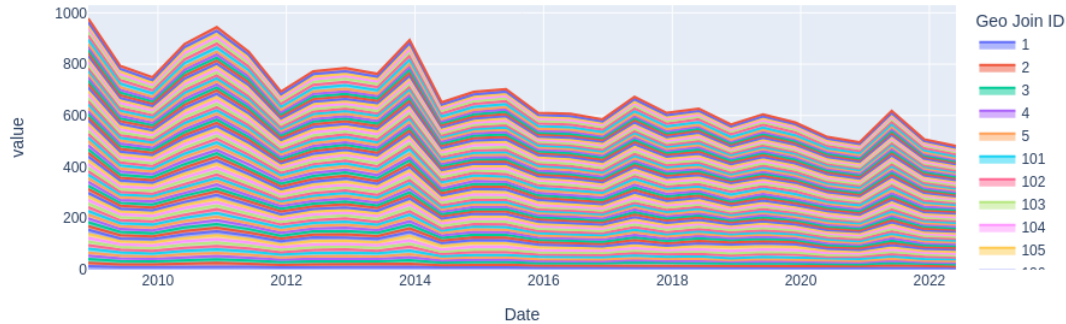
```
+-----+-----+
|Period Type|count|
+-----+-----+
| year| 96|
+-----+-----+
```

Al tener diferentes ‘Period Type’, parece que vamos a necesitar una tabla por cada químico. Usaremos gráficas del tipo área porque los datos son de tipo densidad, por lo que no se verán **tan** influidas por el área o los habitantes de la ubicación y serán fácilmente acumulables.

1.5.2 Presencia de los PM2.5

```
[43]: df_poll.where("`Indicator ID` LIKE 365 AND `Period Type` LIKE 'semester'").
      ↪orderBy('Date').pandas_api()\
      .pivot(index='Date', values='Data Value', columns='Geo Join ID').
      ↪plot(kind='area')
```

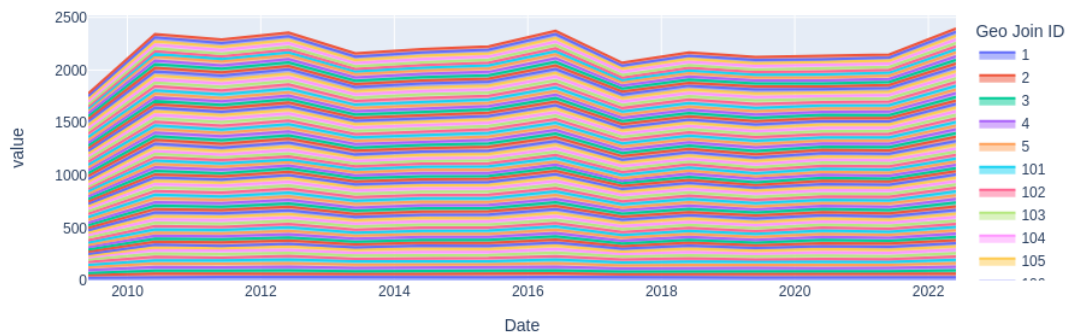
24/06/14 03:24:12 WARN SparkStringUtils: Truncated the string representation of a plan since it was too large. This behavior can be adjusted by setting 'spark.sql.debug.maxToStringFields'.



Podemos observar un pico muy importante en 2014 y una gran bajada en 2016, por lo que a priori, no parece estar estrechamente relacionado con los casos de asma ni PM2.5. Sin embargo (usando gráficas de líneas), se pueden observar algunas ubicaciones con picos en ese año, que podrían estudiarse más adelante, como las ubicaciones 307 y 101.

1.5.3 Presencia de Ozono

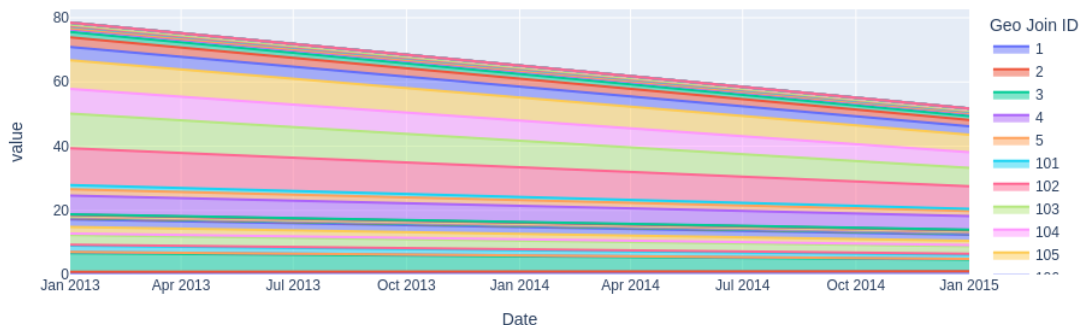
```
[45]: df_poll.where("`Indicator ID` LIKE 386 AND `Period Type` LIKE 'semester'").
      >orderBy('Date').pandas_api()\
      .pivot(index='Date', values='Data Value', columns='Geo Join ID').
      >plot(kind='area')
```



Podemos observar en 2016 hay un pico significativo en la presencia de ozono en la mayoría de ubicaciones, lo que se convierte en un fuerte indicio de la relación entre el ozono y el asma.

1.5.4 Emisiones de PM2.5

```
[47]: # PM25 emitted
df_poll.where("`Indicator ID` LIKE 641 AND `Period Type` LIKE 'year'").
    ↳orderBy('Date').pandas_api()\
        .pivot(index='Date', values='Data Value', columns='Geo Join ID').
    ↳plot(kind='area')
```



Esta última gráfica parece tener muy poca información...

```
[49]: df_poll.where("`Indicator ID` LIKE 641 AND `Period Type` LIKE 'year'").count()
```

```
[49]: 96
```

```
[50]: df_poll.where("`Indicator ID` LIKE 641 AND `Period Type` LIKE 'year'").
    ↳groupBy('Geo Join ID').count().count()
```

```
[50]: 47
```

96 / 47 2

En esta última gráfica, hay 2 puntos por cada valor de 'Geo Join ID'.

1.6 5. Análisis de la información extraída.

A través de los resultados obtenidos, se puede comprobar que hay cierta relación entre los niveles de ozono y las visitas a emergencias debido al asma, sin embargo, los datos clínicos son escasos y las gráficas y conclusiones resultantes pueden no representar la realidad con precisión.

El pico de casos de 2016 es una tendencia lo bastante pronunciada para sacar conclusiones al respecto, pero la falta de continuidad de los datos es un problema para realizar cualquier análisis o comprobar cualquier extrapolación.

1.7 6. Conclusiones.

Con este tipo de datos siempre se puede sacar más y más información si hacemos más y más preguntas, tanto preguntas prudentes (¿Podemos estar seguros de esto? ¿No habrá otras variables?) como inquisitivas (¿Estarán estas dos cosas relacionadas? ¿En otras circunstancias ocurre lo mismo?). El dataset es relativamente pequeño y hay preguntas y visualizaciones que realmente no se podrán explorar. Sin embargo, aunque se podría extraer más información del dataset, los objetivos de este estudio es no tanto el entendimiento de estos fenómenos sino el uso de herramientas de Big Data para el análisis. Para lo cual ha sido suficiente.

En resumen, la interfaz de Spark es una herramienta muy útil para depurar, organizar y representar datos para su análisis, sin embargo podría hacer falta más tiempo y/o datos para sacar conclusiones claras sobre la relación entre contaminantes y su impacto en la salud.

2 Anexo: Buscando métodos más eficientes

```
[55]: import timeit
```

```
[56]: def map_unpack(df, column):  
    return list( map( lambda x:x[0], df.select(column).collect() ))
```

```
[57]: def row_unpack(df, column):  
    return [ row[column] for row in df.select(column).collect() ]
```

```
[58]: def rdd_flatmap(df, column):  
    return df.select(column).rdd.flatMap(lambda x:x).collect()
```

```
[59]: def test_speed(*functions, df=None, column=None, number=10, silent=False):  
    results = []  
  
    for func in functions:  
        time = timeit.timeit( lambda: func(df, column), number=number )  
        name = func.__name__  
        results.append( (time, name) )  
  
    results.sort(key=lambda x:x[0])  
    results = list( map( lambda x: ( x[0] / number, x[1] ) , results))  
  
    if not silent:  
        max_l = max( map(lambda x:len(x[1]), results))  
        print(f'Speed Test Results (average execution time)\n')  
        for i, item in enumerate(results):  
            print(f'{(max_l-len(item[1]))*" "}{item[1]}: time(seconds) =  
↳{item[0]}')  
    return results
```

```
[120]: test_results = test_speed(rdd_flatmap, row_unpack, collect_column, map_unpack, \
                                df=df_airquality, column='Date', number=10)
```

Speed Test Results (average execution time)

```
map_unpack: time(seconds) = 0.17414456219994462
row_unpack: time(seconds) = 0.19861308849976922
collect_column: time(seconds) = 0.20426404130012088
rdd_flatmap: time(seconds) = 0.42369224059984845
```

Queda demostrado que la manipulación de DataFrames es bastante más eficiente que la de RDDs. Incluso con flujos lógicos y manejo de excepciones de por medio, hay una gran diferencia entre los métodos basados en DataFrames y los que están basados en RDDs.

Nota 1: Me sorprende la poca diferencia entre `row_unpack` y `collect_column`, y más me sorprende, que la primera vez que ejecutas el test, `collect_column` es más rápido que `row_unpack`, cuando básicamente es `row_unpack` con pasos extra. Probablemente se deba a que la función `collect_column` ha sido usada antes en el documento y está en cierta forma “cacheada”, pero si ese es el caso, debería normalizarse con un número de ejecuciones de `timeit` grande, y no es el caso, probablemente `timeit`, una vez ejecutado, accede a la función de una forma diferente a como lo harían varias líneas de código llamando varias veces la función, o quizá sea la propia ejecución en celdas la culpable. Tienes que ejecutar varias veces la celda del test para conseguir resultados consistentes. Los datos visibles son después de ejecutar la celda cuatro veces.

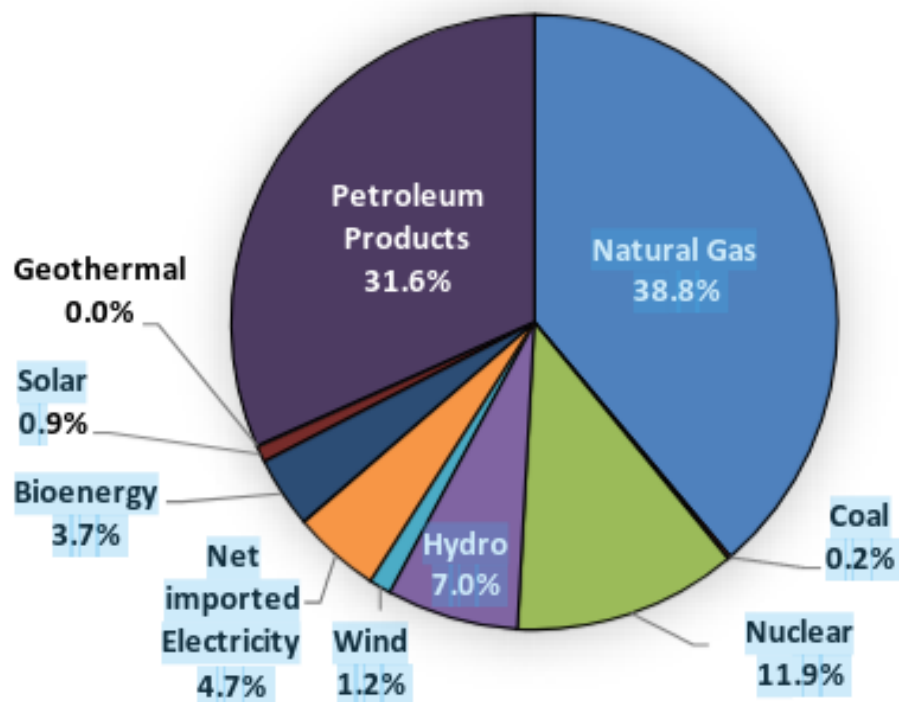
Nota 2: `map_unpack` es una adición posterior, por lo que no la he usado en el análisis, sin embargo parece ser la opción más eficiente.

3 Anexo: NO2 invierno

Aquí se puede observar algo curioso. El indicador 375 corresponde a la presencia de dióxido de nitrógeno; químico resultante de las [combustiones a altas temperaturas](#)⁴. Se pueden observar picos muy pronunciados en invierno. Mi hipótesis es que la presencia de este químico está estrechamente relacionada con la calefacción que se usa en invierno, ya sea por combustión para obtener calor o, más probable, la generación de energía necesaria para fines similares a través de combustibles fósiles ⁷.

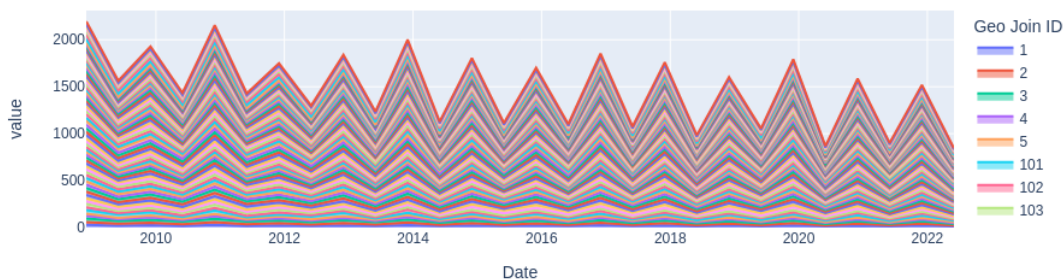
Fuentes de energía del estado de Nueva York:

2020 NYS Fuel Sources



```
[64]: df_poll.where(`Indicator ID` LIKE 375 AND `Period Type` LIKE "semester").
      ↳orderBy('Date').pandas_api()\
      .pivot(index='Date', columns='Geo Join ID', values='Data Value') \
      .plot(kind='area', title='NO2 over Time: Semesters. Everywhere.')
```

NO2 over Time: Semesters. Everywhere.



¿Qué significan estos datos?

Los datos de concentración de NO2 vienen en ppb (partes por mil millones). Según esta [tabla de conversión](#)⁵, 1 ppb es equivalente a 1.88 g/m³.

El límite anual fijado por [directiva](#)⁶ europea es 40 microgramos/m³, mientras que la directiva americana es de 100 microgramos/m³. Con unos cálculos sencillos, podemos ver a qué equivale en ppb y que tanto se cumplen estos límites en nuestros datos.

```
[66]: # 1 ppb = 1.88 g/m3 --> 1 g/m3 = 1 / 1.88 ppb
eur = 40 * ( 1 / 1.88 )
print('Límite europeo en ppb:', eur)
usa = 100 * ( 1 / 1.88 )
print('Límite estadounidense ppb:', usa)
```

Límite europeo en ppb: 21.27659574468085

Límite estadounidense ppb: 53.191489361702125

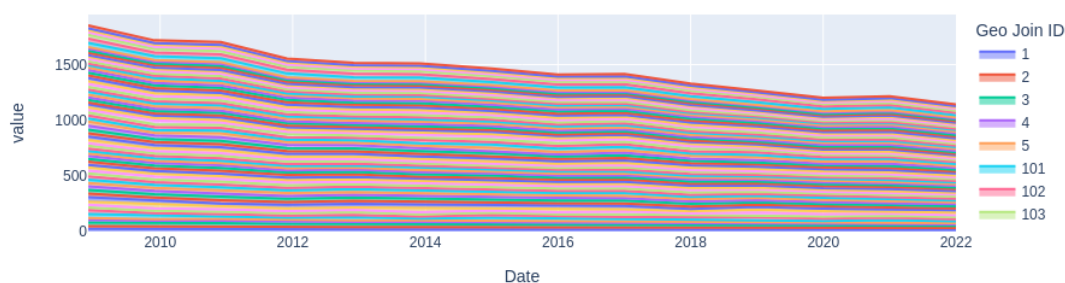
```
[67]: print('La ubicación con menos datos para realizar una tabla es:')
df_poll.where(`Indicator ID` LIKE 375 AND `Period Type` LIKE "year").
    ↳groupby('Geo Join ID') \
        .count().orderBy('count').limit(1).show()
```

La ubicación con menos datos para realizar una tabla es:

```
+-----+-----+
|Geo Join ID|count|
+-----+-----+
|      305307|   14|
+-----+-----+
```

```
[68]: # Al tener suficientes datos disponibles de tipo "year", no necesito operar.
# Con un nuevo select ya puedo hacer el gráfico.
df_poll.where(`Indicator ID` LIKE 375 AND `Period Type` LIKE "year").
    ↳orderBy('Date').pandas_api() \
        .pivot(index='Date', columns='Geo Join ID', values='Data Value') \
        .plot(kind='area', title='NO2 over Time: Years. Everywhere.')
```

NO2 over Time: Years. Everywhere.



```
[69]: df_locations = df_airquality.groupBy('Geo Join ID', 'Geo Place Name', 'Geo Type_  
      ↪Name').count()  
df_locations.where('`Geo Join ID` LIKE 105').collect()
```

```
[69]: [Row(Geo Join ID=105, Geo Place Name='Crotona -Tremont', Geo Type Name='UHF42',  
count=170),  
      Row(Geo Join ID=105, Geo Place Name='Midtown (CD5)', Geo Type Name='CD',  
count=110)]
```

A excepción del Geo Join ID = 105 (Crotona, West Bronx) en 2009, con una concentración de 42.1 ppb, todas las ubicaciones medidas pasan no sólo los estándares estadounidense, sino también los estándares europeos.

Con esto nos podríamos ir a casa con la satisfacción de que todo está bien y nuestro trabajo está hecho, pero la legalidad no lo es todo. Según [Wikipedia5](#), el dióxido de nitrógeno “Es un gas tóxico, irritante [...] [y] afecta principalmente al sistema respiratorio”.

Con unas tendencias tan claras de mayor concentración de este gas en invierno, por muy legal que sea ¿habrá más problemas de salud relacionados con el NO2 en invierno?