uc3m | Universidad Carlos III de Madrid

Grado en Ingeniería Informática
2024-2025

Arquitectura de Datos

"Practica 1.2 (ETL) "

Adrian Cortázar Leiva

Jaime Vaquero Rabahieh

Tomás Mendizábal

Índice

1. Diseño y Construcción de Pipelines

1.1. Descripción del proceso ETL

El proceso de ETL o de Extracción, Transformación y Carga se define como una serie de pasos a realizar con la finalidad de adaptación de la información a nuevos entornos de almacenamiento (ej. Bases de datos, Data Warehouses).

En esta práctica, el objetivo es aplicar un proceso ETL o pipeline para migrar los datos proporcionados por el enunciado a una estructura de base de datos no SQL como es MongoDB.

Nuestro proceso consiste de los siguientes pasos:

- Carga de los datos anteriores con formatos incompatibles y algunas impurezas.
- Transformación de dichos datos (eliminación de valores nulos, imputación de atributos, limpieza de formatos, estandarización de palabras, etc.).
- Creación de las colecciones y esquemas pertinentes de validación asociados a la óptima representación de la información en la base de datos final.
- Inserción de los datos transformados en la base de datos final.

1.2. Proceso de extracción

En el archivo "load.py" extraemos todos los csv. Esto lo hacemos mediante un diccionario de DataFrames de panda, siendo cada clave el nombre de la tabla y el valor de la key la propia tabla/DataFrame. El código usado es el siguiente:

```
import pandas as pd
import os
import re

def load_db():
    # paths usados
    pwd = os.getcwd()
    dataset_dir = f"{pwd}/DatasetsArquiViejos"

    # leer datasets y guardarlos en un diccionario
    df = {}
    for item in os.listdir(dataset_dir):
        try:
            file_name = re.findall("[A-Z][^A-Z]*", item)[0]
            temp_db = pd.read_csv(f"{dataset_dir}/{item}")
            df[file_name] = temp_db
        except IndexError:
            file_name = item.split('.')[0]
            temp_db = pd.read_csv(f"{dataset_dir}/{item}")
            df[file_name] = temp_db
    return df

```

1.3. Proceso de limpieza

En el archivo "change.py" se realizó la limpieza de datos. Para ejecutar este archivo, es necesario importar estas librerías de esta manera:

```
import pandas as pd
import datetime as dt
from dateutil import parser

```

Este archivo está formado por varias funciones cada una con un labor en particular:

1. **capitalize column**: Capitaliza todos los valores de una columna. Se le pasa como argumento el diccionario completo y los nombres de la columna a modificar y la tabla a la que pertenece. Código:

```
# capitalizar una columna entera
def capitalize_column(df: dict, table_name: str, colname: str) -> None:
    df[table_name][colname] = df[table_name][colname].apply(lambda x: x.upper())

```

2. **take atribute**: Busca un atributo en una tabla que tenga un id igual al pasado por argumento. Se hace para tablas en las que existe una columna id. Código:

```
# Encontrar un dato en una tabla si su id es igual al pasado por argumento
def take_atribute(df: dict, tabla: str, columna: str, id: int) -> str | None:
    archivo = df[tabla]
    fila = None
    for i in range(len(archivo["ID"])):
        valor = archivo["ID"][i]
        if id == valor:
            fila = i
            return archivo[columna][fila]
    return None

```

3. **find table by column**: Busca una tabla que contenga la columna pedida. Como extra, dicha tabla no debe estar en la lista "tablas tachadas". Esto se hace para que la funcion sepa que tablas se excluyen de la búsqueda. Código:

```
# Encuentra una tabla sin contar las tachadas que contenga una columna con el mismo nombre que la pasada por argumento
def find_table_by_column(df: dict, tablas_tachadas: list, column: str) -> str | None:
    for tabla in df:
        if tabla not in tablas_tachadas:
            t_analizar = df[tabla]
            if column in t_analizar:
                return tabla
    return None

```

4. **find id**: Busca un id en un tabla (se pasa su nombre como argumento) que se encuentre en la posición también pasada como argumento. Como no todas las columnas deidentificación se llaman ID, se resliza un if-else para primero identificar la tabla y luego devolver el id. Código:

```
# Encuentra un id según que tabla (hay columnas ID que no se llaman ID) y que fila
def find_id(df: dict, tabla: str, fila: int):
    archivo = df[tabla]
    if tabla == "Usuarios":
        return archivo["NIF"][fila]
    elif tabla == "meteo24":
        return archivo["PROVINCIA"][fila]
```

```
    else:
        return archivo["ID"][fila]


```

5. **empty data**: Modifica los NaN encontrados. Coge las casillas de las tablas con NaN y las modifica según estos criterios:

   a) Si es una fila con valor de DIRECCION AUX no nulo y el valor nulo es de nombre de vía, tipo de vía o número de vivienda; usa la función *aux dir* para eliminar el NaN (se explica después.

   b) En cualquier otro caso intenta buscar con la funcion *find table by column* un valor de dicha columna que tenga un id igual al suyo. Si no lo encuentra, se cambia el NaN por uno de estos valores predeterminados:

      1) FECHA: 31 de diciembre de 2018
      2) OTRO CASO: la estructura **id-nombre de columna-ausente**

Cabe aclarar que este función se dieño sin ChatGPT, pero fue corregida por esta IA. Código:

```
# Para rellenar filas vacias
def empty_data(df_dict):
    for tabla_name, tabla in df_dict.items():
        if tabla_name != "Codigo":
            # Convertir a tipo 'object' para poder indicar los casos nulos que no se pueden modificar
            tabla = tabla.astype(object)
            nan_positions = tabla.isna()  # Identificar posiciones NaN
            assignment_dict = {}  # Diccionario para almacenar asignaciones
            for columna in nan_positions.columns:
                nan_indices = nan_positions.index[
                    nan_positions[columna]
                ]  # Se cogen los indices con NaN de dicha columna
                for i in nan_indices:
                    valor = aux_dir(
                        tabla_name, tabla, columna, i
                    )  # Caso de direccion auxiliar. Si funciona, se salta el resto
                    if valor is None:
                        tachadas = [tabla_name]
                        id_val = find_id(
                            df_dict, tabla_name, i
                        )  # Obtener una vez el id
                        while True:
                            new_tab = find_table_by_column(
                                df_dict, tachadas, columna
                            )  # Buscar una tabla con un valor no NaN en la posicion con el mismo
                            # id que el del elemento actual
                            if (
                                new_tab is None
                            ):  # Caso fallido -> cambiar NaN por valor predeterminado
                                if columna in [
                                    "FECHA_INSTALACION",
                                    "FECHA",
                                    "FECHA_REPORTE",
                                    "FECHA_INTERVENCION",
                                ]:  # Caso fecha
                                    valor = dt.datetime.strptime(
                                        "2018-12-31 00:00:00", "%Y-%m-%d %H:%M:%S"
                                    )
```

```
                                else:  # Otro caso
                                    valor = f"{id_val}-{columna}-ausente"
                                break
                            valor = take_atribute(
                                df_dict, new_tab, columna, id_val
                            )  # Se ecnuebtra la tabla y se checkea el valor
                            if valor is not None:
                                break  # Caso exitoso
                            tachadas.append(
                                new_tab
                            )  # Valor NaN, asi que se repite el proceso sin contar con al tabla encontrada

                    # Añadir al diccionario de asignación
                    if columna not in assignment_dict:
                        assignment_dict[columna] = {}
                    assignment_dict[columna][i] = valor

            # Realizar las asignaciones en bloque usando el diccionario de asignación
            for columna, valores in assignment_dict.items():
                tabla.loc[valores.keys(), columna] = list(valores.values())

            # Actualizar la tabla en el diccionario original (si no es una copia)
            df_dict[tabla_name] = tabla

```

6. **aux dir**: Es la función sacada de *empty data* que ayuda a eliminar los NaN de las columnas TIPO VIA, NOM VIA y NUM VIA en caso de existir DIRECCION AUX. Se basa en descomponer DIRECCION AUX en TIPO VIA, NOM VIA y NUM VIA y coger el valor que se pide (se elige según qué nombre de columna de los tres se pase por argumento). Esta función se creo con la ayuda de ChatGPT. Código:

```
# Para el caso de direccion auxiliar, descomponerla en TIPO_VIA, NOM_VIA y NUM_VIA e insertar la que se
# nombra como argumento
def aux_dir(tabla_name, tabla, columna, i):
    if tabla_name in ["Areas", "Juegos"] and columna in [
        "TIPO_VIA",
        "NOM_VIA",
        "NUM_VIA",
    ]:
        if not pd.isna(tabla["DIRECCION_AUX"][i]):
            lista = tabla["DIRECCION_AUX"][i].split(" ")
            via = None
            n = 0
            if "." in lista:
                via = lista[2]
                n = 3
            else:
                via = lista[0]
            nombre, numero = "", ""
            while n < len(lista):
                if lista[n].isdigit():
                    numero = lista[n]
                    break
                if lista[n] == ",":
                    break
                nombre += lista[n]
                n += 1
            if columna == "TIPO_VIA":
```

```
                return via
            elif columna == "NOM_VIA":
                return nombre
            else:
                return numero
    return None

```

7. **reformatear fecha**: Coge cualquier fecha y la pasa a formato mongo DB (YYYY-MM-DD HH:mm:ss). En caso de NaN o error pone la predeterminada (2018-12-31 00:00:00). Esta función se corrigió con ChatGPT. Código:

```
# Coge cualquier fecha y la pasa a formato %Y-%m-%d %H:%M:%S.
# En caso de NaN se coloca la predeterminada (2018-12-31)
def reformatear_fecha(df: dict, table_name: str, column_name: str):
    tablas = [
        "Areas",
        "Encuestas",
        "Incidencias",
        "Incidentes",
        "Juegos",
        "Mantenimiento",
        "meteo24",
    ]
    columnas = ["FECHA_INSTALACION", "FECHA", "FECHA_REPORTE", "FECHA_INTERVENCION"]
    if table_name in tablas and column_name in columnas:
        archivo = df[table_name]
        fechas_nuevas = []
        for fecha in archivo[column_name]:
            try:
                f_formt = parser.parse(fecha)
                f_formt.strftime("%Y-%m-%d %H:%M:%S")
            except (ValueError, TypeError):
                f_formt = dt.datetime.strptime(
                    "2018-12-31 00:00:00", "%Y-%m-%d %H:%M:%S"
                )
            fechas_nuevas.append(f_formt)
        archivo[column_name] = fechas_nuevas

```

8. **delete special**: Se encarga de eliminar las tildes y de los caracteres especiales (menos la ñ y la "-") de las columnas especificadas de todas las tablas posibles. Esta función se diseño con ayuda de ChatGPT. Código:

```
# Elimina tildes y caracteres especiales
def delete_special(df: dict):
    # Columnas donde se eliminan los caracteres especiales
    lista_tildes = [
        "DESC_CLASIFICACION",
        "BARRIO",
        "DISTRITO",
        "NOMBRE",
        "TIPO_INCIDENTE",
        "GRAVEDAD",
        "TIPO_INTERVENCION",
        "DIRECCION_AUX",
    ]
    # Diccionario para reemplazar letras con tildes
    replacements = {
```

```
        "á": "a",
        "é": "e",
        "í": "i",
        "ó": "o",
        "ú": "u",
        "Á": "A",
        "É": "E",
        "Í": "I",
        "Ó": "O",
        "Ú": "U",
    }
    for tabla_n in df:
        tabla = df[tabla_n]
        for columna in tabla:
            if columna in lista_tildes:
                for accented_char, unaccented_char in replacements.items():
                    tabla[columna] = tabla[columna].str.replace(
                        accented_char, unaccented_char
                    )
                tabla[columna] = tabla[columna].str.replace(
                    r"[^a-zA-Z0-9 ñÑ-]", "", regex=True
                )

```

9. **formato tlf**: Cambia el formato de los números de teléfono a int (ej.: pasa de "+xx xxx xxx xxx" a xxxxxxxxxxx). Código:

```
# Covierte todos los números de teléfono al formato int (número de 11 cifras)
def formato_tlf(df: dict):
    columna = df["Usuarios"]["TELEFONO"]
    for i in range(len(columna)):
        valor = columna[i]
        if " " or "+" in valor:
            espacio = ""
            j = 0
            while "34" not in espacio:
                espacio += valor[j]
                j += 1
            numero = valor[j:]
            lista = numero.split(" ")
            new_tlf = "34"
            for x in lista:
                new_tlf += x
            columna[i] = new_tlf

```

10. **no duplicates**: Elimina las filas duplicadas de cada tabla. Primero las que tienen valores iguales (sin contar el id) y luego las que tienen mismo id. Código:

```
# Elimina duplicados
def no_duplicates(df: dict):
    for tabla_n in df:
        if tabla_n == "meteo24" or tabla_n == "Codigo":
            continue
        else:
            columnas_sin_primary = list(df[tabla_n].columns)
            primary_key = columnas_sin_primary.pop(0)
            # quitar las filas que repitan todo menos la primary key
            df[tabla_n] = df[tabla_n].drop_duplicates(
```

```
                subset=columnas_sin_primary, keep="first"
            )
            # quitar las filas que repitan la primary key
            df[tabla_n] = df[tabla_n].drop_duplicates(
                subset=[primary_key], keep="first"
            )

```

11. **adjust gps**: Ajusta las coordenadas gps para que concuerden todas y para poder relacionar áreas con juegos. Las coordenadas no pertenencientes a los que indica la función se sustituyen por la medianan de la lista (cumple las condiciones de la función). Código:

```
def adjust_gps(df: dict) -> None:
    """Limpia los datos del GPS"""
    # Comprueba que la latitud esta entre [-90, 90] y la longitud entre [-180, 180]
    lat_area = df["Areas"]["LATITUD"]
    long_area = df["Areas"]["LONGITUD"]
    lat_juego = df["Juegos"]["LATITUD"]
    long_juego = df["Juegos"]["LONGITUD"]
    # si el valor es erroneo entonces coge la mediana
    for i in range(len(lat_area)):
        if abs(lat_area[i]) > 90:
            lat_area[i] = lat_area.median()
    for i in range(len(long_area)):
        if abs(long_area[i]) > 180:
            long_area[i] = long_area.median()
    for i in range(len(lat_juego)):
        if abs(lat_juego[i]) > 90:
            lat_juego[i] = lat_juego.median()
    for i in range(len(long_area)):
        if abs(long_juego[i]) > 180:
            long_juego[i] = long_juego.median()
    # Ajusta las áreas a 3 decimales para que los juegos puedan pertenecer a estas.
    df["Areas"]["LATITUD"] = df["Areas"]["LATITUD"].apply(lambda x: "{:3.3f}".format(x))
    df["Areas"]["LONGITUD"] = df["Areas"]["LONGITUD"].apply(
        lambda x: "{:3.3f}".format(x)
    )
    # adjust Juegos
    df["Juegos"]["LATITUD"] = df["Juegos"]["LATITUD"].apply(
        lambda x: "{:3.3f}".format(x)
    )
    df["Juegos"]["LONGITUD"] = df["Juegos"]["LONGITUD"].apply(
        lambda x: "{:3.3f}".format(x)
    )

```

12. **adjust ETRS89**: Modifica las coordenadas ETRS89 de las áreas y los juegos para convertirlas en GPS. Código:

```
def adjust_ETRS89(df: dict) -> None:
    """Modifica los datos ETRS89 a GPS."""

    # Ajusta áreas
    df["Areas"]["COORD_GIS_X"] = df["Areas"]["COORD_GIS_X"].apply(
        lambda x: "{:3.3f}".format(x % 180)
    )
    df["Areas"]["COORD_GIS_Y"] = df["Areas"]["COORD_GIS_Y"].apply(
        lambda x: "{:3.3f}".format(x % 90)
    )

```

```
    # Ajusta juegos
    df["Juegos"]["COORD_GIS_X"] = df["Juegos"]["COORD_GIS_X"].apply(
        lambda x: "{:3.3f}".format(x % 180)
    )
    df["Juegos"]["COORD_GIS_Y"] = df["Juegos"]["COORD_GIS_Y"].apply(
        lambda x: "{:3.3f}".format(x % 90)
    )

```

13. **enum checker**: Verifica los enum de las tablas. Código:

```
# Verifica los enum básicos
def enum_checker(db: dict):
    for n_tabla in db:
        tabla = db[n_tabla]
        for n_columna in tabla:
            new_columna = []
            columna = tabla[n_columna]

            if n_tabla == "Areas" and n_columna == "ESTADO":
                for valor in columna:
                    if valor != "OPERATIVO":
                        new_columna.append(None)
                    else:
                        new_columna.append(valor)

            elif n_tabla == "Juegos" and n_columna == "ESTADO":
                for valor in columna:
                    if valor not in ["OPERATIVO", "REPARACION"]:
                        new_columna.append(None)
                    else:
                        new_columna.append(valor)

            elif n_tabla == "Incidencias" and n_columna == "TIPO_INCIDENCIA":
                for valor in columna:
                    if valor not in [
                        "DESGASTE",
                        "ROTURA",
                        "VANDALISMO",
                        "MAL FUNCIONAMIENTO",
                    ]:
                        new_columna.append(None)
                    else:
                        new_columna.append(valor)

            elif n_tabla == "Incidencias" and n_columna == "ESTADO":
                for valor in columna:
                    if valor not in ["ABIERTA", "CERRADA"]:
                        new_columna.append(None)
                    else:
                        new_columna.append(valor)

            elif n_tabla == "Mantenimiento" and n_columna == "TIPO_INTERVENCION":
                for valor in columna:
                    if valor not in ["CORRECTIVO", "EMERGENCIA", "PREVENTIVO"]:
                        new_columna.append(None)
                    else:
                        new_columna.append(valor)

            elif n_tabla == "Mantenimiento" and n_columna in [
```

```
                "ESTADO_PREVIO",
                "ESTADO_POSTERIOR",
            ]:
                for valor in columna:
                    if valor not in ["MALO", "REGULAR", "BUENO"]:
                        new_columna.append(None)
                    else:
                        new_columna.append(valor)

            elif n_tabla == "Incidentes" and n_columna == "TIPO_INCIDENTE":
                for valor in columna:
                    if valor not in [
                        "ROBO",
                        "CAIDA",
                        "VANDALISMO",
                        "ACCIDENTE",
                        "DAÑO ESTRUCTURAL",
                    ]:
                        new_columna.append(None)
                    else:
                        new_columna.append(valor)

            elif n_tabla == "Incidentes" and n_columna == "GRAVEDAD":
                for valor in columna:
                    if valor not in ["ALTA", "BAJA", "MEDIA", "CRITICA"]:
                        new_columna.append(None)
                    else:
                        new_columna.append(valor)

            if len(new_columna) > 0:
                tabla[n_columna] = new_columna
                tabla.dropna(subset=n_columna)

```

14. **nif status**: Verifica el campo nif. Código:

```
# Verifica el campo NIF
def nif_status(db: dict):
    columna = db["Usuarios"]["NIF"]
    new_columna = []
    for elemento in columna:
        if "-" not in elemento:
            new_columna.append(None)
        else:
            lista = elemento.split("-")
            if len(lista) != 3:
                new_columna.append(None)
            else:
                if (
                    not lista[0].isdigit()
                    or not lista[1].isdigit()
                    or not lista[2].isdigit()
                ):
                    new_columna.append(None)
                else:
                    new_columna.append(elemento)
    db["Usuarios"]["NIF"] = new_columna
    db["Usuarios"].dropna(subset=["NIF"])

```

15. **check id**: Verifica los id de cada tabla. Código:

```python
# Verifica los id en cada tabla
def check_id(db: dict):
    lista = [
        "Areas",
        "Juegos",
        "Encuestas",
        "Incidencias",
        "Incidentes",
        "Mantenimiento",
    ]
    for nombre in lista:
        columna = db[nombre]["ID"]
        new_columna = []
        if nombre == "Mantenimiento":
            n = 1
            for valor in columna:
                expected = f"-{n},00\xa0MNT"
                if expected != valor:
                    new_columna.append(None)
                else:
                    valor = f"-{n} MNT"
                    new_columna.append(valor)
                n += 1
        else:
            for valor in columna:
                if not isinstance(valor, int):
                    new_columna.append(None)
                else:
                    new_columna.append(valor)
        db[nombre]["ID"] = new_columna
        db[nombre].dropna(subset=["ID"])

```

16. **incidencias status**: Verifica los ids y nif dentro de incidencias. Código:

```python
# Verifica los ids y nif en incidencias
def incidencias_status(db: dict):
    # Crear copias de los conjuntos de búsqueda para optimizar el acceso
    ids = set(db["Mantenimiento"]["ID"])
    nifs = set(db["Usuarios"]["NIF"])

    # Crear una copia del DataFrame "Incidencias"
    tabla = db["Incidencias"].copy()

    # Procesar la columna "MantenimientoID"
    mantenimiento_ids = []
    for valor in tabla["MantenimientoID"]:
        if valor.startswith("[") and valor.endswith("]"):
            contenido = valor[1:-1]
            partes = contenido.split(", ")
            partes_filtradas = []
            for parte in partes:
                if (
                    parte.startswith("'MNT-")
                    and parte.endswith("'")
                    and parte[5:-1].isdigit()
                ):
                    numero = str(
```

```
                        int(parte[5:-1])
                    )  # Convertir a entero y luego a cadena para quitar ceros iniciales
                    expected = f"-{numero} MNT"
                    if expected in ids:
                        partes_filtradas.append(expected)
            if partes_filtradas:
                mantenimiento_ids.append(f"[{', '.join(partes_filtradas)}]")
            else:
                mantenimiento_ids.append(None)
        else:
            mantenimiento_ids.append(None)

    # Asignar los valores filtrados a la columna "MantenimientoID"
    tabla["MantenimientoID"] = mantenimiento_ids

    # Eliminar las filas con valores None en "MantenimientoID"
    tabla = tabla.dropna(subset=["MantenimientoID"])

    # Procesar la columna "UsuarioID"
    usuario_ids = []
    for valor in tabla["UsuarioID"]:
        if valor.startswith("[") and valor.endswith("]"):
            contenido = valor[1:-1]
            partes = contenido.split(", ")
            partes_filtradas = [parte for parte in partes if parte[1:-1] in nifs]
            if partes_filtradas:
                usuario_ids.append(f"[{', '.join(partes_filtradas)}]")
            else:
                usuario_ids.append(None)
        else:
            usuario_ids.append(None)

    # Asignar los valores filtrados a la columna "UsuarioID"
    tabla["UsuarioID"] = usuario_ids

    # Eliminar las filas con valores None en "UsuarioID"
    tabla = tabla.dropna(subset=["UsuarioID"])

    # Asignar el DataFrame modificado de nuevo en el DataFrame original
    db["Incidencias"] = tabla

```

El resto de limpieza se relizó directamente en los csv.

1.4. Proceso de implementación de columnas

En el archivo de python imputation.py se crearon las funciones para eliminar/añadir columnas de tablas. Para ejecutar este archivo, es necesario importar estas librerías de esta manera:

```
import pandas as pd
import random

```

Las funciones son las siguientes:

1. **check list**: Verifica si el *Punto Muestreo* pasado es correcto. Se tiene en cuenta también si hace referencia a vientos (acaba en 81), temperaturas (acaba en 83) o precipitaciones (acaba en 89). Si termina en cualquier otr cifrs se considerará erroneo. Es útil para la función de restructuración de meteo, que se explicará después. Código:

```python
def check_list(elemento: str, codes) -> list | None:
    # verifica si el punto muestreo es correcto
    if "_" not in elemento:
        return None
    data = elemento.split("_")
    if len(data) != 3:
        return None
    if not data[0].isdigit() or not data[1].isdigit():
        return None
    #Sólo se sacan vientos (81), temperaturas (83) y precipitaciones (89)
    if data[0] not in codes.keys() or data[1] not in ["81", "83", "89"]:
        return None
    return data

```

2. **same line**: También útil para la restructuración de la tabla meteo. Se encarga de identificar una medición según la *FECHA* y el *DISTRITO*. En caso de no existir dicha medición (no existe una linea con ambos valores) devuelve -1. Código:

```python
def same_line(fecha: str, distrito: str, list_f: list, list_d: list) -> int:
    #Identifica una linea donde fecha sea su FECHA y distrito su DISTRITO.
    if fecha not in list_f or distrito not in list_d:
        return -1
    for n in range(0, len(list_f)):
        if list_f[n] == fecha and list_d[n] == distrito:
            return n
    return -1 #Si no hay, devuelve -1

```

3. **post code**: A partir de un código postal pasado como argumento se verifica que existe una estación meteorológica (tabla *Codigo*) con un código postal igual. Si existe se devuelve la posición dentro de la tabla *Codigo*. Código:

```python
def post_code(db: pd.DataFrame, codigo: int):
    #Identifica si el código postal pasado es igual al de una estación meteorológica.
    code_id = db["Codigo"]["CÓDIGO"]
    for n in range(len(code_id)):
        if str(code_id[n]) == str(codigo):
            return n #Posición dentro de Código
    return -1 #No existe

```

4. **new meteo**: Se modifica totalmente la tabla *meteo24*. Se crean las siguientes columnas:

   a) ID: Se crea según la posición de la fila que le toque.

   b) CODIGO POSTAL: Código postal del lugar al que pertenece. Todos tienen 1 excepto Plaza de España que tiene 2.

   c) FECHA: Fecha tipo YYYY-MM-DD hh:mm:ss.

   d) VIENTO: Booleano que indica si dicho día hay vientos fuertes. Nos hemos basado en la velocidad (>=20).

   e) TEMPERATURA: Indica si hay medición de temperatura y cuál.

   f) PRECIPITACION: Indica si hay medición de precipitación y cuál.

   g) DISTRITO: Indica el distrito al que pertenece la estación meteorológica.

La función trata de verificar el código postal, identificar el distrito y la fecha, insertar si no existe dicha combinación y luego insertar los datos faltantes (viento si PUNTO MUESTREO termina en 81, temperatura si termina en 83 y percipitacion si termina en 89). En caso de Plaza de España se realiza una doble inserción (hay 2 códigos postales -> 2 filas). Esta función se planteo sin IA pero se corrigió con ChatGPT. Código:

```
def new_meteo(db: pd.DataFrame):
    # Modifica totalmente la tabla de meteo24 para que muestre lo que pide el enunciado
    meteo = db["meteo24"]
    m_postal = db["Codigo"]  # Lista de códigos postales para la relación área-meteo
    col_postal = m_postal["CodigoPostal"]
    codes = {
        "28079102": "MORATALAZ",
        "28079103": "VILLAVERDE",
        "28079104": "PUENTE DE VALLECAS",
        "28079106": "MONCLOA-ARAVACA",
        "28079107": "HORTALEZA",
        "28079108": "FUENCARRAL-EL PARDO",
        "28079109": "CHAMBERI",
        "28079110": "CENTRO",
        "28079111": "CHAMARTIN",
        "28079112": "VILLA DE VALLECAS",
        "28079113": "VILLA DE VALLECAS",
        "28079114": "ARGANZUELA",
        "28079115": "ARGANZUELA",
        "28079004": "MONCLOA-ARAVACA",
        "28079008": "SALAMANCA",
        "28079016": "CIUDAD LINEAL",
        "28079018": "CARABANCHEL",
        "28079024": "MONCLOA-ARAVACA",
        "28079035": "CENTRO",
        "28079036": "MORATALAZ",
        "28079038": "TETUAN",
        "28079039": "FUENCARRAL-EL PARDO",
        "28079054": "VILLA DE VALLECAS",
        "28079056": "CARABANCHEL",
        "28079058": "FUENCARRAL-EL PARDO",
        "28079059": "BARAJAS",
    }  # Diccionario de distritos
    meses = {
        "1": 31,
        "2": 28,
        "3": 31,
        "4": 30,
        "5": 31,
        "6": 30,
        "7": 31,
        "8": 31,
        "9": 30,
        "10": 31,
        "11": 30,
        "12": 31,
    }  # Diccionario de meses (como la fecha es por día, se necesita saber qué días hay que
    coger)
    ids = []  # Cada id representa la posición de la fila de forma ordenada
    codigo_postal = []
    distritos = []
    temperaturas = []
    precipitaciones = []
    viento = []
    fechas = []
```

```
    id = 1
    for i in range(len(meteo["PUNTO_MUESTREO"])):
        elemento = meteo["PUNTO_MUESTREO"][i]
        data = check_list(elemento, codes)  # Verifica el PUNTO MUESTREO y el MES
        if data is not None and str(meteo["MES"][i]) in meses.keys():
            for j in range(
                meteo["MES"][i]
            ):  # Por cada fecha y distrito, una fila distinta
                dia = str(j + 1)
                mes = str(meteo["MES"][i])
                if j < 9:
                    dia = "0" + dia
                if int(mes) < 10:
                    mes = "0" + mes
                fecha = str(meteo["ANO"][i]) + "-" + mes + "-" + dia
                n_code = post_code(db, data[0])  # Verifica el código postal
                if n_code != -1:
                    # Si no existe una fila con dicha fecha y distrito, se crea una nueva
                    linea = same_line(fecha, codes[data[0]], fechas, distritos)
                    if linea == -1:
                        ids.append(id)
                        id += 1
                        pcode = col_postal[n_code]
                        codigo_postal.append(int(pcode))
                        distritos.append(codes[data[0]])
                        fechas.append(fecha)
                        temperaturas.append("-")
                        precipitaciones.append("-")
                        viento.append(False)
                        linea = len(distritos) - 1
                        if data[0] == "28079004":
                            # Como Plaza de España tiene dos códigos postales, hay doble
    inserción
                            ids.append(id)
                            id += 1
                            distritos.append(codes[data[0]])
                            pcode = col_postal[n_code + 1]
                            codigo_postal.append(int(pcode))
                            fechas.append(fecha)
                            temperaturas.append("-")
                            precipitaciones.append("-")
                            viento.append(False)
                            # Aquí se tiene en cuenta la línea anterior
                    booleano = (
                        "V" + dia
                    )  # Para modificar el valor, el booleano debe ser igual a V
                    valor = "D" + dia  # Datos a insertar
                    if data[1] == "81":
                        if meteo[booleano][i] == "V" and int(meteo[valor][i]) >= 20:
                            viento[linea] = True
                            if (
                                data[0] == "28079004"
                            ):  # Plaza de España -> Doble inserción
                                viento[linea + 1] = True
                        else:
                            viento[linea] = False
                            if (
                                data[0] == "28079004"
                            ):  # Plaza de España -> Doble inserción
                                viento[linea + 1] = False
```

```
                    elif data[1] == "83":
                        if meteo[booleano][i] == "V":
                            temperaturas[linea] = meteo[valor][i]
                            if (
                                data[0] == "28079004"
                            ):  # Plaza de España -> Doble inserción
                                temperaturas[linea + 1] = meteo[valor][i]
                    else:
                        if meteo[booleano][i] == "V":
                            precipitaciones[linea] = meteo[valor][i]
                            if (
                                data[0] == "28079004"
                            ):  # Plaza de España -> Doble inserción
                                precipitaciones[linea + 1] = meteo[valor][i]

    nuevo = pd.DataFrame()  # Se crea la nueva tabla
    nuevo.loc[:, "ID"] = ids
    nuevo.loc[:, "CODIGO_POSTAL"] = codigo_postal
    nuevo.loc[:, "FECHA"] = fechas
    pd.to_datetime(nuevo["FECHA"]).apply(lambda x: x.date())
    nuevo.loc[:, "TEMPERATURA"] = temperaturas
    nuevo.loc[:, "PRECIPITACION"] = precipitaciones
    nuevo.loc[:, "VIENTO"] = viento
    nuevo.loc[:, "DISTRITO"] = distritos
    db["meteo24"] = nuevo

```

5. **are new atributte**: Se añade una columna *AreaID* a la tabla *Juegos* para representar la relación áreas-juegos. Primero se realiza compararndo ambas coordenadas y para el resto se le asigna un id random no repetido. Código:

```
def area_new_atribute(df: pd.DataFrame):
    """Añade el atributo capacidadMax a Areas."""
    df["Juegos"]["AreaRecreativaID"] = 0
    index = 0
    # Calcula el número de juegos por area con lat y long
    for area in df["Areas"].to_numpy():
        games = 0  # Número de juegos
        # Buscar por GPS
        lat_area = area[10]
        long_area = area[11]
        index_juego = 0
        for juego in df["Juegos"].to_numpy():
            lat_juego = juego[10]
            long_juego = juego[11]
            # Si coinciden, se suma el núemrod e juegos y se añade el areaID a Juegos
            if lat_area == lat_juego and long_area == long_juego:
                games += 1
                df["Juegos"].loc[index_juego, "AreaRecreativaID"] = area[0]
            index_juego += 1
        # Aquí crea la columna
        df["Areas"].loc[index, "capacidadMax"] = games
        index += 1
    # Añadir valor a los juegos sin áreas
    index_juego = 0
    for juego in df["Juegos"].to_numpy():
        ref_area = juego[24]
        # Si no hay referencia del juego en área
        if ref_area == 0:
```

```
            in_column = True
            while in_column:
                # Inserta un valor random no repetido
                rand_id = random.randint(1000000, 100000000)
                if rand_id not in df["Juegos"]["AreaRecreativaID"]:
                    df["Juegos"].loc[index_juego, "AreaRecreativaID"] = rand_id
                    in_column = False
        index_juego += 1

```

6. **juegos new atributtes**: Añade a la tabla *Juegos* las columnas *indicadorExposicion* y *desgasteAcumulado*. El indicador de exposición puede ser 100/Bajo, 200/Medio y 300/Alto y se escoge de manera aleatoria. El desgaste acumulado es un valor que se calcula del número de años de uso (entero aleatorio entre 1 y 15), el indicador de exposición y el número de juegos mantenidos. Código:

```
def juegos_new_atributes(df: pd.DataFrame):
    """Añade las columnas indicadorExposicion y desgasteAcumulado a Juegos"""
    # Inserta el valor de indicador exposicion
    indicador_options = {"BAJO": 100, "MEDIO": 200, "ALTO": 300}
    for i in range(len(df["Juegos"])):
        selected_option = random.randint(0, 2)
        df["Juegos"].loc[i, "indicadorExposicion"] = list(indicador_options.keys())[
            selected_option
        ]
    # Calcula el valor de desgasteAcumulado
    index = 0
    wear_values = []
    for juego in df["Juegos"].to_numpy():
        # Número del mantenimiento al juego
        num_mant_juego = 0
        # Uso aleatorio puede ser un entero del 1 al 15
        use_time = random.randint(1, 15)
        for maintenance in df["Mantenimiento"].to_numpy():
            if juego[0] == maintenance[5]:
                num_mant_juego += 1
        # Añadir el wear value a una lista para cambiar su rango
        wear_value = (
            use_time * indicador_options[df["Juegos"]["indicadorExposicion"][index]]
        ) - (num_mant_juego * 100)
        wear_values.append(wear_value)
        index += 1
    # Inserta desgasteAcumulado
    for i in range(len(df["Juegos"])):
        df["Juegos"].loc[i, "desgasteAcumulado"] = adjust_range(
            wear_values, 0, 100, i
        )  # Cambia el rango de sus valores

```

7. **adjust range**: Ajusta un valor de una lista a unos nuevos límites. Código:

```
def adjust_range(values: list, new_min: int, new_max: int, index: int) -> int:
    # Fórmula para ajustar un valor dentro de una lista a unos nuevos límites
    old_min = min(values)
    old_max = max(values)
    new_value = int(
        (((values[index] - old_min) * (new_max - new_min)) / (old_max - old_min))
        + new_min
    )
```

```
    return new_value

```

8. **takeTimebyID**: Coge una fecha de la tabla *Mantenimiento* según un id. Código:

```
def takeTimebyID(base: pd.DataFrame, id: str):
    # Coge una fecha de mantenimiento desde su id
    numero = int(id[1:-4])
    lista = base["Mantenimiento"]["ID"]
    if numero < len(lista):
        return base["Mantenimiento"]["FECHA_INTERVENCION"][numero]
    return None

```

9. **tiempoResolucion**: Se calcula la diferencia de tiempo (en días) entre la fecha de mantenimiento y la de la incidencia. Para realizar dicha resta, ambos mantenimiento e incidencia deben tener el mismo id de mantenimiento, además de que exita la fecha de mantenimiento y que sea mayor a la de la incidencia. Cabe aclarar que, como hay varios id de mantenimiento para una incidencia, se escogerá el tiempo mayor. Toda esta información se alamecenará en la nueva columna *tiempoResolucion* en la tabla *Incidencias*. Esta funcións e hizo con ayuda de chatGPT. Código:

```
def tiempoResolucion(base: pd.DataFrame):
    # Calcula el tiempo entre la incidencia y el mantenimiento
    col_index = base["Incidencias"]["MantenimientoID"]
    # Importante: La incidencia debe contener el id de dicho mantenimiento
    tiempos = []
    for n in range(len(col_index)):
        elemento = col_index[n]
        conjunto = elemento[1:-1]
        conjunto = conjunto.split(", ")
        tiempo = float(0)  # Tiempo predeterminado
        fecha_incidencia = base["Incidencias"]["FECHA_REPORTE"][n]
        for valor in conjunto:
            fecha_mat = takeTimebyID(base, valor)
            # Al tener ambas fechas, se hace la resta si la de mantenimiento es más tardía que
 la de la incidencia
            if fecha_mat != None and fecha_mat > fecha_incidencia:
                sub_t = fecha_mat - fecha_incidencia

                # Convertir sub_t a segundos para poder compararlo con el valor float 'tiempo'
                sub_t_seconds = sub_t.total_seconds()
                if sub_t_seconds > tiempo:
                    tiempo = sub_t_seconds
        if tiempo != 0:
            tiempo = tiempo // (60 * 60 * 24)  # Pasar a días
        tiempos.append(tiempo)
    base["Incidencias"].loc[:, "TIEMPO_RESOLUCION"] = tiempos

```

10. **lastFecha**: Identifica las últimas fechas de mantenimiento de cada juego y las almacena en la nueva columna *ULTIMA FECHA MANTENIMIENTO* de la tabla *Juegos*. Código:

```
def lastFecha(db: pd.DataFrame):
    # Seleccionamos el último mantenimiento para cada juego en la tabla de mantenimiento
    max_mant = (
        db["Mantenimiento"]
        .groupby("JuegoID")["FECHA_INTERVENCION"]
```

```
        .max()
        .reset_index()
        .rename(columns={"FECHA_INTERVENCION": "ULTIMA_FECHA_MANTENIMIENTO"})
    )

    # Fusionamos las fechas de mantenimiento con la tabla de juegos
    db["Juegos"] = db["Juegos"].merge(
        max_mant, left_on="ID", right_on="JuegoID", how="left"
    )

    # Rellenamos con FECHA_INSTALACION en caso de que no haya mantenimientos
    db["Juegos"]["ULTIMA_FECHA_MANTENIMIENTO"].fillna(
        db["Juegos"]["FECHA_INSTALACION"], inplace=True
    )

```

11. **area meteo**: Implementa la columna *MeteoID* a area para identificar a que estación meteorológica pertenece a cada área. Código:

```
def area_meteo(db: pd.DataFrame):
    # Inserta en Áreas el id de la meteo24 que pertenece
    lista_meteo = []  # Columna de los id de meteo
    tabla_area = db["Areas"]
    tabla_meteo = db["meteo24"]
    columna_cod_postal = tabla_area["COD_POSTAL"]
    columna_cod_meteo = tabla_meteo["CODIGO_POSTAL"]

    # Crear un diccionario para el mapeo de códigos postales a ID
    codigo_to_id = {
        columna_cod_meteo[i]: tabla_meteo["ID"][i]
        for i in range(len(columna_cod_meteo))
    }

    for valor in columna_cod_postal:
        estacion = 0  # Valor predeterminado

        # Asegurarse de que ambos valores son enteros
        if pd.notnull(valor) and type(valor) is not str:  # Evita valores nulos
            valor = int(valor) if not isinstance(valor, int) else valor
            estacion = codigo_to_id.get(
                valor, estacion
            )  # Obtener el valor o usar predeterminado

        lista_meteo.append(estacion)

    # Asignar la lista resultante a una nueva columna "MeteoID" en db["Areas"]
    db["Areas"]["MeteoID"] = lista_meteo

```

12. **nivelEscalamiento**: Crea la columna *NIVEL RECONOCIMIENTO* en *Incidencias* y se asigna un valor según si la incidencia está abierta (aleatorio del 1 al 10) o cerrada (1). Código:

```
def nivelEscalamiento(db: pd.DataFrame):
    # Inserta el nivel de escalamiento. Si es abierto -> aleatorio del 1 al 10. Si no, es 1.
    incidencias = db["Incidencias"]
    lista_rec = []
    for n in range(len(incidencias["ESTADO"])):
        if incidencias["ESTADO"][n] == "ABIERTA":
            numero = random.randint(1, 10)
```

```
            lista_rec.append(numero)
        else:
            lista_rec.append(1)
    incidencias.loc[:, "NIVEL_RECONOCIMIENTO"] = lista_rec

```

1.5. Proceso de extración a mongoDB

Con el archivo "formatear mongo.py" se extraen los objetos y se crean las tablas limpias para mongoDB. Para ejecutar este archivo, es necesario importar estas librerías de esta manera:

```
import json

```

Todas estas funciones se encuentran en la clase *Creator*, que destaca por este init:

```
class Creator:
    def __init__(self, bd: dict):

        # Base de datos en el estado requerido
        self.state = bd

        # Correspondencias entre columna a añadir y tablas que la requieren
        self.col_table = {
            "AreaRecreativaID": "meteo24",
            "JueID": "Incidencias",
        }

        # relaciones entre sí
        self.juego_tipo = {}
        self.juego_incidencias = {}
        self.area_meteo = {}
        self.area_encuesta = {}
        self.area_incidente = {}
        self.area_juegos = {}
        self.juego_mantenimientos = {}
        self.mantenimiento_incidencias = {}
        self.incidencia_usuario = {}

        # objetos_extraidos
        self.areas = []
        self.encuestas = []
        self.registrosClima = []
        self.incidentesSeguridad = []
        self.juegos = []
        self.incidencias = []
        self.usuarios = []
        self.mantenimientos = []


```

Además, contiene las siguientes funciones de generación de objetos:

1. **get json data**: Genera los JSON de los objetos extraidos. Además, toda fecha que encuentre le cambia al formato ISO 8601. Código:

```
    def get_json_data(self, objs: list, collection_name: str):

        # Dar formato ISO 8601 a fechas
```

```
        for obj in objs:
            for key in obj:
                if "fecha" in key:
                    obj[key] = (
                        f"{str(obj[key]).split(" ")[0]}T{str(obj[key]).split(" ")[1]}Z"
                    )

        # abrir fichero y escribir datos
        with open(f"DatasetsNuevos/{collection_name}.js", "w") as file:
            file.write(
                f"""db.{collection_name}.insertMany(
                    { json.dumps(objs,indent=4) }
                )"""
            )
        print(f"JSON creado para {collection_name}")

```

2. **generar usuarios**: Extrae los objetos de la tabla *Usuarios*. Código:

```
    def generar_usuarios(self):

        # extraer columnas de datos usuarios
        nifs = self.extraer_columna("Usuarios", "NIF")
        nombre = self.extraer_columna("Usuarios", "NOMBRE")
        email = self.extraer_columna("Usuarios", "EMAIL")
        tlf = self.extraer_columna("Usuarios", "TELEFONO")

        for i in range(len(nifs)):
            self.usuarios.append(
                {
                    "NIF": str(nifs[i]),
                    "nombre": str(nombre[i]),
                    "email": str(email[i]),
                    "telefono": str(tlf[i]),
                }
            )

```

3. **generar encuestas**: Extrae los objetos de la tabla *Encuestas*. Código:

```
    def generar_encuestas(self):

        id = self.extraer_columna("Encuestas", "ID")
        fecha = self.extraer_columna("Encuestas", "FECHA")
        accesibilidad = self.extraer_columna("Encuestas", "PUNTUACION_ACCESIBILIDAD")
        calidad = self.extraer_columna("Encuestas", "PUNTUACION_CALIDAD")
        comantarios = self.extraer_columna("Encuestas", "COMENTARIOS")

        for i in range(len(id)):
            self.encuestas.append(
                {
                    "id": int(id[i]),
                    "fechaEncuesta": str(fecha[i]),
                    "puntuacionAccesibilidad": accesibilidad[i],
                    "puntuacionCalidad": calidad[i],
                    "comentarios": comantarios[i],
                }
            )

```

4. **generar incidentes**: Extrae los objetos de la tabla *Incidentes*. Código:

```
def generar_incidentes(self):

        id = self.extraer_columna("Incidentes", "ID")
        fecha = self.extraer_columna("Incidentes", "FECHA_REPORTE")
        tipo = self.extraer_columna("Incidentes", "TIPO_INCIDENTE")
        gravedad = self.extraer_columna("Incidentes", "GRAVEDAD")

        for i in range(len(id)):
            self.incidentesSeguridad.append(
                {
                    "id": int(id[i]),
                    "fechaDeReporte": str(fecha[i]),
                    "tipoIncidente": tipo[i],
                    "gravedad": gravedad[i],
                }
            )

```

5. **generar incidencias**: Extrae los objetos de la tabla *Incidencias*. Código:

```
    def generar_incidencias(self):

        id = self.extraer_columna("Incidencias", "ID")
        tipo = self.extraer_columna("Incidencias", "TIPO_INCIDENCIA")
        fecha = self.extraer_columna("Incidencias", "FECHA_REPORTE")
        estado = self.extraer_columna("Incidencias", "ESTADO")
        tiempo = self.extraer_columna("Incidencias", "TIEMPO_RESOLUCION")
        escala = self.extraer_columna("Incidencias", "NIVEL_RECONOCIMIENTO")

        for i in range(len(id)):
            if id[i] in self.incidencia_usuario:

                usuarios = self.incidencia_usuario[int(id[i])]

                # encontrar usuarios en la relación incidencia-usuario
                res_users = []
                for user in self.usuarios:
                    for usuarioID in usuarios:
                        if user["NIF"] == usuarioID:
                            res_users.append(user)

                self.incidencias.append(
                    {
                        "id": int(id[i]),
                        "fechaReporte": str(fecha[i]),
                        "tipo": tipo[i],
                        "estado": estado[i],
                        "tiempoResolucion": float(tiempo[i]),
                        "nivelEscalamiento": int(escala[i]),
                        "usuarios": res_users,
                    }
                )

```

6. **generar juegos**: Extrae los objetos de la tabla *Juegos*. Código:

```
    def generar_juegos(self):
```

```
        id = self.extraer_columna("Juegos", "ID")
        modelo = self.extraer_columna("Juegos", "MODELO")
        estado_op = self.extraer_columna("Juegos", "ESTADO")
        accesibilidad = self.extraer_columna("Juegos", "ACCESIBLE")
        fecha_instalacion = self.extraer_columna("Juegos", "FECHA_INSTALACION")
        tipo = self.extraer_columna("Juegos", "tipo_juego")
        desgaste = self.extraer_columna("Juegos", "desgasteAcumulado")
        indicador_exposicion = self.extraer_columna("Juegos", "indicadorExposicion")
        ultima_fecha_mant = self.extraer_columna("Juegos", "ULTIMA_FECHA_MANTENIMIENTO")

        for i in range(len(id)):
            # si el juego no tiene incidencias
            if id[i] not in list(self.juego_incidencias.keys()):
                incidencia = []
            else:
                incidencia = self.juego_incidencias[id[i]]
            # si el juego no tiene mantenimientos
            if id[i] not in list(self.juego_mantenimientos.keys()):
                mantenimiento = [""]
            else:
                mantenimiento = self.juego_mantenimientos[id[i]]
            self.juegos.append(
                {
                    "id": str(id[i]),
                    "modelo": str(modelo[i]),
                    "estadoOperativo": str(estado_op[i]),
                    "accesibilidad": bool(accesibilidad[i]),
                    "fechaInstalacion": str(fecha_instalacion[i]),
                    "tipo": str(tipo[i]),
                    "desgasteAcumulado": int(desgaste[i]),
                    "indicadorExposicion": str(indicador_exposicion[i]),
                    "ultimaFechaMantenimiento": str(ultima_fecha_mant[i]),
                    "mantenimientos": mantenimiento,
                    "incidencias": incidencia,
                }
            )

```

7. **generar clima**: Extrae los objetos de la tabla *meteo24*. Código:

```
    def generar_clima(self):
        id = self.extraer_columna("meteo24", "ID")
        fecha = self.extraer_columna("meteo24", "FECHA")
        temp = self.extraer_columna("meteo24", "TEMPERATURA")
        vent = self.extraer_columna("meteo24", "VIENTO")
        prec = self.extraer_columna("meteo24", "PRECIPITACION")
        for i in range(len(id)):
            # Solo pasa a entero los valores numéricos, no los -
            if temp[i] != "-":
                temp[i] = int(temp[i])
            else:
                temp[i] = 0
            if prec[i] != "-":
                prec[i] = int(prec[i])
            else:
                prec[i] = 0

            self.registrosClima.append(
                {
                    "id": int(id[i]),
```

```
                    "fecha": fecha[i],
                    "temperatura": temp[i],
                    "vientosFuertes": bool(vent[i]),
                    "precipitacion": prec[i],
                }
            )

```

8. **generar area**: Extrae los objetos de la tabla *Areas*. Sólo se tienen en cuenta las columnas pedidas en la práctica. Código:

```
    def generar_area(self):
        id = self.extraer_columna("Areas", "ID")
        barrio = self.extraer_columna("Areas", "BARRIO")
        distr = self.extraer_columna("Areas", "DISTRITO")
        estado = self.extraer_columna("Areas", "ESTADO")
        lat = self.extraer_columna("Areas", "LATITUD")
        long = self.extraer_columna("Areas", "LONGITUD")
        fecha = self.extraer_columna("Areas", "FECHA_INSTALACION")
        capmax = self.extraer_columna("Areas", "capacidadMax")

        for i in range(len(id)):
            # Solo pasa a entero los valores numéricos, no los -

            # extraer incidentes de seguridad de la relación
            res_incidentes = []
            if int(id[i]) in self.area_incidente:
                incidentes = self.area_incidente[int(id[i])]
                for inci in self.incidentesSeguridad:
                    for incidentesID in incidentes:
                        if inci["id"] == incidentesID:
                            res_incidentes.append(int(incidentesID))

            # extraer encuestas de la relación
            res_encuestas = []
            if int(id[i]) in self.area_encuesta:
                encuestas = self.area_encuesta[int(id[i])]
                for enc in self.encuestas:
                    for encuestaID in encuestas:
                        if enc["id"] == encuestaID:
                            res_encuestas.append(int(encuestaID))

            # extraer climas de la relación
            res_climas = []
            if int(id[i]) in self.area_meteo:
                climas = self.area_meteo[int(id[i])]
                for clima in self.registrosClima:
                    for climaID in climas:
                        if clima["id"] == climaID:
                            res_climas.append(int(climaID))

            # extraer juegos de la relación
            res_juegos = []
            if int(id[i]) in self.area_juegos:
                juegos = self.area_juegos[int(id[i])]
                for juegoID in juegos:
                    res_juegos.append(int(juegoID))

            # extraer atributo juego_tipo
            res_juegos_tipo_valor = []
```

```
            for tipo in self.juego_tipo:
                objeto = {"tipo": str(tipo), "valor": int(self.juego_tipo[tipo])}
                res_juegos_tipo_valor.append(objeto)

            self.areas.append(
                {
                    "id": int(id[i]),
                    "barrio": barrio[i],
                    "distrito": distr[i],
                    "estadoOperativo": estado[i],
                    "coordenadasGPS": [lat[i], long[i]],
                    "fecha": fecha[i],
                    "capacidadMaxima": float(capmax[i]),
                    "incidentesSeguridad": res_incidentes,
                    "encuestas": res_encuestas,
                    "registrosClima": res_climas,
                    "juegos": res_juegos,
                    "cantidadJuegosTipo": res_juegos_tipo_valor,
                }
            )


```

9. **generar mantenimientos**: Función para extraer los objetos de la tabla *Mantenimiento*. Código:

```
    def generar_mantenimientos(self):

        id = self.extraer_columna("Mantenimiento", "ID")
        tipo = self.extraer_columna("Mantenimiento", "TIPO_INTERVENCION")
        estado_previo = self.extraer_columna("Mantenimiento", "ESTADO_PREVIO")
        estado_posterior = self.extraer_columna("Mantenimiento", "ESTADO_POSTERIOR")
        fecha = self.extraer_columna("Mantenimiento", "FECHA_INTERVENCION")

        transform = lambda x: f"{x.split(" ")[1]}-{int(x.split(" ")[0][1:])}"
        for i in range(len(id)):

            if transform(id[i]) in self.mantenimiento_incidencias:
                incidencias = self.mantenimiento_incidencias[transform(id[i])]

                # encontrar incidencias en la relación mantenimiento-incidencias
                res_incidencias = []
                for inci in self.incidencias:
                    for incidenciasID in incidencias:
                        if inci["id"] == incidenciasID:
                            res_incidencias.append(int(incidenciasID))

                self.mantenimientos.append(
                    {
                        "id": transform(id[i]),
                        "tipoIntervencion": str(tipo[i]),
                        "estadoPrevio": estado_previo[i],
                        "estadoPosterior": estado_posterior[i],
                        "fechaIntervencion": fecha[i],
                        "incidencias": res_incidencias,
                    }
                )


```

10. **extraer columna**: Función para hacer las listas con cada columna de una tabla indicada. Código:

```
    def extraer_columna(self, table: str, column: str) -> list:
        # Extrae las tablas definitivas completas, iterando por sus columnas
        result = []
        for i in range(len(self.state[table][column])):
            result.append(self.state[table].loc[i, column])
        return result

```

11. **crear area clima**: Crea la relación entre las tablas *Areas-meteo24* extrayendo los id de las estaciones metorológicas de *Areas* y analizando si existen en *meteo24*. Código:

```
    def crear_area_clima(self):
        tabla_areas = self.state["Areas"]

        # Extraer las estaciones meteorologicas de la tabla de areas
        for i in range(len(tabla_areas["MeteoID"])):
            valor_meteo = int(tabla_areas.loc[i, "MeteoID"])
            valor_id = int(tabla_areas.loc[i, "ID"])

            if valor_meteo == 0:
                continue

            if valor_meteo not in self.area_meteo:
                self.area_meteo[valor_id] = [valor_meteo]
            else:
                self.area_meteo[valor_id].append(valor_meteo)

```

12. **crear area juegos**: Crea la relación entre las tablas *Areas-Juegos* extrayendo los id de las áreas recreativas de *Juegos* y analizando si existen en *Areas*. Código:

```
    def crear_area_juegos(self):

        tabla_juegos = self.state["Juegos"]

        # Extraer las areas de la tabla de juegos
        for i in range(len(tabla_juegos["AreaRecreativaID"])):
            valor_area = tabla_juegos.loc[i, "AreaRecreativaID"]
            valor_juego = tabla_juegos.loc[i, "ID"]

            if valor_area not in self.area_juegos:
                self.area_juegos[int(valor_area)] = [int(valor_juego)]
            else:
                self.area_juegos[int(valor_area)].append(int(valor_juego))

```

13. **crear juegos tipo**: Identifica los distintos tipos de juegos que existen, contando el número de veces que aparecen en la tabla *Juegos*. Código:

```
    def crear_juegos_tipo(self):

        tabla_juegos = self.state["Juegos"]

        # Extraer los tipos de la tabla de juegos
        for i in range(len(tabla_juegos["tipo_juego"])):
            valor_tipo = tabla_juegos.loc[i, "tipo_juego"]
            if valor_tipo not in self.juego_tipo:
```

```
                self.juego_tipo[valor_tipo] = 1
            else:
                self.juego_tipo[valor_tipo] += 1

```

14. **crear area encuestas**: Crea la relación entre las tablas *Areas-Encuestas* extrayendo los id de las áreas recreativas de *Encuestas* y analizando si existen en *Areas*. Código:

```
    def crear_area_encuestas(self):
        tabla_encuestas = self.state["Encuestas"]

        # Extraer los mantenimientos de las incidencias. Mapa "AreaRecreativaID -> EncuestasID
    "
        for i in range(len(tabla_encuestas["AreaRecreativaID"])):

            valor_area = tabla_encuestas.loc[i, "AreaRecreativaID"]
            valor_id = tabla_encuestas.loc[i, "ID"]

            if valor_area not in self.area_encuesta:
                self.area_encuesta[valor_area] = [valor_id]
            else:
                self.area_encuesta[valor_area].append(valor_id)

```

15. **crear juego incidencias**: Crea la relación entre las tablas *Juegos-Incidencias*. Esto se realiza con la ayuda de la tabla *Mantenimiento* ya que primero se extraen los id de mantenimiento de la tabla *Incidencias* y luego se extraen los id de los juegos de la tabla *Mantenimiento*. La relación se saca analizando qué ids de los sacados realmente existen en sus tablas representativas (id mantenimiento -> *Mantenimiento*, id juego -> *Juegos*. Código:

```
    def crear_juego_incidencias(self):

        # Tablas a utilizar
        tabla_incidencias = self.state["Incidencias"]
        tabla_mantenimiento = self.state["Mantenimiento"]

        # Operación de transformación de nombre de atributos.
        transform = lambda x: f"{x.split(" ")[1]}-{int(x.split(" ")[0][1:])}"

        # Extraer los mantenimientos de las incidencias. Mapa "MantenimientoID -> IncidenciaID
    "
        for i in range(len(tabla_incidencias["MantenimientoID"])):

            id_manten_str = tabla_incidencias.loc[i, "MantenimientoID"]
            id_manten_list = id_manten_str[1:-1].split(", ")
            valor_id = tabla_incidencias.loc[i, "ID"]

            for manten_id in id_manten_list:
                if manten_id not in self.mantenimiento_incidencias:
                    self.mantenimiento_incidencias[transform(manten_id)] = [valor_id]
                else:
                    self.mantenimiento_incidencias[transform(manten_id)].append(
                        valor_id
                    )

        # Extraer los mantenimientos de los juegos. Mapa "JuegoID -> MantenimientoID"
        for i in range(len(tabla_mantenimiento["JuegoID"])):

```

```
            valor_id = tabla_mantenimiento.loc[i, "JuegoID"]
            valor_manten = tabla_mantenimiento.loc[i, "ID"]

            if valor_id not in self.juego_mantenimientos:
                self.juego_mantenimientos[valor_id] = [transform(valor_manten)]
            else:
                self.juego_mantenimientos[valor_id].append(transform(valor_manten))

        # Deducir las relaciones de juegos e incidencias. Mapa "JuegoID -> IncidenciasID"
        for juego in self.juego_mantenimientos.keys():
            for incidencia in self.mantenimiento_incidencias.values():
                for mantenimiento in self.juego_mantenimientos[juego]:

                    if mantenimiento not in self.mantenimiento_incidencias:
                        continue
                    if self.mantenimiento_incidencias[mantenimiento] == incidencia:
                        for inc in incidencia:
                            if juego not in self.juego_incidencias:
                                self.juego_incidencias[juego] = [int(inc)]
                            else:
                                if int(inc) not in self.juego_incidencias[juego]:
                                    self.juego_incidencias[juego].append(int(inc))

```

16. **crear area incidentes**: Crea la relación entre las tablas *Areas-Incidentes* extrayendo los id de las áreas recreativas de *Incidentes* y analizando si existen en *Areas*. Código:

```
    def crear_areas_incidentes(self) -> None:
        """Crear referencia para Areas-Incidentes"""
        tabla_incidentes_seg = self.state["Incidentes"]
        for i in range(len(tabla_incidentes_seg["AreaRecreativaID"])):
            id_area = tabla_incidentes_seg.loc[i, "AreaRecreativaID"]
            id_incidendes_seg = tabla_incidentes_seg.loc[i, "ID"]
            if id_area is not None and id_area not in self.area_incidente:
                self.area_incidente[id_area] = [id_incidendes_seg]
            else:
                self.area_incidente[id_area].append(id_incidendes_seg)

```

17. **crear incidencia usuario**: Crea la relación entre las tablas *Incidencias-Usuarios* extrayendo los id de los usuarios de *Incidencias* y analizando si existen en *Usuarios*. Código:

```
    def crear_incidencia_usuario(self) -> None:
        """Crear referencia para Usuario-Incidencias"""
        tabla_incidencias = self.state["Incidencias"]
        for i in range(len(tabla_incidencias["UsuarioID"])):
            id_usuarios_str = tabla_incidencias.loc[i, "UsuarioID"].replace("'", '"')
            id_usuarios_list = json.loads(id_usuarios_str)
            id_incidencias = tabla_incidencias.loc[i, "ID"]
            for id_usuarios in id_usuarios_list:
                if (
                    id_usuarios is not None
                    and id_usuarios not in self.incidencia_usuario
                ):
                    self.incidencia_usuario[int(id_incidencias)] = [id_usuarios]
                else:
                    self.incidencia_usuario[int(id_incidencias)].append(id_usuarios)

```

1.6. Proceso de Carga

Esta parte de nuestra pipeline es la que nos permite insertar correctamente los datos en la base de datos de destino, asegurandose de que el formato de los documentos insertados es el correcto y de que no se pierda información en el proceso de transformación de la misma.

En las dos secciones siguientes se describirán la estructura de la base de datos en la que se insertarán los datos finales y el diseño de los esquemas de validación asociados a los agregados a implementar y demás clases.

1.6.1. Estructura final de BBDD

La estructura final planteada para la base de datos se describe a continuación:

1. **AreaRecreativaClima**: Colección que representa el agregado con su mismo nombre. Incluye como raíz la clase *AreaRecreativa* y algunas referencias a los juegos, incidentes de seguridad, encuestas y clima de las areas recreativas.

2. **IncidenteSeguridad**: Colección para almacenar todos los incidentes de seguridad que sucedan en un área recreativa.

3. **EncuestaSatisfaccion**: Colección que almacenará todas las escuestas realizadas por los usuarios a una zona en concreto.

4. **RegistroClima**: Colección que proporciona datos útiles sobre las medidas temporales y precipitaciones entre otras, para el mantenimiento de las áreas.

5. **Juego**: Colección que representa el agregado "Juego" realizado en la BBDD. Éste inlcuirá toda la información relativa a los juegos de un área y referencias hacia mantenimientos realizados a los mismos junto con sus incidencias reportadas por los usuarios.

6. **Mantenimiento**: Colección que almacena todos los documentos que representan los mantenimientos realizados a cada juego. Fuertemente relacionada con el agregado de "Juego".

7. **Incidencia**: Agregado de incidencias a representar en la base de datos. Incluye el almacenamiento de las incidencias de los juegos reportadas por los usuarios y su correspondencia con los usuarios de la plataforma digital.

8. **Usuario**: Colección que almacena a los usuarios de la plataforma digital y beneficiarios de los servicios de las areas recreativas.

1.6.2. Diseño de Esquemas de validación

Dentro de la carpeta "Validadores" se pueden localizar todos los archivos necesarios para crear las colecciones e insertar sus esquemas de validación en la base de datos destino.

Cada archivo representa una colección de las descritas anteriormente en el apartado de la estructura final de la base de datos de destino.

La estructura empleada para la creación de estos archivos es la siguiente:

- Creación de la colección en la base de datos de destino.
- Insertar los índices para los atributos que funcionen como claves primarias.
- Definición de una variable **SCHEME** para el esquema de validación correspondiente.
- Asignación del esquema de validación a la colección creada en el paso 1.

Un ejemplo de estructura de *Esquema de validación* sería:

```
    db.createCollection('<nombre>')

    db.<nombre>.createIndex( {"<PK>":1}, {unique:1} )

    SCHEME= {
        "bsonType": "object",
        "description": ...,
        "required": [ "id","fecha", ... ],
        "properties": {
            "id": {},
            "fecha": {},
            ...
        }
    }

    db.runCommand({ "collMod": "<nombre>", "validator": { $jsonSchema: SCHEME } })
```

1.7. Ejecución

El proceso de Python se ejecuta en el main. Código:

```
import load as ld
import change as ch
import imputation as im
import formater_mongo as mongo_creator


def main():
    #Cargar base de datos sucia
    base = ld.load_db()

    #Limpieza de tablas (ch) + insercion de tablas (im)
    ch.adjust_gps(base)
    ch.adjust_ETRS89(base)
    ch.empty_data(base)
    im.area_new_atribute(base)
```

```
    im.juegos_new_atributes(base)
    im.new_meteo(base)
    ch.no_duplicates(base)
    im.area_meteo(base)
    for tab_name in base: #Funciones que se ejecutan en columnas directamente
        for column_name in base[tab_name]:
            ch.reformatear_fecha(base, tab_name, column_name)
            if type(base[tab_name][column_name][0]) is str:
                ch.capitalize_column(base, tab_name, column_name)
    ch.delete_special(base)
    ch.enum_checker(base)
    ch.check_id(base)
    ch.nif_status(base)
    ch.formato_tlf(base)
    ch.incidencias_status(base)
    im.tiempoResolucion(base)
    im.lastFecha(base)
    im.area_new_atribute(base)
    im.area_meteo(base)
    im.nivelEscalamiento(base)

    # Extracción de objetos
    extractor = mongo_creator.Creator(base)
    extractor.crear_area_clima()
    extractor.crear_juego_incidencias()
    extractor.crear_areas_incidentes()
    extractor.crear_area_encuestas()
    extractor.crear_area_juegos()
    extractor.crear_juegos_tipo()
    extractor.crear_area_clima()
    extractor.crear_incidencia_usuario()

    # Generar datos
    extractor.generar_usuarios()
    extractor.generar_encuestas()
    extractor.generar_incidentes()
    extractor.generar_incidencias()
    extractor.generar_clima()
    extractor.generar_area()
    extractor.generar_mantenimientos()
    extractor.generar_juegos()
    extractor.get_json_data(extractor.incidentesSeguridad, "IncidenteSeguridad")
    extractor.get_json_data(extractor.areas, "AreaRecreativaClima")
    extractor.get_json_data(extractor.encuestas, "EncuestaSatisfaccion")
    extractor.get_json_data(extractor.incidencias, "Incidencia")
    extractor.get_json_data(extractor.juegos, "Juego")
    extractor.get_json_data(extractor.mantenimientos, "Mantenimiento")
    extractor.get_json_data(extractor.registrosClima, "RegistroClima")
    extractor.get_json_data(extractor.usuarios, "Usuario")


if __name__ == "__main__":
    main()


```

Del main se crean los json de datos, que se encuentran en la carpeta *DatasetsNuevos*. Luego, para meter todo en mongoDB se han sacado los siguientes JSON de validación:

1. **Validador AreaRecreativaClima**: JSON de *AreaRecreativaClima*:

```
db.createCollection('AreaRecreativaClima')

db.AreaRecreativaClima.createIndex({ "registrosClima.id": 1 }, { unique: 1, sparse: true })
db.AreaRecreativaClima.createIndex({ "encuestas.id": 1 }, { unique: 1, sparse: true })
db.AreaRecreativaClima.createIndex({ "incidentesSeguridad.id": 1 }, { unique: 1, sparse: true
    })
db.AreaRecreativaClima.createIndex({ "juegos.id": 1 }, { unique: 1, sparse: true })
db.AreaRecreativaClima.createIndex({ "CantidadJuegosTipo.tipo": 1 }, { unique: 1, sparse: true
    })

SCHEME = {
  "bsonType": "object",
  "description": "Agregado de gestión del estado de las áreas recreativas, juegos, incidentes
    de seguridad y datos climáticos.",
  "required": ["id", "coordenadasGPS", "barrio", "distrito", "fecha", "estadoOperativo", "
    capacidadMaxima", "cantidadJuegosTipo", "juegos", "incidentesSeguridad", "registrosClima",
     "encuestas"],
  "properties": {
    "id": {
      "bsonType": "number",
      "description": "Clave Primaria"
    },
    "coordenadasGPS": {
      "bsonType": "array",
      "description": "Array que representa la latitud y la longitud",
      "minItems": 2,
      "maxItems": 2,
      "items": {
        "bsonType": "string"
      }
    },
    "fecha": {
      "bsonType": "string",
      "description": "fecha de instalación del área"
    },
    "barrio": {
      "bsonType": "string",
      "description": "Barrio al que pertenece el area recreativa"
    },
    "distrito": {
      "bsonType": "string",
      "description": "Distrito al que pertenece el area recreativa"
    },
    "estadoOperativo": {
      "bsonType": "string",
      "description": "Estado global del area recreativa",
      "enum": ["OPERATIVO", "CERRADO", "INDISPUESTO"]
    },
    "capacidadMaxima": {
      "bsonType": "number",
      "description": "Numero máximo de juegos que incluye el área",
    },
    "cantidadJuegosTipo": {
      "bsonType": "array",
      "description": "Conteo de cuantos juegos hay de cada tipo en el area actual",
      "minItems": 0,
      "items": {
        "bsonType": "object",
        "description": "objeto resumen de conteo de un tipo de juegos",
        "required": ["tipo", "valor"],
```

```
        "properties": {
          "tipo": {
            "bsonType": "string",
            "description": "tipo de juegos a contar",
            "enum": ["DEPORTIVAS", "INFANTILES", "MAYORES"]
          },
          "valor": {
            "bsonType": "number",
            "minimum": 0
          }
        }
      }
    },
    "juegos": {
      "bsonType": "array",
      "description": "array de referencia a juegos incluidos en el area",
      "minItems": 0,
      "items": {
        "bsonType": "number",
  "description": "juegos en un area",
      }
    },
    "incidentesSeguridad": {
      "bsonType": "array",
      "description": "array de referencias a los incidentes de seguridad",
      "minItems": 0,
      "items": {
        "bsonType": "number",
        "description": "objeto de representacion de incidencias de seguridad"
      }
    },
    "registrosClima": {
      "bsonType": "array",
      "description": "Array de referencias a los registros de clima para esta zona",
      "minItems": 0,
      "items": {
        "bsonType": "number",
      }
    },
    "encuestas": {
      "bsonType": "array",
      "description": "Array de referencias a las encuestas de satisfacción de una zona",
      "minItems": 0,
      "items": {
        "bsonType": "number",
        "description": "objeto referencia de una encuesta de satisfacción"
      }
    },
  },
}

db.runCommand({ "collMod": "AreaRecreativaClima", "validator": { $jsonSchema: SCHEME } })

```

2. **Validador EncuestaSatisfaccion**: JSON de *EncuestaSatisfaccion*:

```
db.createCollection('EncuestaSatisfaccion')

db.EncuestaSatisfaccion.createIndex({ "id": 1 }, { unique: 1 })

```

```
SCHEME = {
  "bsonType": "object",
  "description": "Objeto que representa una encuesta de satisfacción de los usuarios hacia un
    area recreativa",
  "required": ["id", "fechaEncuesta", "puntuacionAccesibilidad", "puntuacionCalidad", "
    comentarios"],
  "properties": {
    "id": {
      "bsonType": "number",
      "description": "Tipo String. Clave Primaria"
    },
    "fechaEncuesta": {
      "bsonType": "string",
      "description": "Fecha de realización de encuesta"
    },
    "puntuacionAccesibilidad": {
      "bsonType": "number",
      "minimum": 1,
      "maximum": 5,
      "description": "puntuaciones de los usuarios respecto a la accesibilidad",
    },
    "puntuacionCalidad": {
      "bsonType": "number",
      "minimum": 1,
      "maximum": 5,
      "description": "puntuaciones de los usuarios respecto a la calidad",
    },
    "comentarios": {
      "bsonType": "string",
      "description": "Lista de comentarios en formato string",
      "enum": ["ACEPTABLE", "DEFICIENTE", "EXCELENTE", "MUY BUENO", "REGULAR"]
    }
  },
}

db.runCommand({ "collMod": "EncuestaSatisfaccion", "validator": { $jsonSchema: SCHEME } })

```

3. **Validador Incidencia**: JSON para *Incidencia*:

```
db.createCollection('Incidencia')

db.Incidencia.createIndex( {"id":1}, {unique:1} )

SCHEME={
   "bsonType": "object",
   "description": "Agregado de gestión del estado del juego, historial de mantenimiento y
    seguimiento de incidencias.",
   "required": ["id","tipo","fechaReporte","estado","tiempoResolucion","nivelEscalamiento","
    usuarios"],
   "properties": {
            "id": {
                "bsonType": "number",
                "description": "id de la incidencia"
            },
            "tipo": {
                "bsonType": "string",
                "description": "tipo de incidencia al juego",
                "enum": ["DESGASTE","MAL FUNCIONAMIENTO","ROTURA","VANDALISMO"]
            },
```

```
            "fechaReporte": {
                "bsonType": "string",
                "description": "fecha de reporte de la incidencia"
            },
            "estado": {
                "bsonType": "string",
                "description": "estado de la inicidencia",
                "enum": ["ABIERTA","CERRADA"]
            },
            "tiempoResolucion": {
               "bsonType": "number",
               "description": "tiempo en el que se resolverá la incidencia",
            },
            "nivelEscalamiento": {
               "bsonType": "number",
               "minimum": 1,
               "maximum": 10,
               "description": "Nivel de urgencia"
            },
            "usuarios": {
               "bsonType": "array",
               "description": "array de referencias a usuarios vinculados con esta incidencia"
    ,
               "minItems": 1,
               "items": {
                    "bsonType": "object",
                    "description": "objeto embedido del usuario",
                    "required": ["NIF","nombre","email","telefono"],
                    "properties": {
                        "NIF": {
                            "bsonType": "string",
                            "description": "Identificador de tipo string",
                        },
                        "nombre": {
                            "bsonType": "string",
                            "description": "nombre del usuario"
                        },
                        "email": {
                            "bsonType": "string",
                            "description": "email del usuario"
                        },
                        "telefono": {
                            "bsonType": "string",
                            "description": "telefono de los usuarios"
                        }
                    }
               }
            },
    },
}

db.runCommand({ "collMod": "Incidencia", "validator": { $jsonSchema: SCHEME } })

```

4. **Validacion IncidenteSeguridad**: JSON para *IncidenteSeguridad*:

```
db.createCollection('IncidenteSeguridad')

db.IncidenteSeguridad.createIndex({ "id": 1 }, { unique: 1 })

```

```
SCHEME = {
  "bsonType": "object",
  "description": "Objeto que representa un incidente de seguridad",
  "required": ["id", "fechaDeReporte", "tipoIncidente", "gravedad"],
  "properties": {
    "id": {
      "bsonType": "number",
      "description": "id del incidente de seguridad"
    },
    "tipoIncidente": {
      "bsonType": "string",
      "description": "tipo de incidencia de seguridad",
      "enum": ["ROBO", "CAIDA", "ACCIDENTE", "VANDALISMO", "DAÑO ESTRUCTURAL"]
    },
    "gravedad": {
      "bsonType": "string",
      "description": "gravedad del incidente",
      "enum": ["BAJA", "MEDIA", "ALTA", "CRITICA"]
    },
    "fechaDeReporte": {
      "bsonType": "string",
      "description": "fecha de reporte de la incidencia de seguridad"
    }
  },
}

db.runCommand({ "collMod": "IncidenteSeguridad", "validator": { $jsonSchema: SCHEME } })

```

5. **Validador Juego**: JSON para *Juego*:

```
db.createCollection('Juego')
db.Juego.createIndex({ "id": 1 }, { unique: 1 })

SCHEME = {
  "bsonType": "object",
  "description": "Agregado de gestión del estado del juego, historial de mantenimiento y
    seguimiento de incidencias.",
  "required": ["id", "modelo", "estadoOperativo", "accesibilidad", "fechaInstalacion", "tipo",
     "desgasteAcumulado", "indicadorExposicion", "ultimaFechaMantenimiento", "mantenimientos",
     "incidencias"],
  "properties": {
    "id": {
      "bsonType": "string",
      "description": "Tipo String. Clave Primaria"
    },
    "modelo": {
      "bsonType": "string",
      "description": "Modelo de fabricacion del juego"
    },
    "estadoOperativo": {
      "bsonType": "string",
      "description": "Estado del juego",
      "enum": ["OPERATIVO", "INDISPUESTO"]
    },
    "accesibilidad": {
      "bsonType": "bool",
      "description": "Si el juego es accesible por la gente o no"
    },
    "fechaInstalacion": {
```

```
      "bsonType": "string",
      "description": "Fecha de instalacion del juego"
    },
    "tipo": {
      "bsonType": "string",
      "description": "Tipo de juego",
      "enum": ["DEPORTIVAS", "INFANTILES", "MAYORES"]
    },
    "desgasteAcumulado": {
      "bsonType": "number",
      "description": "porcentaje de desgaste que tiene un juego",
      "minimum": 0,
      "maximum": 100,
    },
    "indicadorExposicion": {
      "bsonType": "string",
      "description": "Indicador de exposición a temperaturas aridas",
      "enum": ["INTACTO", "BAJO", "MEDIO", "ALTO"]
    },
    "ultimaFechaMantenimiento": {
      "bsonType": "string",
      "description": "ultima fecha de mantenimiento del juego"
    },
    "mantenimientos": {
      "bsonType": "array",
      "description": "array de referencias a registros de mantenimiento del juego",
      "minItems": 0,
      "items": {
        "bsonType": "string",
        "description": "objeto de referencia de mantenimiento realizado"
      }
    },
    "incidencias": {
      "bsonType": "array",
      "description": "array de referencias a las incidencias reportadas para este juego",
      "minItems": 0,
      "items": {
        "bsonType": "number",
        "description": "objeto de referencia a la incidencia del juego"
      }
    },
  },
}

db.runCommand({ "collMod": "Juego", "validator": { $jsonSchema: SCHEME } })

```

6. **Validador Mantenimiento**: JSON para *Mantenimiento*:

```
db.createCollection('Mantenimiento')

db.Mantenimiento.createIndex({ "id": 1 }, { unique: 1 })

SCHEME = {
  "bsonType": "object",
  "description": "Objeto que representa un incidente de seguridad",
  "required": ["id", "fechaIntervencion", "tipoIntervencion", "estadoPrevio", "estadoPosterior
    "],
  "properties": {
    "id": {
```

```
      "bsonType": "string",
      "description": "id del mantenimiento a realizar a un juego"
    },
    "tipoIntervencion": {
      "bsonType": "string",
      "description": "tipo de intervencion",
      "enum": ["CORRECTIVO", "EMERGENCIA", "PREVENTIVO"]
    },
    "estadoPrevio": {
      "bsonType": "string",
      "description": "estado previo a la intervencion",
    },
    "estadoPosterior": {
      "bsonType": "string",
      "description": "estado posterior a la intervencion",
    },
    "fechaIntervencion": {
      "bsonType": "string",
      "description": "fecha de intervencion de mantenimiento"
    },
    "incidencias": {
      "bsonType": "array",
      "description": "incidencias que referencia un mantenimiento",
      "minItems": 0,
      "items": {
        "bsonType": "number",
        "description": "objeto de referencia de incidencias realizadas"
      }
    }
  },
}

db.runCommand({ "collMod": "Mantenimiento", "validator": { $jsonSchema: SCHEME } })

```

7. **Validador RegistroClima**: JSON para *RegistroClima*:

```
db.createCollection('RegistroClima')

db.RegistroClima.createIndex({ "id": 1 }, { unique: 1 })

SCHEME = {
  "bsonType": "object",
  "description": "Objeto que representa los datos del clima en la zona de una area recreativa"
    ,
  "required": ["id", "fecha", "temperatura", "precipitacion", "vientosFuertes"],
  "properties": {
    "id": {
      "bsonType": "int",
      "description": "id del registro del clima"
    },
    "temperatura": {
      "bsonType": "number",
      "description": "temperatura tomada en la zona del area recreativa",
    },
    "precipitacion": {
      "bsonType": "number",
      "description": "precipitacion tomada en la zona del area recreativa",
    },
    "fecha": {
```

```
      "bsonType": "string",
      "description": "fecha de la toma de muestras del clima"
    },
    "vientosFuertes": {
      "bsonType": "bool",
      "description": "si hacen vientos fuertes o no"
    }
  },
}

db.runCommand({ "collMod": "RegistroClima", "validator": { $jsonSchema: SCHEME } })

```

8. **Validacion Usuario**: JSON de *Usuario*:

```
db.createCollection('Usuario')

db.Usuario.createIndex( {"NIF":1}, {unique:1} )

SCHEME={
   "bsonType": "object",
   "description": "Objeto que representa un usuario de la zona recreativa",
   "required": ["NIF","nombre","email","telefono"],
   "properties": {
        "NIF": {
            "bsonType": "string",
            "description": "Identificador del usuario"
        },
        "nombre": {
            "bsonType": "string",
            "description": "nombre de usuario de la zona",
        },
        "email": {
            "bsonType": "string",
            "description": "email del usuario",
        },
        "telefono": {
            "bsonType": "string",
            "description": "telefono del usuario",
        }
    },
}
db.runCommand({ "collMod": "Usuario", "validator": { $jsonSchema: SCHEME } })

```

Por último, ejecutamos tanto los JSONs de datos como el pipeline.txt en mongoDB. Así tendríamos todos los datos y sus relaciones en la base de datos. En el pipeline además contiene la representación *EstadoGlobalÁrea*, que inserta en *AreaRecreativaClima* el estado global teniendo en cuenta las incidencia y manetnimientos de los juegos de dicha área. Esta implementación se hizo con ayuda de ChatGPT Código del pipeline.txt:

```

use ej1

// agregado Areas Recreativas
db.AreaRecreativaClima.aggregate([
    // Encontrar los Incidentes
    {
        $lookup:
        {
```

```
            from: "IncidenteSeguridad",
            localField: "incidentesSeguridad",
            foreignField: "id",
            as: "RefIncidentes"
        }
    },
    // Encontrar los juegos
    {
        $lookup:
        {
            from: "IncidenteSeguridad",
            localField: "Juego",
            foreignField: "id",
            as: "RefJuegos"
        }
    },
    // project para crear todos los atributos
    {
        $project:
        {
            // area recreativa
            "id": 1,
            "coordenadasGPS": 1,
            "barrio": 1,
            "distrito": 1,
            "fecha": 1,
            "estadoGlobalArea": 1,
            "capacidadMaxima": 1,
            "cantidadJuegosTipo": 1,
            "juegos": 1,
            "incidentesSeguridad": 1,
            "registrosClima": 1,
            "encuestas": 1,
            // Referencia Juegos
            RefJuego: "$juegos",
            // Referencia con resumen IncidenteSeguridad
            "RefIncidentes.fechaDeReporte": 1,
            "RefIncidentes.tipoIncidente": 1,
            "RefIncidentes.gravedad": 1,
            // Referencia Clima
            RefClima: "$registrosClima",
            // Encuesta Satisfaccion
            RefEncuestas: "$encuestas"
        }
    },
    // estado global area
    {
        $addFields: {
            estadoGlobalArea: {
                $cond: {
                    if: {
                        $anyElementTrue: {
                            $map: {
                                input: { $ifNull: ["$RefIncidentes.gravedad", []] },  // Asegura que RefIncidentes.gravedad sea un array
                                as: "gravedad",
                                in: { $eq: ["$$gravedad", "CRITICA"] }  // Verifica si alguna gravedad es "CRITICA"
                            }
                        }
```

```
                    },
                    then: "PELIGROSO",  // Si hay "CRITICA", se asigna "PELIGRO"
                    else: {
                        $cond: {
                            if: {
                                $anyElementTrue: {
                                    $map: {
                                        input: { $ifNull: ["$RefIncidentes.gravedad", []] },
    // Asegura que RefIncidentes.gravedad sea un array
                                        as: "gravedad",
                                        in: { $eq: ["$$gravedad", "ALTA"] }  // Verifica si
    alguna gravedad es "ALTA"
                                    }
                                }
                            },
                            then: "PRECAUCION",  // Si hay "ALTA", se asigna "MEDIO"
                            else: "NORMAL"  // Si no hay ni "CRITICA" ni "ALTA", se asigna "
    BAJO"
                        }
                    }
                }
            }
        }
    },
    // Llevarlo a una colección nueva
    {
        $out: {db:"ej1", coll: "AgregadoAreaRecreativaClima"}
    }
])

// Agregado Incidencia
db.Incidencia.aggregate([
    // Crear atributos usando project
    {
  $project:
        {
            "id": 1,
            "fechaReporte": 1,
      "tipo": 1,
      "estado": 1,
      "tiempoResolucion": 1,
      "nivelEscalamiento": 1,
      RefUsuarios: "$usuarios"
        }
    },
    // Llevarlo a una colección nueva
    {
        $out: {db:"ej1", coll: "AgregadoIncidencia"}
    }

])

// Agregado Juego
db.Juego.aggregate([
    // Etapa de $lookup para unir con la colección 'id'
    {
        $lookup: {
            from: "Incidencia",
            localField: "incidencias",
            foreignField: "id",
```

```
            as: "Inc"
        }
    },
    // Etapa de $project para seleccionar y renombrar los campos deseados
    {
        $project: {
            "id": 1,
            "modelo": 1,
            "estadoOperativo": 1,
            "accesibilidad": 1,
            "fechaInstalacion": 1,
            "tipo": 1,
            "desgasteAcumulado": 1,
            "indicadorExposicion": 1,
            "ultimaFechaMantenimiento": 1,
            "mantenimientos": 1,
            "incidencias": 1,
            //RefMantenimiento
            RefMant: "$Mantenimiento",
            //RefIncidencia - selecciona campos de 'Inc' obtenido en $lookup
            "Inc.fechaReporte": 1,
            "Inc.tipo": 1,
            "Inc.estado": 1
        }
    },
    // Etapa de $out para exportar los resultados a otra colección
    {
        $out: { db: "ej1", coll: "AgregadoJuego" }
    }
])

```

2. Datos importantes a tener en cuenta

Primero, cabe destacar que vamos a entregar la memoria y el .txt junto al código de Python y los JSONs. Así, facilitamos el uso del código sin tener que ir en la memoria apartado en apartado copiándolo y ejecutándolo. Cabe destacar que es necesario tener todas las librerías de Python usadas (pandas, datetime, json, etc.) para ejecutarlo sin problemas, además de que puede llegar a tardar entre 5-10 min en su ejecución.

Además, no hemos representado los JSONs con los datos resultantes de main() debido a su gran extensión de líneas. Los hemos dejado en la carpeta *DatasetsNuevos* para su uso en mongoDB.

Por último, hemos indicado el uso de ChatGPT en todas las funciones en las que nos ha ayudado y cómo ha realizado dicha ayuda.