Práctica 1

MONGODB

uc3m | Universidad Carlos III de Madrid

Arquitectura de datos

10 de Noviembre de 2024

Ignacio Fernández Cañedo 100471955

Belén Gómez Arnaldo 100472037

Ana María Ortega Mateo 100472023

ÍNDICE

1. Introducción

En esta práctica realizaremos una migración de datos partiendo del dataset del sistema antiguo. Se realizará un proceso de importación ETL para cargar los datos en MongoDB de acuerdo con el diseño de agregados propuesto. Primero haremos una limpieza de los datos con python para después cargar los datos en una base de datos sin validaciones. Después migraremos los datos a la base final con validaciones, comprobando que todos los datos tienen la estructura y tipo correctos.

2. Limpieza de datos con Python

Antes de subir los datos a la primera base de datos de Mongo los limpiamos utilizando python. A continuación vamos a explicar los cambios que hemos hecho sobre cada archivo y los problemas o anomalías que hemos encontrado. En todos los datasets hemos eliminado las tildes para evitar problemas con estos caracteres en Mongo. En algunos casos también hemos utilizado funciones adicionales para contar el número de valores nulos por cada columna, identificar si había valores repetidos en columnas o identificar los posibles valores de una columna de tipo enumerativo.

2.1. Usuario

En este dataset hemos eliminado la última columna que se llamaba “Email” porque estaba vacía y ya había otra columna con los emails de los usuarios. En la columna que sí contiene los emails de los usuarios hay 20 valores nulos. Hemos considerado que puede haber usuarios que no tengan un email registrado porque sí que tienen número de teléfono, es decir, sí que hay información de contacto del usuario. Se puede actualizar a los usuarios sobre el estado de la incidencia que han reportado a partir del número de teléfono o del email, por lo que es suficiente con que exista el número de teléfono. Por lo tanto, el campo de número de teléfono es obligatorio.

En cuanto a la columna de “Teléfono”, los números de teléfono no estaban en un formato consistente por lo que hemos eliminado los espacios que había en estos datos.

Por último, el NIF de los usuarios tiene que ser único. Hemos identificado que había algunos valores que se repetían dos veces. Analizando estos valores hemos visto que los valores repetidos se referían al mismo usuario pero con distinto email o número de teléfono. Para mantener la integridad de los datos hemos eliminado una de estas filas para que no haya valores repetidos.

Mantenimiento

En este dataset hay una columna que contiene la fecha de cada intervención. En los datos originales estas fechas están en diferentes formatos que no son válidos en mongo. Por ello, hemos modificado todas las fechas para que tengan el formato válido (YYYY-MM-DD). También hemos eliminado la columna “Comentarios” porque no está en el modelo ni se utiliza en ningún caso de uso y la columna “Tipo” porque ya existe una columna igual llamada “TIPO_INTERVENCION”.

En este dataset no hay valores nulos y todos los ID 's son únicos. Sin embargo, hemos modificado el formato del ID para que coincida con el ID que hay en el dataset de incidencia. Hemos hecho esto porque necesitamos datos del dataset de mantenimiento para completar el de incidencia, que explicaremos a continuación. Por ello, el formato de los ID’s de mantenimiento es *MNT-NNNNN*. Esto nos permitirá relacionar los datos de una forma más sencilla y eficiente.

2.2. Incidencia

En este dataset, hemos modificado las fechas con formatos no válidos al formato admitido por mongo (YYYY-MM-DD) y se ha realizado una limpieza de las tildes que el dataset contiene, sustituyendo las vocales acentuadas por su vocal sin acentuar.

Hemos comprobado que este dataset no contiene ningún valor nulo y hemos analizado si había “ID” repetidos. Tras hacer esta última verificación, había varias instancias que tenían los mismos ID's y todo el resto de la información de la incidencia era igual. Se han eliminado estos duplicados para mantener la coherencia de los datos y tener ID’s únicos.

A continuación calculamos el atributo de tiempo de resolución de una incidencia. Hemos creado este campo en el nuevo dataset con ayuda de la fecha de mantenimiento. Cada incidencia tiene uno o varios ids de mantenimiento asociados, por lo que buscamos en el dataset de mantenimiento ese o esos ids de y cogemos la fecha más actual. Esta fecha de mantenimiento se utiliza para calcular la diferencia entre fecha de mantenimiento y la fecha de incidencia, obteniendo así el tiempo de resolución. Como no se especifica en el enunciado hemos decidido dar este tiempo en días.

Hemos identificado dos anomalías en estos datos:

- Hay incidencias que están abiertas pero tienen un mantenimiento. Esto no es un dato incorrecto, una incidencia puede haber tenido un mantenimiento pero que no se haya conseguido cerrar la incidencia con un solo mantenimiento. Por esto también hay incidencias que tienen más de un mantenimiento. Sin

embargo, en estos casos no podemos calcular un tiempo de resolución porque la incidencia no está resuelta.
- Hemos identificado casos en los que la fecha de la incidencia es posterior a la fecha del mantenimiento. Lógicamente estos valores no son correctos.

Para estos dos casos, habíamos pensado en establecer un tiempo de resolución por defecto. Sin embargo, el tiempo de resolución se utiliza en el caso de uso B para producir indicadores de calidad para cada subcontrata de mantenimiento. Si pusiéramos todos estos valores anómalos a 0 o con un valor por defecto se acabaría falseando este indicador de calidad. Por ello, hemos decidido marcar estos valores para que no se utilicen al calcular el indicador porque no son valores reales. En todos estos casos hemos puesto el tiempo de resolución como -1. De esta forma, cuando se vaya a calcular el indicador de calidad solo se tienen que tener en cuenta los valores positivos y así no se falsea el resultado.

Otro atributo que tiene la clase incidencia es el nivel de escalamiento. Este atributo se utiliza para categorizar la prioridad de una incidencia. Es un valor enumerativo que puede tener los valores de “POCO URGENTE”, “URGENTE” o “MUY URGENTE”. Para determinar la urgencia de una incidencia nos hemos basado en el número de usuario que han reportado la incidencia. Si muchos usuarios han reportado la incidencia implica que es más urgente. Por ello, si la incidencia solo ha sido reportada por un usuario se categoriza como “POCO URGENTE”, si ha sido reportada por dos usuarios se categoriza como “URGENTE” y si ha sido reportada por más de dos usuarios se categoriza como “MUY URGENTE”.

Por último, hemos añadido el atributo de JuegoID porque luego nos va a ayudar a hacer los agregados en Mongo ya que el agregado de Juego tiene una referencia a las incidencias de este juego. Para poder obtener el id del juego al que pertenece la incidencia hemos accedido a los mantenimientos asociados a la incidencia. Todas las incidencias tienen mantenimientos y los mantenimientos tienen el id del juego al que pertenecen, por lo que lo hemos obtenido a partir de ese dataset. Además, con esto también hemos comprobado que todos los mantenimientos de una incidencia están asociados al mismo juego, que es lo que debería pasar.

2.3. Incidente seguridad

Este dataset ha necesitado pocas modificaciones. En primer lugar, hemos modificado todas las fechas con formato incorrecto del campo “FECHA_REPORTE” al formato de fecha que mongo reconoce: YYYY-MM-DD. A continuación, los campos “TIPO_INCIDENTE” y “GRAVEDAD” contienen tildes por lo que nos aseguramos de quitarlas y poner la vocal correspondiente sin tilde. Finalmente, comprobamos que no

hay ningún campo con valores nulos y detectamos que hay instancias del dataset repetidas, con el mismo ID e información, por lo que decidimos eliminarlas.

2.4. Encuesta satisfacción

Este dataset ha necesitado pocas modificaciones. Se han corregido los formatos de fechas no válidos del campo “FECHA”, cambiándose al formato YYYY-MM-DD. También, se han modificado todas las tildes que el dataset puede llegar a contener por su vocal correspondiente. Esta dataset no contiene valores nulos o ID’s repetidos.

2.5. Meteorología

Para poder limpiar este dataset tuvimos que leer la interpretación de datos meteorológicos del ayuntamiento de Madrid para poder entender la distribución y formato. El dataset estaba compuesto por muchas columnas con el formato “DXX” y “VXX”, donde D representaba el día del mes (por ejemplo, D01 para el 1 de cada mes) y V indicaba si el dato es válido para ese día específico. Como también había columnas de “AÑO” y “MES”, la primera modificación que se realizó fue eliminar todas estas columnas para unificarlas en una sola que sería “FECHA” con formato YYYY-MM-DD. Los datos se organizaron de manera que por cada sitio o zona, cada día se convirtiera en una nueva instancia en el dataset, con su correspondiente fecha y validación. La distinción entre los datos meteorológicos de cada sitio se podía obtener gracias al “PUNTO_MUESTREO”, donde los primeros 8 caracteres hacen referencia a un lugar en específico. Por lo tanto, cada fila del dataset limpio tiene el punto de muestreo y la fecha, es decir, no hay filas con el mismo punto de muestreo y fecha porque serían datos repetidos.

Para limpiar más a fondo el conjunto de datos decidimos eliminar todas aquellas instancias con valores no válidos (‘N’) y también eliminar todas aquellas magnitudes que no hacen referencia a temperatura, precipitaciones o viento, eliminando las instancias con la columna MAGNITUDES distinta a 83, 89 o 81.

A continuación, creamos las columnas de TEMPERATURA, PRECIPITACION y VIENTO, donde en función de la fecha, los ocho primeros dígitos del punto de muestreo y la magnitud se iban rellenando estos campos. El valor de magnitud 81 hace referencia al viento, el 83 a temperatura y el 89 a precipitaciones. Por ello, optimizamos el número de instancias agrupando aquellas que tenían misma fecha y punto de muestreo(8 primeros dígitos), asignando los valores correspondientes a las nuevas columnas según su magnitud. Por ejemplo, un registro con la magnitud 81 se coloca en la columna de VIENTO, mientras que uno con la magnitud 83 se añade a la columna de TEMPERATURA. Estos nuevos campos creados pueden tener valores nulos, ya que

muchas de las instancias creadas no cuentan con información para todas las magnitudes.

La forma de relacionar los datos del clima con las áreas es a través del código postal. En el dataset de “estaciones_meteo_CodigoPostal.csv” tenemos el código de cada estación, que en nuestro nuevo dataset es el punto de muestreo, y el código postal de las áreas a las que afectan los datos climáticos de esa estación. Con estos datos hemos creado una nueva columna que incluye los códigos postales asociados a cada estación. De esta forma podremos hacer las referencias entre Área y Registro Climático en el agregado de mongo.

Por último hemos añadido un ID único a cada registro.

2.6. Juego

Lo primero que hicimos en este archivo fue eliminar las tildes. Después creamos las columnas de latitud y longitud a partir de las coordenadas geográficas.

Algunos de los atributos de los juegos están relacionados con el área a la que pertencen. Además, en el agregado de Área se hace un referencia al juego, por lo que queríamos obtener el ID del área correspondiente a cada juego. En un principio utilizamos los datos de las columnas NDP y COD_INTERNO ya que había casos en los que coincidían. Sin embargo, no obtuvimos muy buenos resultados porque había juegos o áreas que no tenían estos valores.

Finalmente decidimos relacionar los juegos con sus áreas en función de las coordenadas geográficas. Para esto hemos supuesto que las coordenadas de las áreas hacen referencia al centro del área por lo que asignamos cada juego al área más cercana. Teniendo en cuenta que hay un gran número de áreas y juegos esto podría tardar mucho y ser poco eficiente. Para evitar esto hemos usado la librería de python *scipy.spatial*, en concreto hemos usado la estructura cKDTree. Esta biblioteca crea una estructura de árbol a partir de las coordenadas de las áreas. Este árbol permite encontrar de forma eficiente el área más cercana a una coordenada dada, es decir, encontrar el área más cercana a cada juego en función de su latitud y longitud. De esta forma hemos creado la columna de AreaRecreativaID en el dataset de Juegos.

A continuación encontramos que había juegos que no tenían una fecha de instalación o tenían “fecha_incorrecta”. Para que los datos tuvieran sentido decidimos suponer que era la misma que la fecha de instalación del área. Esto es una solución mejor que poner una fecha por defecto porque nos aseguramos que la fecha no es anterior a la fecha de instalación del área, que no tendría sentido. También modificamos el formato de las fechas para que tuvieran el formato válido de mongo.

El valor de modelo hay casos en los que está vacío. En esos casos se utiliza el valor de la columna “DES_CLASIFICAION”. Esta columna tiene referencias al modelo del juego pero es menos completa, por lo que preferimos quedarnos con la columna de modelo excepto en los casos en los que esté vacía.

También hemos creado la columna de “ultimaFechaMantenimiento”. Esta columna se obtiene a través del dataset de mantenimiento, escogiendo la fecha más antigua para cada juego, ya que el mantenimiento tiene el ID del juego correspondiente. Sin embargo, esta columna puede tener valores vacíos porque puede haber juegos que todavía no hayan tenido ningún mantenimiento. Esto no es un error en los datos.

Hemos creado la columna de indicador de exposición de forma aleatoria. Esta columna puede tener los valores de “alto”, “medio” o “bajo”. Con este indicador y con el número de mantenimientos que ha tenido el juego se ha obtenido el valor de “desgasteAcumulado” como:

*(tiempo_de_uso *indicadorExposicion*100) - (numero_de_mantenimientos * 100)*

Para esta fórmula se considera que el tiempo de uso es un valor aleatorio entre 1 y 15.

Para calcular la accesibilidad del juego usamos los datos del dataset de encuestas. Las encuestas están relacionadas con las áreas, no con los juegos. Como hemos creado una columna con el id del área en la que está cada juego, con este id podemos acceder a la puntuación de accesibilidad del área. No hay una puntuación para cada juego, por lo que usaremos la media de la puntuación del área al que pertenecen. De esta forma, si la media es mayor que 3 diremos que el juego es accesible (*True*), en caso contrario diremos que no es accesible (*False*).

En este dataset hay muchas columnas que no se utilizan ni en el modelo ni en los casos de uso. Tampoco son necesarias para referenciar a otras clases o para la limpieza de datos, por lo que hemos decidido eliminarlas. Las columnas son: 'NDP', 'CODIGO_INTERNO', 'DESC_CLASIFICACION', 'COORD_GIS_X', 'COORD_GIS_Y', 'SISTEMA_COORD', 'CONTRATO_COD', 'BARRIO', 'COD_BARRIO', 'DISTRITO', 'COD_DISTRITO', 'LATITUD', 'LONGITUD', 'TIPO_VIA', 'NOM_VIA', 'NUM_VIA', 'COD_POSTAL','DIRECCION_AUX'.

Por último, detectamos que en el dataset había juego con ID’s e información repetida por lo que eliminamos los duplicados.

2.7. Área

En este dataset hemos realizado varias modificaciones para estandarizar y limpiar los datos. Se eliminó cualquier tilde en los datos para evitar problemas de compatibilidad

con MongoDB. Las fechas, que estaban en diferentes formatos, se unificaron al formato YYYY-MM-DD para estandarización. Además, el formato de las coordenadas se cambió de las columnas LATITUD y LONGITUD a una única columna coordenadasGPS en el formato [latitud, longitud]. Los textos en las columnas DISTRITO y BARRIO se convirtieron todos a mayúsculas, ya que estaban escritos con mayúsculas y minúsculas aleatoriamente. El CODIGO_POSTAL fue convertido a una cadena ya que es como lo usaremos en MongoDB.

Para representar la capacidad máxima, se creó una nueva columna calculada al sumar un valor aleatorio (entre 1 y 10) al valor de la columna TOTAL_ELEM.

Respecto al manejo de los valores nulos, se observaron varios valores nulos en las columnas COD_DISTRITO y DISTRITO. Para llenar estos valores, creamos dos diccionarios: uno donde el código del distrito es la clave y el nombre del distrito el valor, y otro inverso. Recorrimos el dataset para rellenar estos diccionarios. De esta forma, cuando se encuentre un distrito sin código se puede acceder al diccionario y obtener el código correspondiente a ese distrito. Lo mismo sucede cuando hay un nombre de distrito sin el código.

Otra anomalía que detectamos es que había áreas que tenían un valor de 0 en el código postal. Para arreglar esto utilizamos el mismo enfoque, creamos un diccionario en el que la clave fuera el barrio y el valor el código postal asociado. Cuando se encuentre un valor del código postal de 0 o nulo se accede al diccionario y se obtiene el código asociado al barrio de ese área.

En cuanto a la fecha de instalación, hay muchas áreas que tienen ‘fecha incorrecta’ o no tienen fecha. Para poder obtener este valor de forma adecuada decidimos buscar las fechas de los juegos instalados en ese área y usar la fecha más antigua. Como en el dataset de juegos incluímos un ID del área al que pertenecen fue fácil obtener esta fecha. Sin embargo, hemos detectado que hay áreas que no tienen juegos instalados en ellas. No hemos considerado esto como un error de los datos, puede ser que en el área todavía no se hayan instalado juegos. Aunque no se diga en el enunciado, también hemos supuesto que se pueden desinstalar juegos si están muy desgastados o son muy antiguos, por lo que no hemos considerado que un área sin juego sea un error.

Después de obtener la fecha de instalación con la fecha de los juegos seguía habiendo 5 áreas sin fecha. Por lo que hemos explicado de poder desinstalar juegos, hemos supuesto que, aunque en ese momento el área no tenga juegos, si los ha tenido en el pasado puede tener encuestas o incidentes, por lo que hemos obtenido la fecha de estos dataset. Sin embargo, aún con esto hay un área que no tiene fecha de instalación. Este caso no se podrá incluir en la migración porque no cumplirá la validación y habrá que guardarlo en un área de trabajo para su procesamiento

manual. Esto no crearía inconsistencias en los datos porque es un área que no tiene ni juegos, ni encuestas ni incidentes por lo que no tiene ningún dato o documento asociado.

Otro atributo que tiene que tener el área es la cantidad de juegos por tipo. Para obtener esto hemos creado un diccionario con los tres tipos de juegos disponibles: deportivas, mayores o infantiles. Hemos rellenado este diccionario accediendo a los valores del dataset de juegos con el ID del área.

La última columna que hemos añadido a este dataset es el estado global del área. Este valor se calcula en función del número de juegos en mantenimiento, los incidentes de seguridad y la satisfacción de los usuarios de la siguientes forma:

*puntuacion_calidad *10 - (nº juegos en mantenimiento + nº incidentes de seguridad)*

El valor de estado global es un valor enumerativo que puede ser “INSUFICIENTE” si el resultado de la fórmula es menos que 0, “SUFICIENTE” si el resultado está entre 0 y 25 o “BUENO” si el resultado es mayor que 25.

Por último eliminamos columnas que no están en el modelo y no se utilizan en los casos de uso. Las columnas que eliminamos son las siguientes: 'COD_DISTRITO', 'COORD_GIS_X', 'COORD_GIS_Y', 'SISTEMA_COORD', 'CONTRATO_COD', 'tipo', 'COD_BARRIO', 'DESC_CLASIFICACION', 'TOTAL_ELEM', 'TIPO_VIA', 'NOM_VIA', 'NUM_VIA', 'DIRECCION_AUX', 'NDP', 'CODIGO_INTERNO'.

3. MongoDB

3.1. Cargar los datos

Antes de cualquier implementación de Mongo, cargamos los datasets en nuestra base de datos inicial (bd_sin_validacion). Para ello, utilizamos el siguiente comando:

```
Unset
mongoimport --uri="mongodb://localhost:27017" \
            --db bd_sin_validacion \
            --collection nombre_coleccion \
            --type csv \
            --file "/home/lab/csv_coleccion.csv" \
           --headerline
```

Sustituimos "nombre_coleccion", por el nombre que queremos asignar a la colección que queremos cargar y "csv_coleccion.csv" por el nombre del csv que almacena los datos de dicha colección.

3.2. Validaciones

Para poder migrar los datos de una base de datos a otra, los datos que se almacenan en la base de datos limpia tienen que cumplir con las validaciones que se han creado. Por ello, cada colección que se añade en esta base de datos tiene un validador asociado que se encargará de comprobar que todos los datos que se migran cumplen con las condiciones establecidas. Para las validaciones, optamos por empezar por colecciones con menor número de atributos y más simples y luego progresivamente avanzamos hacia colecciones más complejas como por ejemplo juego o área.

3.2.1. Encuesta

Para realizar el validador de encuesta, se crea una colección con el nombre de "Encuestas_con_validacion" con reglas de validación, que aseguran que los datos migrados a esta colección tengan todos los campos necesarios y cumplan con el tipo de cada campo. El validador implementado es el siguiente:

```
Unset
db.createCollection("Encuestas_con_validacion", {
    validator: {
        $jsonSchema: {
            bsonType: "object",
                                required:  ["ID",  "PUNTUACION_ACCESIBILIDAD",
"PUNTUACION_CALIDAD", "FECHA", "AreaRecreativaID"],
            properties: {
                 ID: { bsonType: "string", description: "Debe ser un ID único
en formato string" },
                   PUNTUACION_ACCESIBILIDAD: { bsonType: "int", minimum: 0,
maximum: 5, description: "Puntuación de 0 a 5" },
                  PUNTUACION_CALIDAD: { bsonType: "int", minimum: 0, maximum:
5, description: "Puntuación de 0 a 5" },
                 COMENTARIOS: { bsonType: "string", description: "Comentarios
de texto opcionales" },
                      FECHA: { bsonType: "date", description: "Fecha de la
encuesta en formato de fecha" },
                 AreaRecreativaID: { bsonType: "string", description: "ID del
area"}
            }
        }
    }
```

```
})
```

En *required* se especifican los campos que son necesarios en cada documento, en este caso son: el identificador, la puntuación de accesibilidad, la puntuación de calidad, la fecha en la que se realizó la encuesta y el identificador del área.

A continuación, se define el tipo de dato de cada campo usando bsonType. Los campos ID, COMENTARIOS y AreaRecreativaID deben de ser de tipo string. Los campos PUNTUACION_CALIDAD y PUNTUACION_ACCESIBILIDAD deben ser enteros y tener valores entre 0 y 5 (minimum y maximum). Por último, el campo FECHA debe ser de tipo date. Si alguno de estos datos no cumple con el tipo establecido o rango, se genera un error que muestra la descripción que nosotros hemos definido.

Los campos definidos como necesarios se han establecido en base al diagrama UML del enunciado. Además, para poder referenciar las encuestas de cada área se ha añadido el campo de AreaRecreativaID.

Para establecer un identificador único a cada encuesta del dataset se ha creado un índice en el campo ID, garantizando de esta manera que todos los valores del atributo ID tienen sean únicos. El índice se ha creado de la siguiente forma y se le ha atribuido un nombre en especifico.

```
Unset
db.Encuestas_con_validacion.createIndex({ ID: 1 }, { unique: true,
name:"indiceEncuesta" })
```

3.2.2. Incidentes de Seguridad

El validador de incidentes de seguridad, crea una colección con el nombre de “IncidentesSeguridad_con_validacion” en la base de datos limpia y se asegura de que los datos migrados tengan todos los campos especificados y cumplan con el tipo de cada campo. El validador es el siguiente:

```
Unset
db.createCollection("IncidentesSeguridad_con_validacion", {
    validator: {
```

```
        $jsonSchema: {
            bsonType: "object",
              required: ["ID", "FECHA_REPORTE", "TIPO_INCIDENTE", "GRAVEDAD",
"AreaRecreativaID"],
            properties: {
                ID: { bsonType: "string", description: "ID del incidente"},
             TIPO_INCIDENTE: {
                    bsonType: "string",
                    enum: ["Vandalismo", "Robo", "Caida", "Accidente", "Daño
estructural"],
                    description: "Tipo de incidente"
                    },
             GRAVEDAD: {
                    bsonType: "string",
                    enum: ["Baja", "Media", "Alta", "Critica"],
                    description: "Gravedad del incidente"
                    },
                 FECHA_REPORTE: { bsonType: "date", description: "Fecha del
reporte en formato de fecha" },
                AreaRecreativaID: { bsonType: "string", description: "ID del
área recreativa en formato string" }
            }
        }
    }
})
```

Los campos especificados en *required*, son los campos que cada documento de incidente de seguridad debe de tener. En properties se especifican los tipos de cada campo, en el que ID tiene que ser un string. El campo TIPO_INCIDENTE tiene que ser un string y es un enumerativo donde los valores permitidos son "Vandalismo", "Robo", "Caida", "Accidente" y "Daño estructural". El campo GRAVEDAD tiene que ser un string y también es un enumerativo donde los valores permitidos son "Baja", "Media", "Alta" y "Critica". El campo FECHA_REPORTE tiene que ser de tipo date. Por último, AreaRecreativaID tiene que ser de tipo string. Se asocia una descripción a cada campo para que, en caso de que algún dato no se migre, se pueda saber por qué no ha entrado en la base de datos limpia.

Al igual que en los demás validadores, los campos definidos como necesarios se han establecido en base al diagrama UML del enunciado. Además, para poder referenciar los incidentes de seguridad de cada área se ha añadido el campo de AreaRecreativaID.

En esta colección, para asegurar la integridad de los datos, se ha creado un índice único en el campo ID, por lo que cada ID tiene que ser único y no puede haber ningún incidente de seguridad con el mismo ID. El índice es creado con el siguiente comando:

```
Unset
db.IncidentesSeguridad_con_validacion.createIndex({ ID: 1 }, { unique: true, name: "indiceIncidente" })
```

3.2.3. Mantenimiento

Para realizar el validador de mantenimiento, se crea una colección con el nombre de "Mantenimiento_con_validacion" con reglas de validación que aseguran que los datos migrados tengan los campos necesarios y que el tipo de cada campo sea el definido. El validador que hemos creado es el siguiente:

```
Unset
db.createCollection("Mantenimiento_con_validacion", {
    validator: {
        $jsonSchema: {
            bsonType: "object",
                 required: ["ID", "FECHA_INTERVENCION", "TIPO_INTERVENCION", "ESTADO_PREVIO", "ESTADO_POSTERIOR", "JuegoID"],
            properties: {
                  ID: { bsonType: "string", description: "ID único en formato string" },
                  FECHA_INTERVENCION: { bsonType: "date", description: "Fecha de intervención en formato de fecha" },
                TIPO_INTERVENCION: { bsonType: "string",
                    enum: ["Correctivo", "Emergencia", "Preventivo"],
                    description: "Categoriza el tipo de intervencion"
                    },
                   ESTADO_PREVIO: { bsonType: "string", description: "Estado previo al mantenimiento" },
                 ESTADO_POSTERIOR: { bsonType: "string", description: "Estado posterior al mantenimiento" },
                JuegoID: { bsonType: "string", description: "ID del juego"}
            }
        }
    }
})
```

Cada documento que se migra tiene que ser un objeto que incluya los campos ID, FECHA_INTERVENCION, TIPO_INTERVENCION, ESTADO_PREVIO, ESTADO_POSTERIOR (en base al UML del enunciado). Estos campos se han establecido como necesarios en *required.* Además, para poder referenciar los mantenimientos de cada juego se ha añadido el campo de AreaRecreativaID.

Los campos ID, ESTADO_PREVIO, ESTADO_POSTERIOR y JuegoID deben ser de tipo string; el campo FECHA_INTERVENCION debe ser de tipo date y el campo TIPO_INTERVENCION debe ser un string donde los valores permitidos son “Correctivo”, “Emergencia” y “Preventivo”. En caso de que algún dato migrado no cumpla con el tipo establecido, se mostrará como error la descripción que hemos definido.

Para que la colección tenga un campo único, hemos creado un índice en ID, es decir, todos los mantenimientos tendrán un identificador único y no habrá ningún mantenimiento con el mismo ID. El índice es creado con el siguiente comando:

```
Unset
db.Mantenimiento_con_validacion.createIndex({ ID: 1 }, { unique: true,
name:"indiceMantenimiento" })
```

Le atribuimos un nombre a este índice para que en caso de necesitar eliminarlo se pueda utilizar el nombre que le hemos puesto.

3.2.4. RegistroClima

El validador del clima, crea una colección con el nombre “RegistroClima_con_validacion” en la base de datos limpia, asegurando que todos los objetos que se migren a esta colección tengan los campos necesarios y cumplan con las propiedades de cada uno de los atributos. El validador utilizado es el siguiente:

```
Unset
db.createCollection("RegistroClima_con_validacion", {
    validator: {
        $jsonSchema: {
            bsonType: "object",
            required: ["ID", "Punto_Muestreo", "Fecha"],
            properties: {
```

```
                        ID: { bsonType: "string", description: "ID único en
formato int" },
                         Punto_Muestreo: { bsonType: "string", description:
"Identificación del punto de muestreo" },
                    Fecha: { bsonType: "date", description: "Fecha de registro
en formato de fecha" },
                     Temperatura: { bsonType: ["double", "null"], description:
"Temperatura registrada, opcional" },
                        Viento: { bsonType: ["bool", "null"], description:
"Velocidad del viento, si es fuerte o no, opcional" },
                          Precipitaciones: { bsonType: ["double", "null"],
description: "Cantidad de precipitaciones, opcional" },
                     CODIGO_POSTAL: { bsonType: "string", description: "Código
postal del punto de muestreo" }
                }
            }
        }
})
```

Los campos necesarios en la colección de RegistroClima son ID, Punto_Muestreo y Fecha. Los demás campos no se han establecido como necesarios ya que no hay información de todas las métricas para una fecha y lugar determinado, por lo que los campos de Temperatura, Viento y Precipitaciones no son obligatorios. En properties se definen las propiedades de cada uno de los campos donde ID, CODIGO_POSTAL y Punto_Muestreo tiene que ser de tipo string; Fecha de tipo date; Temperatura y Viento de tipo double; y Precipitaciones de tipo booleano. En caso de que alguna instancia migrada no cumpla con estas propiedades salta un error con la descripción que hemos definido. Los tipos definidos se han establecido en función del enunciado de la práctica. En el modelo UML no aparece el campo de código postal pero lo hemos añadido para poder relacionar el clima con las áreas.

Para que la colección tenga un campo único, hemos creado un índice en ID, con el nombre de “indiceClima”. El índice se crea con el siguiente comando:

```
Unset
db.RegistroClima_con_validacion.createIndex({  ID:  1  },  {  unique:  true,
name:"indiceClima" })
```

3.2.5. Incidencia

El validador de incidencia, crea la colección “IncidenciasUsuario_con_validacion” en nuestro dataset de colecciones limpias. Este validador define los campos que debe tener una incidencia y las propiedades de cada uno de ellos. El validador de esta colección es el siguiente:

```
Unset
db.createCollection("IncidenciasUsuario_con_validacion", {
    validator: {
        $jsonSchema: {
            bsonType: "object",
              required: ["ID", "TIPO_INCIDENCIA", "FECHA_REPORTE", "ESTADO",
"tiempoResolucion", "JuegoID", "Usuario", "nivelEscalamiento"],
            properties: {
                 ID: { bsonType: "string", description: "ID único en formato
string" },
               TIPO_INCIDENCIA: {
                   bsonType: "string",
                   enum:   ["Vandalismo",   "Desgaste",   "Rotura",   "Mal
funcionamiento"],
                   description: "Tipo de incidencia"
                   },
             ESTADO: {
                   bsonType: "string",
                   enum: ["Abierta", "Cerrada"],
                   description: "Estado de la incidencia"
                   },
                  FECHA_REPORTE: { bsonType: "date", description: "Fecha de
reporte en formato de fecha" },
                tiempoResolucion: { bsonType: "double", description: "Tiempo
de resolución en dias" },
             nivelEscalamiento: {
                   bsonType: "string",
                   enum: ["MUY URGENTE", "URGENTE", "POCO URGENTE"],
                   description: "Categoriza la urgencia de la incidencia"
                   },
       JuegoID: {bsonType: "string", description: "Id del juego"},
             Usuario: {
                   bsonType: "array",
                   description: "Lista de usuarios",
                   items: {
                         bsonType: "object",
                         required: ["NIF", "NOMBRE", "TELEFONO"],
                         properties: {
```

```
                                NIF:  {  bsonType:  "string",  pattern:
"^[0-9]{3}-[0-9]{2}-[0-9]{4}$", description: "NIF en formato de ddd-dd-dddd"
},
                                NOMBRE: { bsonType: "string", description:
"Nombre del usuario" },
                                EMAIL:  {  bsonType:  "string",  pattern:
"^(\\S+@\\S+\\.\\S+)?$", description: "Correo electrónico válido, opcional"
},
                                TELEFONO: { bsonType: "string", pattern:
"^34[0-9]{9}$", description: "Número de teléfono de 9 dígitos" }
                                },
                    minItems: 1}
                    }
            }
        }
    }
})
```

Los campos necesarios de una incidencia están definidos en *required*, siendo todos los campos de una incidencia obligatorios. En *properties* se definen una por una las propiedades que debe de tener cada campo.

- El identificador tiene que ser de tipo string.
- El tipo de incidencia tiene que ser un string, donde los valores permitidos solo pueden ser "Vandalismo", "Desgaste", "Rotura" o "Mal funcionamiento".
- El estado tiene que ser un string y solo puede tener los valores "Abierta" o "Cerrada".
- La fecha de reporte de la incidencia tiene que ser de tipo date.
- El tiempo de resolución tiene que ser un double.
- El nivel de escalamiento tiene que ser un string, donde los valores permitidos solo pueden ser "MUY URGENTE", "URGENTE" o "POCO URGENTE".
- El identificador del juego tiene que ser de tipo string. Este atributo se utiliza para referenciar incidencia en juego y poder relacionar cada incidencia con juego.
- Los usuarios están embebidos en incidencias, por lo que cada incidencia contiene la información de todos los usuarios que la han reportado. El campo Usuario es un array de objetos con la información de todos los usuarios de una incidencia. Cada usuario (objeto) del array debe de tener los campos "NIF", "NOMBRE" y "TELEFONO", siendo “EMAIL” un atributo opcional. El NIF tiene que ser un string con el formato ddd-ddd-dddd. El nombre del usuario debe de ser de tipo string. El email tiene que ser un string que puede ser o bien una cadena vacía o un email válido sin ningún espacio y que debe incluir un @ y después al menos un punto. El teléfono tiene que ser de tipo string, tiene que

empezar por 34 y seguir con nueve dígitos adicionales. Con minItems, definimos que al menos debe de haber un usuario por cada incidencia.

En caso de que alguno de los datos migrados no cumpla con las propiedades se muestra la descripción que hemos establecido para cada uno de los campos.

Para que cada incidencia tenga un identificador único, creamos un índice en el campo ID con el nombre de “indiceIncidencia”.

```
Unset
db.IncidenciasUsuario_con_validacion.createIndex({ ID: 1 }, { unique: true, name: "indiceIncidencia"})
```

3.2.6. Juego

El validador de juego crea la colección “Juegos_con_validacion”. Cada documento debe de ser de tipo objeto, que tendrá como necesarios los campos especificados en *required* y cada campo tendrá unas propiedades específicas que hemos definido. El validador es el siguiente:

```
Unset
db.createCollection("Juegos_con_validacion", {
    validator: {
        $jsonSchema: {
            bsonType: "object",
                  required: ["ID", "ESTADO", "FECHA_INSTALACION", "MODELO", "tipo_juego",       "indicadorExposicion",       "ultimaFechaMantenimiento", "desgasteAcumulado", "AreaRecreativaID"],
            properties: {
                    ID: { bsonType: "string", description: "ID único de cada registro, debe ser un entero" },
                  ESTADO: { bsonType: "string", description: "Estado actual del registro", enum: ["OPERATIVO", "EN MANTENIMIENTO"]},
                FECHA_INSTALACION: {bsonType: "date", description: "Fecha de instalación"},
                  AreaRecreativaID: {bsonType: "string", description: "ID de Area"},
                     MODELO: { bsonType: "string", description: "Modelo del elemento" },
                    tipo_juego: { bsonType: "string", description: "Tipo de juego", enum: ["mayores", "deportivas", "infantiles"]},
                      ACCESIBLE: { bsonType: ["bool", "null"], description: "Indica si el elemento es accesible o no"},
```

```
                    indicadorExposicion: { bsonType: "string", description:
"Indicador de exposicion", enum: ["alto" , "medio", "bajo"]},
                     ultimaFechaMantenimiento: {bsonType: ["date", "null"],
description: "Fecha de último mantenimiento"},
                desgasteAcumulado: { bsonType: "int",description: "Nivel de
desgaste acumulado"},
            Mantenimiento: { bsonType: "array", description: "lista de ids
de mantenimientos",
                  items: {bsonType: "string", description: "referencia a
un mantenimiento mediante su id"}},
            Incidencia: { bsonType: "array",
                  description: "Lista de referencias a incidentes de
seguridad, usando el campo 'idIncidente' del dataset de incidentes",
                  items: {
                        bsonType: "object",
                        required: ["ID", "TIPO_INCIDENCIA", "ESTADO",
"FECHA_REPORTE"],
                        properties: {
                              ID: {
                                    bsonType: "string",
                                    description: "ID del incidente"
                              },
                        TIPO_INCIDENCIA: {
                              bsonType: "string",
                              enum: ["Vandalismo", "Desgaste", "Rotura",
"Mal funcionamiento"],
                              description: "Tipo de incidencia"
                              },
                        ESTADO: {
                              bsonType: "string",
                              enum: ["Abierta", "Cerrada"],
                              description: "Estado de la incidencia"
                              },
                        FECHA_REPORTE: {
                              bsonType: "date",
                              description: "Fecha en que se reportó la
incidencia"
                              }
                        }
                  }
            }
        }
    }
    }
});
```

Se definen en *required* como campos obligatorios "ID", "ESTADO", "FECHA_INSTALACION", "MODELO", "tipo_juego", "indicadorExposicion", "ultimaFechaMantenimiento", "desgasteAcumulado" y "AreaRecreativaID", siendo opcional “ACCESIBLE”, “Mantenimiento”, “Incidencia” y “ultimaFechaMantenimiento” (no todos los juegos tienen información sobre estos campos). En *properties* se definen una por una las propiedades que debe de tener cada campo.

- El identificador tiene que ser de tipo string.
- El estado tiene que ser un string y solo puede tener los valores "OPERATIVO" o "EN MANTENIMIENTO".
- La fecha de instalación de un juego tiene que ser de tipo date.
- El identificador de área tiene que ser de tipo string. Este atributo se utiliza para referenciar juego en área y poder relacionar cada juego con el área a la que pertenece.
- El modelo tiene que ser de tipo string.
- El tipo de juego tiene que ser un string, donde los valores permitidos solo pueden ser "mayores", "deportivas"o "infantiles".
- La accesibilidad puede ser o un booleano o nulo. Esto es debido a que es un atributo que se calcula a partir de las encuestas de satisfacción de cada área y hay áreas que no tienen ninguna encuesta. Esto no es un error y por eso el calor puede ser nulo.
- El indicador de exposición de un juego tiene que ser de tipo string, donde los valores permitidos solo pueden ser "alto" , "medio"o "bajo".
- La última fecha de mantenimiento puede ser de tipo date o null. Esto es debido a que no todos los juegos han tenido un mantenimiento por lo que en este campo permitimos que haya valores nulos.
- El desgaste acumulado tiene que ser un entero que indica el nivel de desgaste de un juego.
- En el agregado de juego los mantenimientos tienen que estar referenciados. Por ello, cada juego contiene el o los identificadores de todos los mantenimientos que un juego ha tenido. El campo Mantenimiento es un array de strings con los identificadores de cada mantenimiento asociado a ese juego. Cada identificador de mantenimiento del array tiene que ser de tipo string. En caso de que un juego no tenga ningún mantenimiento, este campo será un array vacío.
- En el agregado de juego las incidencias se referencian con resumen, por lo que cada juego contiene información de algunos de los campos con información más relevante de una incidencia (los especificados en el enunciado). El atributo Incidencia es un array de objetos con la información necesaria de las incidencias de un juego. Cada incidencia (objeto) debe de tener los campos "ID", "TIPO_INCIDENCIA", "ESTADO" y "FECHA_REPORTE". El

identificador tiene que ser de tipo string. El tipo de incidencia tiene que ser un string, donde los valores permitidos solo pueden ser "Vandalismo", "Desgaste", "Rotura" o "Mal funcionamiento". El estado tiene que ser un string y solo puede tener los valores "Abierta" o "Cerrada". La fecha de reporte de la incidencia tiene que ser de tipo date. En caso de que un juego no tenga ninguna incidencia reportada , este campo será un array vacío.

Si los datos migrados no cumplen con las propiedades, se muestra la descripción que hemos establecido para cada uno de los campos.

Para que el ID de los juegos sea único se crea un índice para asegurar que no hay dos instancias de juego con un mismo identificador y garantizar la integridad de la base de datos limpia.

```
Unset
db.Juegos_con_validacion.createIndex({ ID: 1 }, { unique: true, name:
"indiceJuego" })
```

3.2.7. Área Recreativa

Para realizar el validador de área recreativa , se crea una colección con el nombre de “Encuestas_con_validacion”. Cada dato que se migra debe cumplir con los campos necesarios y las propiedades establecidas en este validador:

```
Unset
db.createCollection("Areas_con_validacion", {
    validator: {
        $jsonSchema: {
            bsonType: "object",
              required: ["ID", "BARRIO", "DISTRITO", "ESTADO", "COD_POSTAL",
"FECHA_INSTALACION",     "coordenadasGPS",     "capacidadMax",     "Juegos",
"cantidadJuegosPorTipo", "estadoGlobalArea"],
            properties: {
                    ID: { bsonType: "string", description: "ID único de cada
registro, debe ser un entero" },
                     BARRIO: { bsonType: "string", description: "Nombre del
barrio" },
                    DISTRITO: { bsonType: "string", description: "Nombre del
distrito" },
```

```
                    ESTADO: { bsonType: "string", description: "Estado actual
del   registro",   enum:   ["OPERATIVO",   "EN   MANTENIMIENTO",   "FUERA   DE
SERVICIO"]},
                      COD_POSTAL: {
                    bsonType: "string",
                    pattern: "^[0-9]{5}$",
                    description: "Código postal de 5 dígitos"
                },
                  FECHA_INSTALACION: { bsonType: "date", description: "Fecha
de instalación en formato de fecha" },
                         coordenadasGPS: { bsonType: "string", description:
"Coordenadas GPS del área"},
                    capacidadMax: { bsonType: "int", description: "Capacidad
máxima del área, debe ser un entero" },
                               cantidadJuegosPorTipo:  {bsonType:  "object",
description: "Cantidad de juegos por tipo",
                    properties: {
                           deportivas: {bsonType: "int", description: "Juegos
de tipo deportivos"},
                           infantiles: {bsonType: "int", description: "Juegos
de tipo infantiles"},
                           mayores: {bsonType: "int", description: "Juegos de
tipo mayores"}
                    }
                },
                         estadoGlobalArea: {bsonType: "string", description:
"Estado global del area", enum: ["BUENO", "SUFICIENTE", "INSUFICIENTE"]},
                         Juegos: {bsonType: "array", description: "lista de
referencias a juegos",
                    items: {
                           bsonType: "string", description: "referencia a un
juego mediante su id"}
                    },
                IncidenteSeguridad: {
                    bsonType: "array",
                         description: "Lista de referencias a incidentes de
seguridad, usando el campo 'idIncidente' del dataset de incidentes",
                    items: {
                        bsonType: "object",
                              required: ["ID", "TIPO_INCIDENTE", "GRAVEDAD",
"FECHA_REPORTE"],
                        properties: {
                            ID: {
                                    bsonType: "string",
                                    description: "ID del incidente"
                                },
                            TIPO_INCIDENTE: {
                                    bsonType: "string",
```

```
                                        enum: ["Vandalismo", "Robo", "Caida",
"Accidente", "Daño estructural"],
                                    description: "Tipo de incidente"
                                },
                            GRAVEDAD: {
                                    bsonType: "string",
                                              enum: ["Baja", "Media", "Alta",
"Critica"],
                                    description: "Gravedad del incidente"
                                },
                            FECHA_REPORTE: {
                                    bsonType: "date",
                                     description: "Fecha en que se reportó el
incidente"
                                }
                        }
                    }
                },
                   RegistroClima: {bsonType: "array", description: "lista de
referencias a clima",
                    items: {
                           bsonType: "string", description: "referencia a un
clima mediante su id"}
                    },
                      EncuestaSatisfaccion: {bsonType: "array", description:
"lista de referencias a encuestas",
                    items: {
                          bsonType: "string", description: "referencia a una
encuesta mediante su id"}
                        }
            }
        }
    }
});
```

Los campos necesarios de un área recreativa están definidos en *required*. En *properties* se definen una por una las propiedades que debe de tener cada campo. Las propiedades de cada campo son las siguientes y han sido definidas en función del enunciado y diagrama UML:

- El identificador tiene que ser de tipo string.
- El barrio tiene que ser de tipo string.
- El distrito tiene que ser de tipo string.
- El estado tiene que ser un string, donde los valores permitidos solo pueden ser "OPERATIVO", "EN MANTENIMIENTO" o "FUERA DE SERVICIO".

- El código postal tiene que ser de tipo string y debe de cumplir con el formato ddddd, donde d hace referencia a un carácter numérico. Este valor no está en el modelo pero es necesario para referenciar los datos del clima.
- La fecha de instalación de un área tiene que ser de tipo date.
- Las coordenadas GPS tienen que ser de tipo string.
- La capacidad máxima de un área tiene que ser un entero.
- La cantidad de juegos por tipo es un objeto. Este objeto contiene tres propiedades: deportivas, infantiles y mayores, cada una de las cuales tienen que ser un entero.
- El estado global del área tiene que ser un string, donde los valores permitidos solo pueden ser "BUENO", "SUFICIENTE" o "INSUFICIENTE".
- En el agregado de área los juegos tienen que estar referenciados. Por ello, cada área contiene el o los identificadores de todos los juego que un área recreativa tiene. El campo juegos es un array de strings con los identificadores de cada juego asociado a ese área. Cada identificador de juego del array tiene que ser de tipo string.
- En el agregado de área los incidentes de seguridad se referencian con resumen, por lo que cada área contiene información de algunos de los campos con información más relevante de un incidente. El atributo IncidenteSeguridad es un array de objetos con la información necesaria de los incidentes de un área. Cada objeto debe de tener los campos "ID", "TIPO_INCIDENTE", "GRAVEDAD" y "FECHA_REPORTE". El identificador tiene que ser de tipo string. El tipo de incidente tiene que ser un string, donde los valores permitidos solo pueden ser "Vandalismo", "Robo", "Caida", "Accidente", "Daño estructural". La gravedad tiene que ser un string y solo puede tener los valores "Baja", "Media", "Alta" o "Critica". La fecha de reporte tiene que ser de tipo date. En caso de que un área no tenga ningún incidente de seguridad asociado, este campo será un array vacío.
- En el agregado de área los registros del clima tienen que estar referenciados. El atributo RegistroClima es un array de strings que tiene los identificadores de todos los registros que un área recreativa tiene. En caso de que un área no tenga ningún registro de clima, este campo será un array vacío.
- Las encuestas de satisfacción también tiene que estar referenciadas en área. El atributo EncuestaSatisfaccion es un array de strings que tiene los identificadores de todos las encuestas que un área recreativa tiene. En caso de que un área no tenga ninguna encuesta, este campo será un array vacío.

Si los datos migrados no cumplen con las propiedades, se muestra la descripción que hemos establecido para cada uno de los campos.

Para que el ID de las áreas sea único se crea un índice para asegurar que no haya dos áreas con un mismo identificador.

```
Unset
db.Areas_con_validacion.createIndex({ ID: 1 }, { unique: true, name:
"indiceArea" })
```

3.3. Diseño

Para poder realizar la migración de los datos de una base de datos a otra, hemos necesitado crear tres bases de datos:

- **Base de datos sucia (bd_sin_validacion) :** en esta base de datos se cargan los datos y se crean las colecciones iniciales sin ningún tipo de validación, es decir, se cargan los datasets resultantes de la limpieza de python.

- **Base de datos limpia (bd_con_validacion)** : esta base de datos tiene todas las validaciones de cada colección por lo que todos los datos que se migran están limpios y con la estructura definida.

- **Base de datos temporal (bd_temporal) :** esta base de datos ha sido creada para poder hacer las referencias que requieren los agregados. A lo hora de hacer las migraciones teníamos problemas para referenciar documentos que estaban en bases distintas. Por ejemplo, para crear la colección de área partíamos de la colección de área en la base de datos sin validación y había que referenciar a juegos, encuestas e incidentes de la base de datos con validación. Mongo no permite hacer estas referencias por lo que creamos la base temporal, en la que, en el caso de este ejemplo, tendríamos tanto la colección de áreas sin validación como la de juegos, encuestas e incidentes de seguridad. Decidimos crear esta base de datos temporal ya que no queríamos mezclar en las bases de datos que ya teníamos colecciones sucias y limpias.
  Para mover las colecciones a la base de datos temporal o hacemos lo siguiente:

```
Unset
db.coleccion.aggregate([
    {
        $match: {}
    },
    {
        $merge: {
            into: { db: "bd_nombre", coll: "nombrecoleccion_temporal" },
            whenMatched: "merge",  // Para fusionar los documentos
```

```
            whenNotMatched: "insert"         }
    }
]);
```

Donde “coleccion” hace referencia al nombre de la colección que se quiere mover a la base de datos “bd_nombre” y “nombrecoleccion_temporal” hace referencia al nombre de la colección que se va a crear.

3.4. Migraciones

Para migrar los datos decidimos seguir una estrategia similar a las validaciones, optamos por empezar por colecciones con menor número de atributos y más simples y luego progresivamente avanzar hacia colecciones más complejas como juego o área.

3.4.1. Encuesta

La migración de datos de la encuesta se realiza a partir de la colección “Encuestas_sin_validacion” en la base de datos sucia y se migra a la colección de la base de datos limpia llamada “Encuestas_con_validacion”.

```
Unset
db.Encuestas_sin_validacion.aggregate([
    {
        $project: {
            ID: { $toString: "$ID" },
            PUNTUACION_ACCESIBILIDAD: 1,
            PUNTUACION_CALIDAD: 1,
            COMENTARIOS: 1,
            FECHA: {
                $dateFromString: {
                    dateString: "$FECHA",
                    format: "%Y-%m-%d"
                }
            },
            AreaRecreativaID: { $toString: "$AreaRecreativaID" }
        }
    },
    {
        $merge: {
```

```
                                into:  {  db:  "bd_con_validacion",  coll:
"Encuestas_con_validacion" },
            on: "ID",
            whenMatched: "keepExisting",
            whenNotMatched: "insert"
        }
    }
]);
```

En *project* se definen los campos que se van a migrar y se realizan las transformaciones de tipo de los campos necesarios, cumpliendo con los tipos establecidos en el validador de esta colección. El ID en el dataset inicial es un entero, por lo que lo convertimos en string al igual que AreaRecreativaID. PUNTUACION_ACCESIBILIDAD, PUNTUACION_CALIDAD y COMENTARIOS se mantienen sin modificaciones pero son campos que deben estar. El campo FECHA se convierte a una fecha ya que en el archivo csv se detecta como un string. Es importante que todos los datos de este campo estén en el formato "%Y-%m-%d" porque sino mongo no lo puede convertir a tipo date.

Con *merge,* los datos se migran a la colección especificada con el nombre “Encuestas_con_validacion” en la base de datos con validación. Si el ID ya existe en la base de datos, no se insertará, respetando que el ID debe de ser único. Si el ID no se encuentra en la colección de validación el documento se insertará.

3.4.2. Incidente Seguridad

La migración de datos de incidente de seguridad se realiza a partir de la colección “IncidentesSeguridad_sin_validacion” en la base de datos sucia y se migra a la colección de la base de datos limpia llamada “IncidentesSeguridad_con_validacion”. La migración se realiza de la siguiente manera:

```
Unset
db.IncidentesSeguridad_sin_validacion.aggregate([
    {
        $project: {
            ID: { $toString: "$ID" },
            FECHA_REPORTE: {
                $dateFromString: {
                    dateString: "$FECHA_REPORTE",
                    format: "%Y-%m-%d"
```

```
                }
            },
            TIPO_INCIDENTE: 1,
            GRAVEDAD: 1,
            AreaRecreativaID: { $toString: "$AreaRecreativaID" }
        }
    },
    {
        $merge: {
                                into:  {  db:  "bd_con_validacion",  coll:
"IncidentesSeguridad_con_validacion" },
            on: "ID",
            whenMatched: "keepExisting",
            whenNotMatched: "insert"
        }
    }
]);
```

En *project* se definen los campos de la colección que se van a migrar y se realizan las conversiones de tipo para que sean válidas para la colección con validación. ID y AreaRecreativaID se convierten en strings; el campo FECHA_REPORTE, que es un string con el formato que Mongo reconoce, se convierte a date y TIPO_INCIDENTE y GRAVEDAD son atributos que deben estar pero no se realiza ningún cambio.

Los datos se migrarán a la base de datos con validación y a la colección correspondiente siempre y cuando el campo ID sea único.

3.4.3. Mantenimiento

En el proceso de migración de mantenimiento se migra la colección “Mantenimiento_sin_validación” a la colección “Mantenimiento_con_validacion” de la base de datos con validadores.

Unset

```
db.Mantenimiento_sin_validacion.aggregate([
    {
        $project: {
            ID: 1,
            FECHA_INTERVENCION: {
                $dateFromString: {
                    dateString: "$FECHA_INTERVENCION",
                    format: "%Y-%m-%d"
                }
```

```
        },
            TIPO_INTERVENCION: 1,
            ESTADO_PREVIO: 1,
            ESTADO_POSTERIOR: 1,
            JuegoID: { $toString: "$JuegoID" }
        }
    },
    {
        $merge: {
                                into:  {  db:  "bd_con_validacion",  coll:
"Mantenimiento_con_validacion" },
           on: "ID",
            whenMatched: "keepExisting",
            whenNotMatched: "insert"
        }
    }
]);
```

Al igual que con las demás, en *project* se definen los campos necesarios para la migración y se convierten sus tipos para que puedan cumplir con la validación. Los campos ID y JuegoID se convierten a strings; el campo FECHA_INTERVENCION se convierte a date y los demás campos no requieren ninguna transformación.

Para poder realizar merge a la base de datos con validación en la colección especificada, todos los identificadores (ID) tienen que ser únicos, sino no lo insertará para no sobrescribir el dato con el ID ya migrado.

3.4.4. RegistroClima

Los datos de registro clima se migran de la colección “RegistroClima_sin_validacion”, almacenada en la base datos inicial, a la colección “ResgistroClima_con_validacion” de la siguiente manera:

```
Unset
db.RegistroClima_sin_validacion.aggregate([
    {
        $project: {
            ID: { $toString: "$ID" },
            Punto_Muestreo: { $toString: "$Punto_Muestreo" },
            CODIGO_POSTAL: { $toString: "$CODIGO_POSTAL" },
            Fecha: {
                $dateFromString: {
                    dateString: "$Fecha",
```

```
                    format: "%Y-%m-%d"
                }
            },
            Temperatura: {
                $cond: {
                if: { $eq: ["$Temperatura", ""] },
                then: null,
                else: {$toDouble: "$Temperatura"}
                }
            },
            Viento: {
                $cond: {
                if: { $eq: ["$Viento", "True"] },
                then: true,
                else: {
                    $cond: {
                        if: { $eq: ["$Viento", "False"] },
                        then: false,
                        else: null
                    }
                }
                }
            },
               Precipitaciones: {
                $cond: {
                if: { $eq: ["$Precipitaciones", ""] },
                then: null,
                else: {$toDouble: "$Precipitaciones"}
                }
            }
        }
    },
    {
        $merge: {
                                into:   {   db:   "bd_con_validacion",   coll:
"RegistroClima_con_validacion" },
               on: "ID",
            whenMatched: "keepExisting",
            whenNotMatched: "insert"
        }
    }
]);
```

En *project* se especifican los campos de la migración y se realizan conversiones. Los campos ID, Punto_Muestreo y CODIGO_POSTAL se convierten a cadenas de caracteres. El campo FECHA, que debe estar en formato "%Y-%m-%d" y tiene que ser

un string, se transforma al tipo date de Mongo. Los campos Temperatura y Precipitaciones si su valor es vacío (“”), se asigna null y si no son vacíos se convierten a dobles. Para el campo Viento, si el valor es “True” (booleano en python) se tiene que convertir a true (formato de boolean que reconoce Mongo), si es “False”, se convierte a false, y si está vacío se le asigna el valor null.

Los datos se migran a la colección “RegistroClima_con_validacion”, asegurando que ningún documento tiene el mismo ID.

3.4.5. Incidencia

En incidencia, los dato se migran de la colección “IncidenciasUsuario_sin_validacion” de la base de datos sin validación a la colección “IncidenciasUsuario_con_validacion” de la base de datos con validación, embebiendo en la incidencia los datos de la coleccion “Usuarios_sin_validacion”. La migración realizada es la siguiente:

```
Unset
db.IncidenciasUsuarios_sin_validacion.aggregate([
    {
               $addFields: {
            UsuarioNIFs: {
                $map: {
                    input: {
                        $split: [
                               { $trim: { input: "$UsuarioID", chars: "[] " } 
},
                            ", "
                        ]
                    },
                    as: "nif",
                    in: { $trim: { input: "$$nif", chars: "'\"" } }
                }
            }
        }
    },
    {
               $lookup: {
            from: "Usuarios_sin_validacion",
            localField: "UsuarioNIFs",
            foreignField: "NIF",
             as: "UsuarioData"
        }
    },
    {
        // Proyectar los datos necesarios
```

```
        $project: {
            ID: { $toString: "$ID" },
            TIPO_INCIDENCIA: 1,
            FECHA_REPORTE: {
                $dateFromString: {
                    dateString: "$FECHA_REPORTE",
                    format: "%Y-%m-%d"
                }
            },
            ESTADO: 1,
            tiempoResolucion: { $toDouble: "$tiempoResolucion"},
            nivelEscalamiento: 1,
            JuegoID: { $toString: "$JuegoID" },
            Usuario: {
              $map: {
                    input: "$UsuarioData",
                    as: "user",
                    in: {
                        NIF: "$$user.NIF",
                        NOMBRE: "$$user.NOMBRE",
                        EMAIL: "$$user.EMAIL",
                        TELEFONO: { $toString: "$$user.TELEFONO" }
                    }
                }
            }
        }
    },
    {
               $merge: {
                                    into:   {   db:   "bd_con_validacion",   coll:
"IncidenciasUsuario_con_validacion" },
            on: "ID",
            whenMatched: "keepExisting",
            whenNotMatched: "insert"
        }
    }
]);
```

El primer problema que tuvimos con esta migración fue que, en la base de datos de las incidencias, los datos de los usuarios que han reportado una incidencia se encuentran en el siguiente formato: "['877-15-7376', '467-34-2729']". De estas cadenas, que pueden contener uno o varios NIFs, tenemos que extraer únicamente los NIFs porque es el identificador en la base de datos de usuarios sin validación. Necesitamos quitar los corchetes y las comillas simples para poder obtener el NIF de el o los usuarios de las incidencias. Por ello, creamos un nuevo campo con *addField*

llamado UsuarioNIFs, que es un array que contendrá todos los usuarios de una incidencia, sacados a partir del campo UsuarioID. Para poder meter en el nuevo array los usuarios, eliminamos los corchetes y espacios, separamos los datos por comas (obteniendo cada NIF) y creamos el array con estos datos. Finalmente, se utiliza map para extraer las comillas simples de cada elemento del array y se almacena el array modificado en el nuevo campo UsuarioNIFs.

A continuación, se realiza una búsqueda con *lookup* en la colección "Usuarios_sin_validacion", utilizando los NIFs del nuevo array UsuarioNIFs para encontrar los usuarios que han reportado la incidencia. Los usuarios encontrados se almacenan en UsuarioData, el cual contendrá todos los documentos de usuarios que coincidan con los NIFs que tenía el dataset de incidencias.

Después especificamos los campos que incidencia debe tener y se realizan las conversiones de los datos. El campo ID y JuegoID se convierten a string. El campo FECHA_REPORTE, que es un string con formato "YYYY-MM-DD", se transforma a date de Mongo. El campo tiempoResolución se convierte a double. Los demás campos se mantienen igual.

El campo Usuario se encuentra embebido e incluye los usuarios que han reportado esa incidencia. Para poder hacer esto, con el operador map recorremos el array UsuarioData que contiene toda la información de los usuarios de esa incidencia y se almacena la información de cada usuario en el campo correspondiente. Además, el campo TELEFONO de cada usuario se convierte a string.

Finalmente, se realiza la migración de una base de datos a otra, siendo el campo ID único.

3.4.6. Juego

A diferencia de las demás colecciones explicadas, en el caso de juego se hace la migración desde la colección "Juego_temporal" de la base de datos temporal a la colección llamada "Juegos_con_validacion" de la misma base de datos. Esto se hace por el problema de las referencias de Mongo. Al terminar la migración se pasará la colección a la base de datos con validación.

```
Unset
db.Juego_temporal.aggregate([
        {
        $project: {
            ID: { $toString: "$ID" },  // Convertir ID de entero a string
            ESTADO: 1,
            FECHA_INSTALACION: {
```

```
                $dateFromString: {
                    dateString: "$FECHA_INSTALACION",
                    format: "%Y-%m-%d"
                }
            },
            MODELO: { $toString: "$MODELO" },
            AreaRecreativaID: { $toString: "$AreaRecreativaID" },
            tipo_juego: 1,
            ACCESIBLE: {
                $cond: {
                    if: { $eq: ["$ACCESIBLE", "True"] },
                    then: true,
                    else: {
                        $cond: {
                            if: { $eq: ["$ACCESIBLE", "False"] },
                            then: false,
                            else: null
                        }
                    }
                }
            },
            indicadorExposicion: 1,
            ultimaFechaMantenimiento: {
                $cond: {
                    if: { $eq: ["$ultimaFechaMantenimiento", "Ausente"] },
                    then: null,
                    else: {
                        $dateFromString: {
                            dateString: "$ultimaFechaMantenimiento",
                            format: "%Y-%m-%d"
                        }
                    }
                }
            },
            desgasteAcumulado: 1,
            Mantenimiento: 1,
           Incidencia: 1,
        }
    },
       {
        $lookup: {
            from: "Mantenimiento_temporal",
            localField: "ID",
        foreignField: "JuegoID",
                   as: "MantenimientoID"
        }
    },
    // Tercer paso: realizar el segundo $lookup
```

```
{
    $lookup: {
        from: "IncidenciasUsuario_temporal",
        localField: "ID",
        foreignField: "JuegoID",
        as: "IncidenciaID"
    }
},
    {
    $project: {
        ID: 1,
        ESTADO: 1,
        FECHA_INSTALACION: 1,
        MODELO: 1,
        AreaRecreativaID: 1,
        tipo_juego: 1,
        ACCESIBLE: 1,
        indicadorExposicion: 1,
        ultimaFechaMantenimiento: 1,
        desgasteAcumulado: 1,

        Mantenimiento: {
            $map: {
                input: "$MantenimientoID",
                as: "mant",
                in: "$$mant.ID"
            }
        },

        Incidencia: {
            $map: {
                input: "$IncidenciaID",
                as: "inc",
                in: {
                    ID: "$$inc.ID",
                    TIPO_INCIDENCIA: "$$inc.TIPO_INCIDENCIA",
                    ESTADO: "$$inc.ESTADO",
                    FECHA_REPORTE: "$$inc.FECHA_REPORTE"
                }
            }
        }
    }
},
   {
    $merge: {
        into: { db: "bd_temporal", coll: "Juegos_con_validacion" },
        on: "ID",
        whenMatched: "keepExisting",
```

```
            whenNotMatched: "insert"
        }
    }
]);
```

Un problema que tuvimos en esta migración fue para hacer las referencias a los mantenimientos y a las incidencias. En las colecciones de mantenimiento e incidencias con validacion el campo de JuegoID es un string, pero en el dataset original de juego es un entero. Aunque tengan el mismo valor, si son tipos diferentes mongo no los detecta y no se encontraban las referencias. Por ello, dividimos la migración en dos operaciones de *project*.

En la primera realizamos todos los cambios necesarios sobre todas las columnas pero sin hacer las referencias a mantenimiento e incidencia. El campo ID, el campo AreaRecreativaID y el campo MODELO se convierten a string. El campo FECHA_INSTALACION se convierte a date de Mongo. Para el campo ACCESIBLE, si el valor es “True” (booleano en python) se tiene que convertir a true (booleano que reconoce Mongo), si es “False”, se convierte a false, y si está vacío o tiene cualquier otro valor se toma como nulo. Para el campo ultimaFechaMantenimiento, si el valor es “Ausente” se convierto en nulo, sino será una fecha por lo que se convierte a date. Los demás campos no necesitan ninguna conversión, pero tienen que estar.

Con estas transformaciones ya podemos hacer las operaciones de lookup porque el campo de ID ya se ha convertido a string.

Se realiza un lookup para que en MantenimientoID (array) se almacenen todos los mantenimientos que están relacionados con ese juego. Se meten en el array todos aquellos mantenimientos cuyo JuegoID coincide con el ID del juego.

Se realiza otro lookup para que en IncidenciasID (array) se almacenen todas las incidencias que están relacionadas con ese juego. Se meten en el array todas aquellas incidencias de usuario cuyo JuegoID coincide con el ID del juego.

Se realiza un segundo *project* para definir los campos de Mantenimiento e Incidencia. En el campo de mantenimiento se utiliza map para recorrer todos los mantenimientos de MantenimientoID y guardar en el array el atributo ID de todos los mantenimientos asociados a ese juego. De esta forma se realiza la referencia de mantenimiento en juego. En el campo Incidencia, que es una referencia con resumen, se guarda información de cada incidencia relacionada con ese juego, almacenando los atributos ID, TIPO_INCIDENCIA, ESTADO y FECHA_REPORTE. Para ello, con map se recorre el

array de IncidenciasID y crea un objeto incidencia por cada incidencia del array con esos campos.

Finalmente, se realiza la migración de una base de datos a otra, siendo el campo ID único.

3.4.7. Área Recreativa

El caso de área recreativa es similar al de juego, es necesario utilizar la base de datos temporal para hacer las referencias y después pasar la colección a la base de datos con validación. La migración se hace de la siguiente manera:

```
Unset
db.Areas_temporal.aggregate([
    {
        $addFields: {
            cantidadJuegosPorTipoObj: {
                $function: {
                    body: function(cantidadJuegosPorTipo) {
                              // Reemplaza las comillas simples con comillas
dobles
                                                          cantidadJuegosPorTipo  =
cantidadJuegosPorTipo.replace(/'/g, "\"");
                        // Intenta convertir la cadena a un objeto JSON
                        try {
                            return JSON.parse(cantidadJuegosPorTipo);
                        } catch (e) {
                                return null; // Devuelve null si no se puede
convertir
                        }
                    },
                    args: ["$cantidadJuegosPorTipo"],
                    lang: "js"
                }
            }
        }
    },
    {
        $project: {
            ID: { $toString: "$ID" },  // Convertir ID de entero a string
            ESTADO: 1,
            FECHA_INSTALACION: {
                $cond:{
                    if: { $eq: ["$FECHA_INSTALACION", ''] },
                    then: null,
                    else: {
```

```
                        $dateFromString: {
                            dateString: "$FECHA_INSTALACION",
                            format: "%Y-%m-%d"
                        }
                    }
                }
            },
            BARRIO: 1,
            DISTRITO: 1,
            coordenadasGPS: 1,
            capacidadMax: 1,
            cantidadJuegosPorTipo: {
                 deportivas: { $toInt: "$cantidadJuegosPorTipoObj.deportivas" },
                 infantiles: { $toInt: "$cantidadJuegosPorTipoObj.infantiles" },
                mayores: { $toInt: "$cantidadJuegosPorTipoObj.mayores" }
            },
            estadoGlobalArea: 1,
            Juegos: 1,
            IncidenteSeguridad: 1,
            RegistroClima: 1,
            EncuestaSatisfaccion: 1,
            COD_POSTAL: { $toString: "$COD_POSTAL" }
        }
    },
    {
        $lookup: {
              from: "Juegos_con_validacion",  // Especifica el nombre de la colección externa (o colección "destino") con la que se hará la unión.
             localField: "ID",          // Es el campo en el documento de la colección principal (colección "origen") que MongoDB utiliza para hacer la unión
            foreignField: "AreaRecreativaID",             // Es el campo en la colección "destino" (especificada en from) en el que MongoDB busca valores coincidentes con localField.
            as: "JuegosID"            // Define el nombre del campo en el que se almacenará el resultado de la unión dentro del documento principal.
        }
    },
    {
        $lookup: {
            from: "IncidentesSeguridad_temporal",
            localField: "ID",
            foreignField: "AreaRecreativaID",
            as: "IncidenteSeguridadID"
        }
    },
```

```
    {
    $lookup: {
        from: "RegistroClima_temporal",
        let: {
            areaCodPostal: "$COD_POSTAL",
            areaFechaInstalacion: "$FECHA_INSTALACION"
        },
        pipeline: [
            {
                $match: {
                    $expr: {
                        $and: [
                            {
                                $regexMatch: {
                                    input: "$CODIGO_POSTAL",
                                                regex: { $concat: [",?",
"$$areaCodPostal", ",?"] },
                                    options: "i"
                                }
                            },
                            {
                                    $gt: ["$Fecha", "$$areaFechaInstalacion"]
// Comprobación para que la fecha en clima sea posterior a FECHA_INSTALACION
                            }
                        ]
                    }
                }
            }
        ],
        as: "RegistrosClimaID"
    }
    },
    {
        $lookup: {
            from: "Encuestas_temporal",
            localField: "ID",
            foreignField: "AreaRecreativaID",
            as: "EncuestaSatisfaccionID"
        }
    },
    {
        $project: {
            ID: 1,
            BARRIO: 1,
            DISTRITO: 1,
            ESTADO: 1,
            COD_POSTAL: 1,
            NDP: 1,
```

```
            FECHA_INSTALACION: 1,
            CODIGO_INTERNO: 1,
            coordenadasGPS: 1,
            capacidadMax: 1,
            cantidadJuegosPorTipo: 1,
            estadoGlobalArea: 1,
                  Juegos: { $map: { input: "$JuegosID", as: "juego", in:
"$$juego.ID" }},
            IncidenteSeguridad: {
                $map: {
                    input: "$IncidenteSeguridadID",
                    as: "incidente",
                    in: {
                        ID: "$$incidente.ID",
                        TIPO_INCIDENTE: "$$incidente.TIPO_INCIDENTE",
                        GRAVEDAD: "$$incidente.GRAVEDAD",
                        FECHA_REPORTE: "$$incidente.FECHA_REPORTE"
                    }
                }
            },
                  RegistroClima: { $map: { input: "$RegistrosClimaID", as:
"clima", in: "$$clima.ID" }},
                                EncuestaSatisfaccion: { $map: { input:
"$EncuestaSatisfaccionID", as: "encuesta", in: "$$encuesta.ID" }}
        }
    },
    {
        $merge: {
            into: { db: "bd_temporal", coll: "Areas_con_validacion" },
            whenMatched: "replace",
            whenNotMatched: "insert"
        }
    }
]);
```

La primera transformación que necesitamos hacer en esta migración es en el campo de cantidadJuegosPorTipo. Aunque durante la limpieza de python este campo lo creamos como un diccionario, al pasarlo a un csv se guarda como un string con el siguiente formato: "{'deportivas': 6, 'mayores': 0, 'infantiles': 0}" . Para extraer los datos de este string decidimos hacer fue transformar la cadena en un objeto JSON válido porque ya tiene el formato del JSON, pero sustituyendo las comillas simples por comillas dobles.

Para hacer este cambio primero creamos el nuevo campo de cantidadJuegosPorTipoObj usando addFields. Aquí definimos una función que recibe el string, sustituye las comillas simples por comillas dobles adaptando la cadena al formato JSON y por último utiliza la función JSON.parse() para convertir la cadena a un JSON. Este JSON se guarda en el campo cantidadJuegosPorTipoObj. Más adelante en la migración, para obtener el número de cada tipo de juego se accede de la siguiente forma: cantidadJuegosPorTipoObj.tipo, siendo tipo cualquiera de los tres definidos. Este valor se pasa a entero en la migración para que coincida con la validación.

En el caso de área tuvimos el mismo problema que en juego para hacer las referencias ya que en ID del área en el dataset original era un entero. Por ello, también hicimos dos *project*. En el primero convertimos el ID y el código postal a string y la fecha al formato de date de Mongo. El resto de campos no necesitaban transformaciones.

Con estas transformaciones ya podemos hacer las operaciones de lookup porque el campo de ID ya se ha convertido a string.

Se realiza un lookup para que en JuegosID (array) se almacenen todos los juegos que hay en ese área. Se meten en el array todos aquellos juegos cuyo AreaRecreativaID coincide con el ID del juego.

Se realiza otro un lookup para que en IncidenteSeguridadID (array) se almacenen todos los incidentes de ese área. Se meten en el array todos aquellos incidentes cuyo AreaRecreativaID coincide con el ID del juego.

Se realiza otro un lookup para que en EncuestaID (array) se almacenen todas las encuestas de satisfacción de ese área. Se meten en el array todas las encuestas cuyo AreaRecreativaID coincide con el ID del juego.

Por último, se realiza un lookup para que en RegistroClimaID (array) se almacenen los datos del clima que han afectado a ese área. Se meten en el array aquellos registros de clima cuyo código postal coincide con el del área. El problema en este caso es que el código postal que se almacena en el registro del clima puede tener uno de estos dos formatos: ‘28011’ o "28008, 28013". En el caso de que solo hubiera un código postal se podría realizar el *lookup* por código postal y obtener los documentos que coinciden. Sin embargo, como hay datos en el clima que tienen varios códigos postales esto no funcionaría.

Para tratar estas situaciones queríamos que se buscara el código del área dentro de la base de datos del clima sin importar si estaba solo o junto con otros códigos. Para hacer esto necesitamos usar un pipeline dentro en el *lookup*. En este pipeline

utilizamos *regexMatch* para definir una expresión regular que busca que el código del área esté en el campo de la base de clima, permitiendo que el código esté al principio, al final o en el medio de una lista separada por comas. De esta forma podemos obtener todos los documentos de clima que tienen el código postal del área independientemente de si tienen más códigos o no. Al principio tratamos de hacer esto sin usar un pipeline pero no podíamos hacerlo solo con *lookup* porque no perminte aplicar condiciones avanzadas directamente sobre el campo del registro del clima.

En este caso es necesario comprobar también la fecha del registro del clima. Por mucho que el clima afecte a todas las áreas de un código postal, a un área solo le han afectado los registros posteriores a su fecha de instalación. Todos los registros anteriores a esta fecha no han afectado al área porque no existía todavía. Esto lo hacemos también en el pipeline con el operador gt, comprobando que la fecha del registro del clima sea posterior a la fecha de instalación del área.

A continuación realizamos el segundo *project* donde utilizamos map para recorrer cada uno de los arrays e ir creando las referencias. En el caso de Juegos, RegistroClima y EncuestasSatisfacción, se crean arrays con los IDs de cada documento relacionado. En el caso de IncidenteSeguridad, se crea una referencia con resumen donde, además de incluir el ID del incidente, también se incluye el tipo del incidente, la gravedad y la fecha de reporte.

Finalmente se migran los datos a la colección de Areas_con_validacion de la base temporal. Esta colección después se migra a la base de datos con validaciones.

Diseño de Clusters

Estrategia de Replicación

La replicación se basa en la distribución de servidores Mongodb en diferentes nodos de la red en los que cada nodo tiene su propia copia de los datos. La estrategia de replicación asegura alta disponibilidad, redundancia y recuperación de los datos. La replicación en mongo permite tener varias copias de los datos distribuidas en diferentes nodos, garantizando que si el sistema falla por algún motivo, se pueda restablecer rápidamente la información utilizando otro nodo. En esta estrategia, hay que tener en cuenta el número y el tipo de nodos a utilizar.

Dado que el sistema almacenará datos críticos de áreas recreativas, juegos, y gestión de incidencias, es importante que los datos estén disponibles en todo momento, especialmente si se implementa en múltiples ciudades o países, ya que también hay

datos que requieren referencias a otros datos. Para los datos actuales, consideramos crear tres nodos: un nodo principal y dos secundarios. Opcionalmente se puede crear un nodo árbitro. Se despliega ese número de nodos porque como cada nodo almacena todos los datos, incluir muchos nodos podría superar la carga máxima de capacidad de almacenamiento. Las funciones de estos nodos son las siguientes:

Nodo primario: recibe las operaciones de escritura y replica los datos a los nodos secundarios. Solo puede haber un nodo primario y también puede recibir operaciones de lectura.

Nodos secundarios: se usan para realizar consultas de lecturas y garantizar la redundancia de los datos. Reciben los datos del nodo primario y están disponibles para operaciones de lectura y respaldo, pero no pueden realizar operaciones de escritura. Si el nodo primario falla, uno de los nodos secundarios puede convertirse en primario.

Nodo árbitro (opcional): no almacena una copia de datos como el resto de nodos secundarios, este actúa para ayudar a mantener quórum en elecciones de failover si hay una pérdida temporal del nodo primario. No añade costos de almacenamiento innecesarios.

En los clusters de replicación, cuando el nodo primario falla, se asigna un nuevo nodo primario. Para esto se realiza una estrategia Failover basado en el algoritmo RAFT. Todos los nodos del sistema están interconectados entre sí y se envían señales de actividad. En el momento en el que un nodo deja de recibir una señal de actividad del nodo primario en un determinado intervalo de tiempo este nodo pasa a ser un nodo candidato. En ese momento se realiza un proceso de votación en el que cada nodo envía al resto su candidatura para convertirse en líder. Después de este proceso se escoge el nuevo líder en función del nodo que tenga el término más alto y los datos más actualizados. En este proceso de votación para el caso de empate se utiliza al nodo árbitro. Cuando un nodo se convierte en líder se aumenta el término mencionado en uno, dejando claro que los nodos con término inferior a este son seguidores.

En un caso de implementación real, es importante la sincronización de los nodos secundarios para que reciban rápidamente las actualizaciones del nodo primario y minimizar la cantidad de datos que deben aplicar si se convierten en primario.

Los cluster de replicación también permiten distribuir la carga de lectura de los nodos. Como el despliegue es a nivel de ciudad, un cliente accede al nodo más cercano, reduciendo la latencia de lectura y la velocidad de respuesta, así como la sobrecarga del nodo primario.

La ventaja de esta implementación es que el sistema siga funcionando incluso si un nodo falla, ya que los otros nodos mantienen la información y se asignan como primario con un proceso de votación de manera eficiente. Además, las lecturas se distribuyen entre los nodos, asignando por proximidad las lecturas a los nodos más cercanos al cliente.

Para el caso de ampliar el problema, para por ejemplo un despliegue a nivel internacional, se necesitarán más nodos, ya que la distribución espacial será más amplia. La estrategia de despliegue podría constar de cinco nodos, uno primario y cuatro secundarios, además de un nodo árbitro para ayudar en el proceso de votación.

Estrategia de Fragmentación

Un cluster de fragmentación de mongo permite distribuir los datos en nodos como resultado de la fragmentación del sistema de información, mejorando la escalabilidad del sistema. Esto también deriva en otros beneficios como la localidad de la información. Para diseñar el cluster de fragmentación nos hemos basado en los casos de uso de forma que se optimicen las consultas más frecuentes.

Para el caso del conjunto de datos de Juegos, este se fragmenta por la clave distrito en el caso del problema actual. Si se decide ampliar a un nivel de datos internacional habría que fragmentarlo a nivel de ciudad o país, ya que sino se produciría un número muy elevado de fragmentaciones. Los datos se fragmentan de forma que los Juegos están en el distrito al que pertenecen, lo cual permitiría un rápido acceso a los juegos a nivel local (distrito), que seguramente sea la acción más utilizada (CU_1).

El conjunto de datos de mantenimiento será fragmentado por juegoID. Esta columna de los mantenimientos tiene el Id del juego al que pertenecen. De esta forma, todos los mantenimientos del mismo juego estarán agrupados en el mismo nodo. Esto facilita el acceso a los mantenimientos específicos realizados a cada juego y permite realizar consultas a dichos mantenimientos sin necesidad de revisar fragmentos no relacionados (CU_1).

El dataset de Registro climático será fragmentado por código postal asociado al área recreativa si se trata de un despliegue a nivel de una ciudad. Para un despliegue más amplio se podría fragmentar a nivel de ciudad. Esta fragmentación facilita el análisis local del impacto del clima en el desgaste de los juegos en áreas específicas y permite priorizar el mantenimiento en zonas afectadas por condiciones climáticas adversas (CU_3). Si se implementara a nivel internacional, hacerlo a nivel de código postal sería muy costoso ya que habría muchas fragmentaciones en nodos y esto requeriría mucho coste de almacenamiento. Esta estrategia soporta una rápida localización de datos climáticos según la ubicación.

Para los incidentes de seguridad, se ha decidido fragmentar por localización, en este caso por Areald. Al fragmentarse por areald permite recoger y almacenar los incidentes de seguridad reportados en las áreas recreativas de forma más eficiente (CU_4). De igual forma ocurre con las encuestas de satisfacción, estas se realizan para evaluar la experiencia de los usuarios en las áreas y con los juegos (CU_4). Por esta razón las encuestas de satisfacción también se fragmentarán por Areald, agrupando en cada área las encuestas y los incidentes, correlacionando los datos de seguridad y satisfacción para identificar zonas de riesgo o áreas que necesitan mejoras en accesibilidad o calidad (CU_4).

En el caso de las Áreas, se fragmentarán en función del distrito. Esto permite optimizar la generación de informes que detallan la frecuencia de mantenimiento realizada en cada distrito y ayuda a proporcionar una visión para la planificación estratégica de recursos en las áreas de cada distrito (CU_5). Si se ampliara mucho el volumen de los datos, por ejemplo a nivel europeo, esta fragmentación no sería eficiente porque se generaría muchos nodos. En este caso habría que considerar una fragmentación por ciudad o incluso por país. El problema que podría de esta fragmentación sería que hubiera nodos con muchos más datos que otros, pero aún así consideremos que es la forma más eficiente de fragmentar los datos en función de las consultas de los casos de uso.

En el caso de los Usuarios, tanto en un despliegue a nivel de ciudad como un despliegue a nivel internacional, se fragmentarán en nodos en función de su NIF, dividiéndolos en rangos para garantizar una distribución uniforme y equilibrada. Esto facilita el acceso a los datos de usuario cuando se necesita información de los ciudadanos que reportan incidencias o completan encuestas de satisfacción. Al distribuir los NIF en rangos se evita una distribución desigual de los datos, manteniendo un equilibrio entre los nodos.

Por último, respecto a las incidencias reportadas por los usuarios, estas se fragmentarán de acuerdo con su tiempo de resolución, agrupándolas en intervalos de tiempo independientemente de la amplitud del despliegue. Esto permite realizar consultas para generar informes de seguimiento de calidad en función de los tiempos de respuesta y resolución de las incidencias (CU_2).

Utilización de IA

Hemos utilizado inteligencia artificial para resolver errores de Mongo que no entendíamos. Había situaciones en las que pensábamos que el código de Mongo era

correcto pero devolvía errores que no entendíamos lo que eran, por lo que usamos inteligencia artificial para entender estos errores.

También usamos inteligencia artificial para tratar de hacer código más eficiente. Por ejemplo, en el proceso de limpiado de datos con python teníamos que calcular el área a la que pertenecía cada juego y el código inicial que hicimos era muy poco eficiente y tardaba mucho. Usamos inteligencia artificial para detectar si había formas más eficientes de hacerlo y descubrimos la librería que utilizamos finalmente.

Por último utilizamos IA para resolver dudas que teníamos con Mongo, sobre todo en el momento de transformar datos más complejos que al principio no sabíamos cómo hacer o que hacíamos pero no obteníamos el resultado esperado.