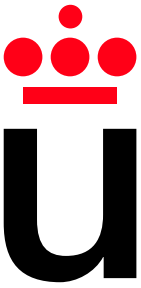

Universidad
Rey Juan Carlos

Escuela Técnica Superior
de Ingeniería Informática

Grado en Ingeniería Informática

Curso 2023-2024

Trabajo Fin de Grado

DISEÑO Y DESARROLLO DE BASES DE DATOS ORIENTADAS A DOCUMENTOS: CASO PRÁCTICO EN MONGODB

Autor: Álvaro Martín Martín
Tutora: Dolores Cuadra Fernández
Cotutor: Carlos Lara Romero

©2023 Álvaro Martín Martín

Este documento se distribuye bajo la licencia *Atribución-CompartirIgual 4.0 Internacional* de Creative Commons, disponible en: https://creativecommons.org/licenses/by-sa/4.0/deed.es

Agradecimientos

Gracias a todo aquel que en este camino me ha acompañado, en especial a mi familia por su apoyo incondicional.

Resumen

En el marco de la Estrategia Nacional de Conservación y Utilización de Parientes Silvestres de los Cultivos (PSC) y Plantas Silvestres de Uso Alimentario (PSUA), se dispone de información sobre el Inventario Nacional In Situ de PSC y PSUA organizada en 3 bloques principales: datos sobre las especies, poblaciones y registros de observaciones.

Además, se cuenta con un diseño conceptual que refleja la configuración de la base de datos de manera organizada por entidades y atributos. Este diseño será fundamental para el desarrollo informático posterior.

Se han implementado en R [1] diferentes estructuras de los datos a partir de los 3 datasets proporcionados, se ha cargado la información en la base de datos NoSQL MongoDB [2] y se ha comparado el rendimiento de cada una de ellas en términos de tiempo de lectura y escritura. También se ha tenido en cuenta el tamaño total resultante en la base de datos así como el tiempo de ejecución en definir cada una de las estructuras.

Cada estructura implementada es fiel al modelo conceptual proporcionado y se almacenará en su colección o colecciones correspondientes en MongoDB. En algunos casos se ha modificado la naturaleza de los datos de determinados atributos con el objetivo de aumentar la funcionalidad de la base de datos. También se ha tenido que lidiar con dificultades y proporcionar soluciones a problemas para conseguir que cada una de las estructuras funcionen de la mejor manera posible.

Los tiempos de lectura se medirán haciendo consultas a la base de datos basadas en una lista de consultas frecuentes proporcionada en la que se distinguen diferentes tipos de consulta.

Además de concluir cuál es la mejor estructura, se proporcionarán los scripts desarrollados, un documento con las consultas que se han utilizado para medir los tiempos de lectura y una guía para generar y cargar las estructuras en la base de datos.

Palabras clave:

- MongoDB
- R
- Dataframe
- Dataset
- Estructura
- Python
- Script
- NoSQL
- MongoDB Atlas
- Query
- Consulta
- Base de datos
- PSC y PSUA

Índice de contenidos

Índice de tablas **XI**

Índice de figuras **XIII**

Índice de códigos **XV**

1. Introducción **1**
1.1. Contexto y alcance 4
1.2. Estructura del documento 4

2. Objetivos **5**
2.1. Objetivo principal 5
2.1.1. Estudio y comprensión del contexto y los datos 5
2.1.2. Análisis de diferentes opciones de diseño 6
2.1.3. Modelado de cada una de las opciones 6
2.1.4. Inserción de datos a MongoDB 6
2.1.5. Evaluación de rendimiento y conclusiones 6

3. Desarrollo del proyecto **8**
3.1. Primera iteración 8
3.1.1. Entendimiento del contexto 8
3.1.1.1. R 8
3.1.1.2. MongoDB 9
3.1.1.3. Naturaleza y contexto de la base de datos 10
3.1.2. Entendimiento de los datos 11
3.1.3. Preparación de los datos 13
3.1.3.1. Identificación y refinamiento de datos potencialmente mejorables 13
3.1.3.2. Dificultades encontradas 15
3.1.4. Modelado 16
3.1.4.1. Definición de las diferentes estructuras 17
3.1.4.2. Evolución en la creación de las estructuras 20
3.1.5. Evaluación 25

3.2. Segunda iteración . . . 26
3.2.1. Entendimiento del contexto . . . 27
3.2.2. Modelado . . . 27
3.2.2.1. Solución 1 (Relaciones 1:N) . . . 28
3.2.2.2. Solución 2 (Relaciones 1:1) . . . 30
3.2.3. Evaluación . . . 33
3.3. Tercera iteración . . . 35
3.3.1. Entendimiento del contexto . . . 35
3.3.2. Modelado . . . 36
3.3.2.1. Solución al límite de 16 MB por documento en MongoDB . . . 36
3.3.2.2. Script en Python . . . 37
3.3.2.3. Atributo _row creado inesperadamente . . . 39
3.3.2.4. Conversión de dataframe a JSON . . . 41
3.3.2.5. Modelado de las consultas . . . 42
3.3.3. Evaluación . . . 43
3.3.3.1. Evaluación de los tiempos en generar y cargar las estructuras a MongoDB . . . 44
3.3.3.2. Evaluación del espacio resultante en la base de datos 46
3.3.3.3. Evaluación de las consultas a la base de datos . . . 47
3.3.4. Despliegue . . . 52

4. Conclusiones y trabajos futuros **55**

Bibliografía **58**

Apéndices **60**

A. Adjuntos a este documento **61**
A.1. Scripts de R implementados . . . 61
A.2. Script de Python implementado . . . 61
A.3. Listado de consultas a la base de datos . . . 61

B. Guía para la configuración y carga de información a MongoDB **62**
B.1. Descargar y preparar los adjuntos . . . 62
B.2. Preparar el entorno . . . 62
B.3. Ejecutar . . . 63

Índice de tablas

3.1. Cálculo de la relación media entre pruebas 26
3.2. Comparación de tamaños de dataframe y JSON para las dos soluciones implementando relaciones 1:1 en la estructura Registro_TaxonPoblacion. 33
3.3. Tiempos totales de ejecución para diferentes estructuras 45
3.4. Espacio de almacenamiento de las diferentes colecciones en MongoDB. 46

Índice de figuras

1.1. Procedimiento iterativo para la determinación de la estructura de la base de datos . . . 2
1.2. Metodología Ciencia de Datos . . . 3

3.1. Diseño conceptual de la base de datos . . . 11
3.2. Estructura Taxon_Poblaciones_Registros . . . 17
3.3. Estructura Poblacion_TaxonRegistros . . . 18
3.4. Estructura Registro_TaxonPoblacion . . . 18
3.5. Estructura ColeccionesSeparadas . . . 19
3.6. Tiempos en insertar los registros en las primeras N poblaciones . . . 25
3.7. Ejemplo visual correspondiente a la primera línea de código de la solución final (1:N) . . . 29
3.8. Ejemplo visual correspondiente a la segunda línea de código de la solución final (1:N) . . . 29
3.9. Ejemplo visual del funcionamiento del comando match entre 2 dataframes . . . 31
3.10. Ejemplo visual que muestra de manera simplificada la estructura resultante del dataframe RegistroDeOcurrencia . . . 32
3.11. Ejemplo de código en JSON de un documento simplificado con los atributos _row generados . . . 40
3.12. Comparación de tiempos en función de el número de cores y división de los datos . . . 42
3.13. Medición de tiempos para las diferentes tareas de cada estructura 44
3.14. Tiempos de respuesta ante las consultas a nivel de taxón . . . 48
3.15. Tiempos de respuesta ante las consultas a nivel de población . . . 49
3.16. Tiempos de respuesta ante las consultas a nivel de registro de ocurrencia . . . 50
3.17. Referencias entre colecciones . . . 51

Índice de códigos

3.1. Código para aplicar la función ConvertirEnLista a todos los elementos de la columna . . . 15
3.2. Código para aplicar la función ConvertirEnLista a los elementos que no son NA ni cadena vacía . . . 16
3.3. Pseudocódigo para asignar las poblaciones a un taxón. . . . 22
3.4. Pseudocódigo para asignar las poblaciones a un taxón usando la función de R *split* . . . 24
3.5. Código para asignar los registros de ocurrencia a las poblaciones (1:N) . . . 28
3.6. Ejemplo (erróneo) de asignación de un dataframe a una celda específica . . . 30
3.7. Código para asignar las poblaciones a los registros de ocurrencia (1:1) . . . 31
3.8. Ejemplo de URL de conexión a MongoDB . . . 34
3.9. Código para enviar la información en JSON al servidor en escucha 38
3.10. Consulta en MongoDB para recuperar todas las poblaciones asociadas a una especie . . . 52

1 Introducción

Este trabajo de fin de grado se centra en la creación de una base de datos que almacenará información sobre el Inventario Nacional de Parientes Silvestres de los Cultivos (PSC) y Plantas Silvestres de Uso Alimentario (PSUA) de España. La estructura de la base de datos se generará a partir de un script de R implementado, barajando diferentes posibilidades sobre la organización de los datos. Se propone como objetivo generarla en el menor tiempo posible, preparando los datos y relacionándolos de tal manera que ocupen el menor espacio en la base de datos de MongoDB, donde se alojará la información. Esta base de datos debe ser rápida y eficiente, para ello se tendrá en cuenta los tiempos en obtener respuesta ante determinadas consultas. Será de vital importancia también evitar la inclusión de información redundante en la base de datos.

Para el diseño de la base de datos, han sido proporcionados 3 datasets con información correspondiente a 3 áreas. Por un lado, un dataset con la información sobre las 521 especies prioritarias de PSC y PSUA de España. Por otro lado, la información correspondiente a los registros de ocurrencia de dichas especies, en total hay 1806711 observaciones. Y por último, la información correspondiente a las poblaciones. Una población es un conjunto de registros de ocurrencia de una misma especie en un área determinada. En total se ha proporcionado información sobre 611611 poblaciones.

La estructura general de la base de datos está definida en base a un proceso iterativo previo a la realización de este proyecto. En este proceso se han analizado los estándares europeos e internacionales para la recopilación de los datos, se han seleccionado fuentes externas, entidades y características relevantes a la par que se ha procedido al filtrado, limpieza y transformación de los datos. En base a esto,

se ha diseñado la organización de los datos, es decir, el conjunto de entidades y atributos que la forman, así como las relaciones entre ellas. La Figura 1.1 representa el proceso iterativo empleado:

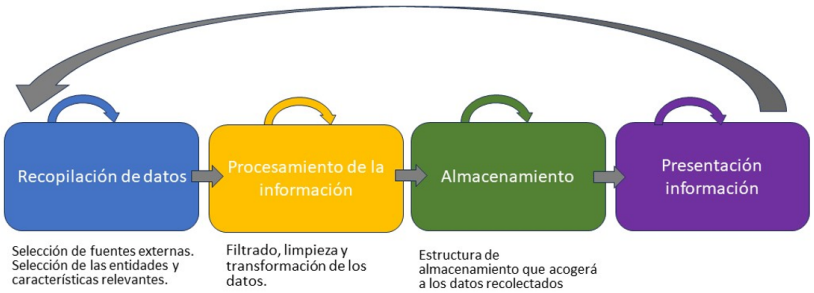

Figura 1.1: Procedimiento iterativo para la determinación de la estructura de la base de datos

Esto es un procedimiento previo a la realización de este proyecto. En otras palabras, para completar este trabajo, la estructura de la base de datos y su información han sido proporcionados de antemano. Destacar también que la recopilación de los datos que se manejarán y almacenarán está basada en el siguiente trabajo: [3].

Para el diseño de la base de datos, se pueden implementar diferentes tipos de organizaciones. Estas organizaciones representarán la manera en la que estarán estructuradas cada una de las entradas a la base de datos. Cada una de ellas se implementarán en R, estableciendo el objetivo de optimizar el tiempo en su configuración y la minimización del espacio resultante en la base de datos. Es importante también comprender la naturaleza de los datos y sus características ya que, de esta manera, se tendrán conocimientos para estructurarlos de tal forma que aumente la funcionalidad de la base de datos. Para esto, se puso a mi disposición un documento de apoyo en el que se proporciona una descripción sobre cada uno de los datos que alberga la base de datos.

Tras la implementación de las diferentes organizaciones en R, se debe cargar la información en MongoDB, gestor de bases de datos elegido para almacenar los datos. Debido a las peculiaridades de cada organización de los datos, habrá organizaciones que ocuparán más espacio que otras, otras incluso que será necesario ajustarlas para que se pueda almacenar toda su información. Una vez los datos estén cargados, es importante valorar el tiempo empleado en cargarlas así como el espacio de almacenamiento empleado. También se realizarán consultas basadas en una lista de consultas frecuentes proporcionada, donde se distinguen diferentes tipos de consultas. Se medirán los tiempos empleados en obtener respuesta a dichas consultas. Este es, sin duda, el tiempo más importante en comparación con otros tiempos que se puedan evaluar (generación de estructuras, carga de datos a MongoDB, etc.) ya que las consultas se realizan diariamente, mientras que el

resto de tareas solo se ejecutan una vez.

Tras la implementación y evaluación de los aspectos relevantes para la investigación, se proporcionará una conclusión sobre cual de todas las organizaciones es la que mejores resultados ofrece. Será esta la encargada de definir la forma en la que los datos sobre el Inventario Nacional In Situ de PSC y PSUA se almacenarán en la base de datos.

La metodología empleada durante el desarrollo del proyecto es la metodología de Ciencia de Datos , sin duda se ha adaptado muy bien al desarrollo ya que su proceso iterativo ha facilitado la forma en la que se recopila información, modela y evalúa antes de ofrecer la solución al problema. A continuación se presenta una imagen que muestra cada una de las etapas de dicha metodología:

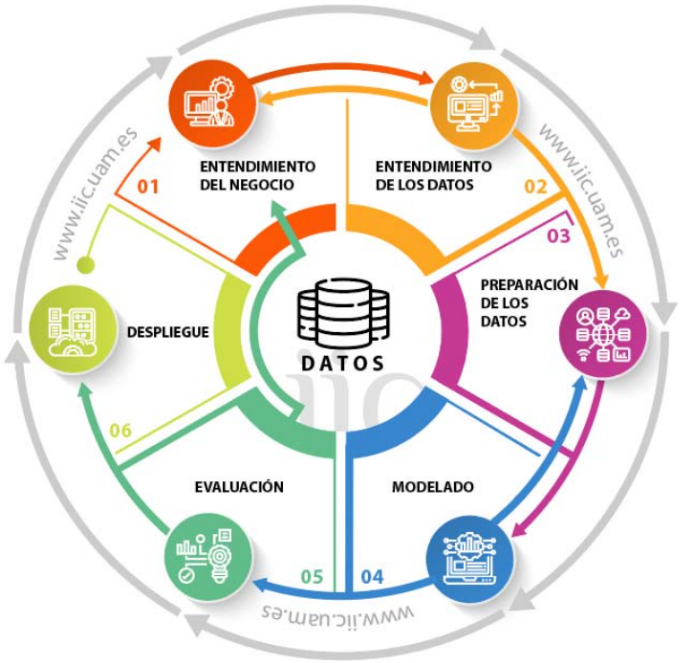

Figura 1.2: Metodología Ciencia de Datos

Al ser un proceso iterativo, se han realizado tantas iteraciones como se han necesitado. En cada iteración, ha sido necesario prescindir de alguna etapa en función del rumbo que ha ido tomando el desarrollo del trabajo de fin de grado. A pesar de que la primera etapa se nombra como *Entendimiento del negocio*, las particularidades del proyecto han llevado su modificación a *Entendimiento del contexto*, etapa dirigida a la recopilación de información e investigación acerca de asuntos y tecnologías que se tratarán en dicha iteración. La etapa de *Despliegue* solo se tratará en la última iteración, ya que es en esta cuando el diseño estará concluido y las evaluaciones hayan permitido ofrecer una conclusión sobre cual de las organizaciones de los datos (estructuras) es la mejor.

1.1. Contexto y alcance

El propósito de este trabajo de fin de grado es el desarrollo de una base de datos para el Inventario Nacional de Parientes Silvestres de los Cultivos (PSC) y Plantas Silvestres de Uso Alimentario (PSUA) de España, siguiendo los principios FAIR (Localizable, Accesible, Interoperable y Reutilizable) [4]. Estos principios son un conjunto de guías para mejorar la gestión de los datos y se han tratado de seguir lo mejor posible, dentro del marco en el que soy capaz de actuar, considerando los recursos y capacidades disponibles.

La base de datos debe servir para gestionar la conservación in situ de los recursos fitogenéticos en España y potenciar su uso. Su implementación aporta valor a la Estrategia Nacional de Conservación y Utilización de Parientes Silvestres de los Cultivos y Plantas Silvestres de uso Alimentario [5], cuyos objetivos principales son mejorar el conocimiento sobre los PSC/PSUA e inventariar los PSC/PSUA del Catálogo. Con esta estrategia, se pretende que España contribuya información a la base de datos EURISCO [6], que recoge la información sobre los recursos fitogénicos en Europa y a la base de datos del Sistema Mundial de Información sobre Recursos Fitogenéticos (GLIS) de la Organización de las Naciones Unidas para la Alimentación y la Agricultura (FAO). [7].

Este proyecto concreto forma parte de la Estrategia Nacional comentada anteriormente, en la cual están involucradas una gran cantidad de investigadores y técnicos, así como organismos como el Ministerio de Agricultura, Pesca y Alimentación [8] y el European Cooperative Programme for Plant Genetic Resources (ECPGR)[9].

Una de las posibilidades barajadas sobre el futuro del proyecto es la publicación de estos datos a través una interfaz web, en concreto en un servidor alojado en el Ministerio de Agricultura, Pesca y Alimentación o en la Universidad Rey Juan Carlos. Esto facilitaría el acceso a los datos, ofreciendo un servicio de consultas adaptados a los usuarios. También se podrían implementar formas para introducir datos a la base de datos desde esta interfaz. De esta manera, no será necesario generar un nuevo script o modificar el que ya se dispone.

1.2. Estructura del documento

El documento está estructurado en base a la plantilla sugerida que proporciona la ETSII [10]. En primer lugar se dispone de una introducción y una enumeración de los objetivos. El apartado de *Desarrollo del proyecto* está organizado en función de las distintas iteraciones que se han llevado a cabo siguiendo la metodología de Ciencia de Datos. Por último, se exponen las conclusiones obtenidas y se reservan otras dos secciones para referencias a datos bibliográficos y apéndices.

2

Objetivos

El trabajo fin de grado persigue un objetivo único, el cual puede descomponerse en varias tareas específicas que, en su conjunto permitirán alcanzar el resultado deseado.

2.1. Objetivo principal

Se tratará de realizar todo el ciclo de vida de desarrollo de una base de datos óptima ya que es importante que se construya y funcione de la mejor manera posible. Entre las tareas relacionadas con su análisis, diseño e implementación destacan las siguientes:

2.1.1. Estudio y comprensión del contexto y los datos

Comprender el contexto en el que se está desarrollando el trabajo, investigar sobre las tecnologías que se van a usar y familiarizarse con los datos proporcionados que se manejarán será de vital importancia como paso previo en cada etapa de cada iteración de la elaboración del proyecto. De esta forma se asentará una base sólida antes de modelar, que permitirá realizar el trabajo de manera más efectiva.

2.1.2. Análisis de diferentes opciones de diseño

La información que se dispone se puede organizar de diferentes maneras, siendo por supuesto fiel a la estructura de los datos que desde un principio está fijada. Al ser un desarrollo dirigido por consultas, las posibles organizaciones de los datos a implementar estarán basadas en una lista de consultas frecuentes proporcionada.

2.1.3. Modelado de cada una de las opciones

Tras seleccionar las posibles organizaciones de los datos, será necesario implementar los diferentes scripts en R que, tras su ejecución, fijarán la forma en la que la información dentro de cada entrada a la base de datos estará ordenada. Se tendrá en cuenta el tiempo invertido en la generación de cada una de las organizaciones de los datos.

2.1.4. Inserción de datos a MongoDB

La información sobre los PSC/PSUA se almacenará en MongoDB. Será necesario realizar el proceso de carga de manera correcta, de modo que la base de datos no contenga información irrelevante. Se tendrá en cuenta el tiempo empleado en cargar la información a la base de datos.

2.1.5. Evaluación de rendimiento y conclusiones

Una vez se haya almacenado en MongoDB la información sobre cada una de las opciones, se deberá evaluar el rendimiento general de cada una de ellas. Para ello se pondrá sobre la balanza el tiempo empleado en generar las estructuras, el tiempo empleado en cargar la información a la base de datos, el espacio que ocupará en ella y la velocidad de respuesta ante determinadas consultas. Todo esto en conjunto permitirá proporcionar una conclusión sobre cual es la mejor configuración de los datos a implementar para almacenar la información

3

Desarrollo del proyecto

3.1. Primera iteración

En esta sección se detallarán las etapas que se han llevado a cabo durante la primera iteración de la metodología de Ciencia de Datos.

3.1.1. Entendimiento del contexto

Para desarrollar un buen trabajo es fundamental establecer las bases sobre las tecnologías que se van a utilizar así como entender la naturaleza y el contexto en el que se desarrollará el proyecto. Es por esto que antes de iniciar el desarrollo informático, me vi en la necesidad de adquirir un conocimiento sólido en R y MongoDB además de indagar en la documentación proporcionada para tener más claro el contexto en el que se está desarrollando el proyecto, su alcance y las características de la base de datos que se quiere implementar.

A continuación se desglosan las partes fundamentales que se exploraron y comprendieron en esta etapa.

3.1.1.1 R

Fue en mi primer año de carrera cuando entré en contacto con R, apenas recuerdo los contenidos y tareas de la asignatura de Estadística. Lo que si recuerdo es que utilizábamos R para crear gráficos, ajustar rectas de regresión y lo más

significativo para mi actual proyecto, manipular dataframes.

Si bien esto supone no adentrarme en una tecnología completamente desconocida para mí, ha sido necesario refrescar conceptos además de entender y considerar las particularidades de mi proyecto para centrarme así en el manejo de dataframes, dejando de lado aspectos como el análisis estadístico.

Mi formación en R se ha basado principalmente en la exploración de recursos en internet. En un primer lugar, como toma de contacto viendo vídeos introductorios, donde te enseñan las bases de la programación en R, el manejo de dataframes y operaciones básicas como bucles.

Hay que destacar que R es un lenguaje ampliamente utilizado que cuenta con una comunidad activa. Gracias a esto es posible encontrar soluciones a problemas que otras personas han resuelto con anterioridad además de una gran cantidad de recursos aparte de los que la documentación oficial [11] aporta.

A pesar de que al principio se asentaron las bases y conceptos fundamentales de R, a medida que avanzaba el proyecto fue necesario seguir formándose, ya que se emplearán librerías que en un principio no se tuvieron en cuenta.

3.1.1.2 MongoDB

A diferencia de R, MongoDB es una tecnología con la que no había trabajado anteriormente. De hecho, no tenía experiencia previa con bases de datos no relacionales, ya que únicamente había utilizado bases de datos relacionales como PostgreSQL, SQLServer o H2 en aplicaciones web.

Si bien no tuve que elegir MongoDB como sistema de gestión de bases de datos debido a que el proyecto demandaba su uso, entendí que era necesario y adecuado en cuanto me puse a investigar acerca de las características de las bases de datos no relacionales. Dada la naturaleza de los datos que estamos tratando así como la propia naturaleza de la base de datos final, se busca un diseño flexible, capaz de adaptar nueva información sobre el Inventario Nacional In Situ de PSC y PSUA ya que la metodología que se usa para completar la información de la base de datos es iterativa. Es decir, es probable que en un futuro se introduzcan nuevas características y atributos que amplíen la información de los documentos de la base de datos. Por lo tanto, MongoDB es una buena opción dada su alta flexibilidad y escalabilidad. A modo de aclaración, un documento en MongoDB representa la unidad básica de almacenamiento, cada entrada a la base datos. Además, cada uno de estos documentos, dentro de una misma colección puede tener una estructura distinta.

De la misma forma que con R, antes de adentrarme en el desarrollo de los scripts para establecer las distintas estructuras de la base de datos, decidí informarme e investigar en internet sobre MongoDB. No tardé mucho tiempo en

darme cuenta de que esta tecnología cuenta con una comunidad activa así como una gran cantidad de recursos oficiales [12] para aprender.

Sobre MongoDB aprendí bastante y adquirí una base sólida de su funcionamiento. Desde cómo crear colecciones o insertar datos hasta realizar consultas avanzadas usando su lenguaje de consulta MQL(Mongo Query Language) [13]. Esto último me resultó de gran valor ya que una de las partes del proyecto consiste en hacer consultas a la base de datos en base a una lista de consultas frecuentes y comparar los tiempos de respuesta entre las diferentes estructuras implementadas. También dediqué tiempo a evaluar las ventajas y desventajas sobre los diferentes tipos de esquemas que se pueden implementar en la base de datos.

Por último, destacar que aprendí a manejar herramientas afines a MongoDB, como son MongoDB Atlas [14], un servicio para implementar y administrar la base de datos, así como MongoDB Compass [15] como interfaz gráfica de usuario que facilita la exploración del contenido de la base de datos y permite realizar consultas gracias a su Shell integrada MongoDB Shell (mongsh) [16].

3.1.1.3 Naturaleza y contexto de la base de datos

Entender el contexto en el que se está desarrollando el proyecto, las características de los datos que se van a tratar y las especificaciones de la base de datos es fundamental como paso preliminar al desarrollo informático. Esto ha sido posible gracias a un documento de apoyo proporcionado en los inicios del proyecto que facilita la comprensión de todos estos aspectos. Este documento no es público.

El documento detalla la metodología que se ha seguido para recopilar, filtrar y transformar los datos que posteriormente haré uso y completarán la información de la base de datos así como su procedencia. También se habla de los estándares europeos e internacionales que se han seguido como punto de partida para definir su estructura. Se presenta también el diseño conceptual de la base de datos, donde se describen las diferentes entidades y sus respectivos atributos, así como los dominios en los que están definidas y las relaciones entre ellas.

También se dejan claros los objetivos y alcance del proyecto, proporcionándome una visión de la magnitud del mismo, así como los diferentes organismos y personas involucradas en su ejecución. Se plantea la idea de desarrollar una interfaz web para ofrecer un servicio de consultas a los usuarios.

En resumen, el documento de apoyo proporcionado contiene toda la información relativa al proyecto que se está desarrollando y me ha sido de gran ayuda para llevar a cabo la implementación informática, asegurando que mi trabajo esté alineado con las expectativas del proyecto.

3.1.2. Entendimiento de los datos

Tal y como se comentaba anteriormente, junto a los datasets proporcionados en los cuales albergan todos los datos que se van a manejar, se proporciona un documento de apoyo cuyo objetivo es proporcionar una visión global del proyecto, con la finalidad de comprender los objetivos y alcance del mismo. En este documento se habla sobre los diferentes tipos de datos que se almacenan, organizándolos en tres áreas donde en cada una de ellas residen entidades que a su vez tienen atributos. Las entidades también presentan relaciones entre ellas.

La organización general de la base de datos se muestra a continuación:

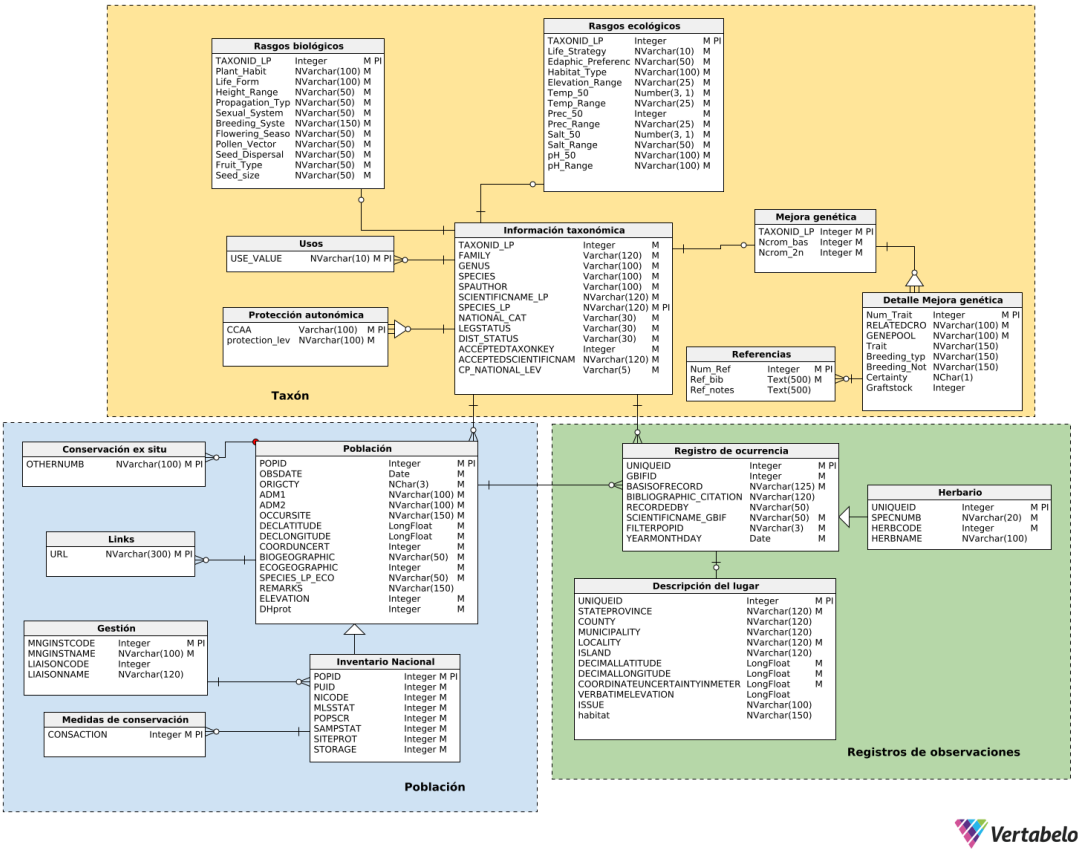

Figura 3.1: Diseño conceptual de la base de datos

Tal y como se aprecia en la Figura 3.1, se distinguen tres áreas de información a los que me referiré a ellos como áreas. Por un lado, de color amarillo se distingue entre la información relativa a las especies del catálogo Nacional de PSC y PSUA. Por otro lado, de color azul se presenta la información relativa a las poblaciones, una población consta de varios registros de ocurrencias asociados a una misma especie en un área determinada. Por último, de color verde se muestra

la información correspondiente a los registros de ocurrencia. Un registro de ocurrencia es una observación documentada de una especie en un lugar y momento específico. Este diseño conceptual ha sido modelado con la herramienta Vertabelo [17] y proporcionado de manera previa al desarrollo del trabajo de fin de grado.

La información que se almacenará en la base de datos ha sido proporcionada en tres datasets distintos, cada uno se corresponde con un área diferente. Un dataset es el conjunto de datos que se van a manipular. En mi caso, fue proporcionado en formato *.r*, aunque también se pueden manejar en R datasets con formato *.csv*. Tenemos el dataset *20231220_PSC_taxon*, correspondiente al área *Taxón* (amarillo), el dataset *20231220_PSC_poblacion* para el área *Población* (azul) y el dataset *20231220_PSC_registro* para el área *Registro de observaciones* (verde).

Dentro de cada una de las áreas hay diferentes entidades, las cuales a su vez tienen atributos. Destacar que no se tiene información como tal de las entidades en los datasets proporcionados, se tiene información de cada uno de sus atributos que, en su conjunto dan significado a la entidad. Por ejemplo, el dataset correspondiente al área de Registro de observaciones contiene la información correspondiente a los atributos *UNIQUEID*, *SPECNUMB*, *HERBCODE* y *HERBNAME* que, agrupándolos dan como resultado la entidad *Herbario*.

Agrupando los atributos para formar cada una de las entidades que aparecen en el diseño conceptual, se aprecia que hay una entidad principal por cada una de las áreas. Para el área *Taxón*, la entidad principal es *Información taxonómica* para el área *Población* es *Población* y para el área *Registro de observaciones* es *Registro de ocurrencia*. Estas entidades principales son las encargadas de representar el área en el que se encuentran y establecer el punto de entrada para el resto de entidades principales del resto de áreas.

Dada la información anteriormente nombrada, podemos establecer el significado de *estructura* como la organización interna que tendrá cada uno de los documentos de la base de datos, dicha estructura estará modelada de manera general, teniendo en cuenta todos los datos de la base de datos. Cada una de las estructuras que se implementarán se cargarán a su colección o colecciones correspondientes en la base de datos. Por lo tanto, todos los documentos de una misma colección compartirán la estructura que les corresponda.

Una vez se ha comprendido la forma en la que los datos se organizan y relacionan, se pueden ampliar conocimientos gracias al documento de apoyo, donde se encuentra toda la información relativa a cada uno de los atributos que se contemplan. En él se proporciona una descripción útil sobre cada atributo de cada entidad, descripción que nos ayuda a comprenderlo. En algunos casos se incluyen también los posibles valores que pueden tomar los datos de estos atributos. Por ejemplo, el atributo *LEGSTATUS* puede tomar únicamente los valores *HD*, *NAT* y *SUB*.

Conocer los datos que estamos manejando es de vital importancia para la siguiente etapa de esta iteración ya que se realizarán diversos ajustes en base a su naturaleza y características. Además, para realizar consultas a la base de datos es primordial entender el significado de estos, ya que debemos estar completamente seguros de que lo que estamos consultando es realmente lo que queremos consultar.

3.1.3. Preparación de los datos

Una vez se han analizado cada una de las descripciones de los atributos de cada entidad de la base datos y se ha entendido lo que representa cada uno de ellos, se ha llegado a la conclusión de que es posible modificar la naturaleza y características de los mismos, con el objetivo de mejorar la forma en la que se representa la información y aumentar así la funcionalidad en cuanto a las operaciones que se puedan llevar a cabo en la base de datos. A continuación se explica el proceso de tratamiento y refinamiento de los datos de la base de datos.

3.1.3.1 Identificación y refinamiento de datos potencialmente mejorables

En primer lugar, comentar que las futuras estructuras que se implementarán harán uso de estos datos refinados de la misma manera. Es decir, todas las colecciones de MongoDB albergarán los datos que se han refinado en esta etapa y, por lo tanto, los datos de las diferentes colecciones serán de la misma naturaleza.

Si bien no todos los datos se han modificado, se han detectado atributos potencialmente mejorables a nivel funcionalidad, estética o relevancia.

Empezando por estos últimos, se entiende por atributo con poca relevancia a aquellos que no guardan información. En MongoDB los atributos que no guardan información directamente no se añaden a la base de datos, esto es un proceso que hace automáticamente MongoDB a la hora de insertar los datos pero solo lo hará si en R, el valor correspondiente del documento para ese atributo es "NA". Son muchos casos en los que se da que el valor asociado a un atributo en un determinado documento es la cadena vacía. Esto Mongo lo interpreta como información y por lo tanto añade esa información irrelevante a la base de datos. De nada nos sirve almacenar en la base de datos información que no aporta nada, lo único que se consigue es aumentar el tamaño final de la base de datos.

Ante este problema, en primera instancia no se ha hecho nada en R. Será mas adelante, al realizar el proceso de inserción a la base de datos cuando estos datos se tratarán, no insertándolos en la base de datos. Pero ante todo, es importante ser consciente de este aspecto y saber que este problema existe.

Siguiendo con el proceso de identificación de datos potencialmente mejorables, se han identificado datos estéticamente mejorables. Es el caso del atributo *Yearmonthday* de la entidad *Poblacion*. En muchos casos, los datos asociados a este atributo aparecen como cadena vacía pero son otros los que aparece la siguiente cadena: *YYYYMMDD*. Esta cadena representa al igual que la cadena vacía que este dato no ha sido introducido y por lo tanto no representa información. Aprovechando que en un futuro no se insertarán los datos que contengan la cadena vacía, se ha implementado una función que transforma todos los datos con la cadena *YYYYMMDD* en la cadena vacía. De esta manera, los datos que contengan esa cadena no se introducirán tampoco en la base de datos.

Por último, se han identificado atributos que contienen información la cual se puede modificar con el objetivo de aumentar la funcionalidad de la base de datos. Es en este caso donde más atributos con información de estas características se han encontrado. Específicamente se han refinado los datos correspondientes a los siguientes atributos: Para el área de Taxón se han refinado *USE_VALUE*, *LEGSTATUS*, *GENEPOOL* y *RELATEDCROP*. Para el área de Registro de ocurrencia se han refinado *ISSUE* y *RECORDEDBY* y para el área de Población únicamente *URL*.

Como se comentaba antes, el refinamiento de estos atributos tiene el objetivo de aumentar la funcionalidad de la base de datos. Explicando esto con más profundidad, todos estos atributos tienen algo en común, almacenan la información como una lista de elementos pero en forma de cadena de texto, donde cada elemento está separado por un punto y coma. Aumentar la funcionalidad supone transformar estas cadenas de elementos en una lista convencional de elementos, y así poder realizar otro tipo de operaciones más intuitivas y lógicas con la base de datos.

Por ejemplo, centrándonos en el atributo *USE_VALUE*, hay taxones que tienen ese valor a "CWR", otros a "WFP" y otros a "OTHER", hay algunos casos incluso que un taxón tiene varios de estos valores (separados por punto y coma). Tras el refinamiento de estos datos, la cadena de caracteres con los elementos separados por punto y coma pasará a ser un array y se podrán realizar consultas en la base de datos mucho más específicas. Un ejemplo sería recuperar los taxones que tengan más de 1 elemento de el campo *USE_VALUE*. Otra podría ser recuperar los taxones que tengan únicamente el elemento "CWR", ninguno más. Otra podría ser recuperar el número de elementos que tiene un determinado taxón en el campo *USE_VALUE*. Estas consultas no se podían hacer anteriormente de manera directa ya que la información que guardaba el atributo era una cadena (string). Teniendo ahora una lista, se podrán realizar este tipo de consultas.

La única manera de obtener este tipo de información sin realizar ningún tratamiento y refinado de datos es, sabiendo que en un futuro se tiene pensado implementar una aplicación web para manejar la información de la base de datos, obtener la información en forma de cadena (string) y realizar en back-end el

tratamiento para, dada una cadena con elementos separados por punto y coma, convertirla en lista. Al poder hacerse esto directamente en R y guardar la información correctamente en MongoDB, se ha optado por implementar esta solución ya que evita futuros problemas.

Para llevar a cabo la transformación se ha implementado la función *ConvertirEnLista*, la cual recibe el string que se quiere modificar y lo transforma a lista, de manera que el punto y coma delimita cada uno de los elementos de la lista.

Por último, dejar claro que los datos no han sido alterados a nivel de información, únicamente se ha aplicado una transformación en la forma en la que se almacenará la información en la base de datos.

3.1.3.2 Dificultades encontradas

En los atributos para los que se aplica la función *ConvertirEnLista*, no siempre se dispone de un dato para todas las entradas del dataset. Por ejemplo, para el atributo *RECORDEDBY*, el cual representa el nombre de la persona, personas, grupo u organización responsable de registrar el registro de ocurrencia, es posible que en el dataset no figure ningún dato para alguna entrada. Esto se debe simplemente a que no se dispone de dicha información. Ocurre con la mayoría de los atributos a refinar. Únicamente los atributos *USE_VALUE* y *LEGSTATUS* poseen información para todas y cada una de las entradas de su dataset correspondiente (Taxón).

Lo primero que intenté para aplicar la función a los atributos potencialmente mejorables a nivel de funcionalidad fue aplicarla a todas las entradas de ese atributo, sin distinguir los NAs o las cadenas vacías(“”). Hice uso del comando *lapply*, el cual sirve para aplicar una función a cada elemento de una lista. En mi caso le estoy aplicando la función a una columna de un dataframe de R que, técnicamente es una lista.

El código que en un primer lugar empleé para realizar esta operación es el siguiente:

```
1 RegistroDeOcurrencia$recordedBy <- lapply(
    RegistroDeOcurrencia$recordedBy, transform_to_list)
```

Código 3.1: Código para aplicar la función ConvertirEnLista a todos los elementos de la columna

Pero esto se puede llevar más allá planteándose lo siguiente: Si no todas las entradas tienen un dato asociado, ¿por qué no aplicar la función a aquellas entradas en las que se dispone de un dato para transformar?. Esto es una buena idea para reducir el tiempo de ejecución. Se ha implementado de la siguiente manera:

```
indices_validos <- which(!is.na(RegistroDeOcurrencia$recordedBy)
   & RegistroDeOcurrencia$recordedBy != "")
RegistroDeOcurrencia$recordedBy[indices_validos] <- lapply(
   RegistroDeOcurrencia$recordedBy[indices_validos],
   transform_to_list)
```

Código 3.2: Código para aplicar la función ConvertirEnLista a los elementos que no son NA ni cadena vacía

En un primer lugar se detectan los índices de aquellas filas que no son NA ni cadenas vacías para posteriormente aplicar la función a los elementos correspondientes a dichos índices. Esto mejora en cierto punto el tiempo que se demora en generar las entidades en comparación con el código anterior, pasando en el caso de la estructura *Registro_TaxonPoblación* de 1 minuto y 35 segundos a 57 segundos, una notable mejora.

Si bien esta es la solución final que se ha implementado, presenta el siguiente problema: tras transformar en lista alguno de los elementos de la columna asociada al atributo que se quiere refinar, el resto de elementos que no se han tratado porque son NAs o cadenas vacías han pasado a ser de tipo lista también. Esto es algo que en principio no tiene sentido ya que esos datos no se están tocando, pero tiene que ver con cómo R realiza la coerción de tipos en estos casos con el objetivo de que todos los datos de la columna sean consistentes, es decir, del mismo tipo.

En R, es imposible tener una columna de un dataframe que contenga múltiples tipos de datos diferentes. En este caso, R convierte los NAs de tipo *logical* a tipo *list*, y las cadenas vacías de tipo *character* a tipo *list*. En este caso R convierte todos los elementos al tipo más general que pueda abarcar todos los valores, asegurando así la consistencia dentro de la columna.

Todo esto supone un problema ya que cuando se insertan los datos a MongoDB, se insertan listas con un único valor donde ese valor es *None* en el caso de los NAs o “” en el caso de que la lista indeseada creada surja a partir de una cadena vacía.

Esto es algo que no se puede permitir, por lo que se buscarán soluciones alternativas en la siguiente iteración.

3.1.4. Modelado

Modelar es la parte de cada iteración en la que sin duda más horas he invertido. He tratado de perfeccionar al máximo cada una de las estructuras que se han implementado, buscando optimizar el espacio resultante en la base de datos dentro de lo que permite cada una de las soluciones. También me he esforzado en reducir

al máximo el tiempo que se demora en la creación de cada una de ellas. Considero que he hecho un buen trabajo ya que he logrado ambos objetivos en las etapas de modelado. A continuación se detalla todo lo relacionado con el modelado en esta iteración.

3.1.4.1 Definición de las diferentes estructuras

Las decisiones sobre los diferentes tipos de estructuras a implementar se han tomado en base a la lista de consultas frecuentes proporcionada. Es en esta lista donde se distinguen 3 tipos de consultas, a nivel de taxón, de población y de registro de ocurrencia. Por esto, se ha decidido implementar varias estructuras donde cada una guarda en primera instancia la información relativa al área al que pertenecen.

Con el objetivo de aclarar las diferencias entre los distintos tipos de estructuras que se han implementado, a continuación se expondrá un resumen de cada una de ellas, con una imagen explicativa en la que se aprecia la estructura interna de cada documento de la colección de MongoDB a la que corresponde dicha estructura.

- **Estructura Taxon_Poblaciones_Registros**
  - Estructura organizada por especie (taxón). Con esta organización, cada uno de los documentos que se aloja en la base de datos tendrán en primera instancia los atributos correspondientes a la información taxonómica. Seguidamente aparecerá una lista cuyos elementos contienen la información relativa a cada una de las poblaciones asociadas a esa especie y dentro de cada una de esas poblaciones se tendrá otra lista cuyos elementos son cada uno de los registros de ocurrencia asociados a dicha población.
    Esta estructura se almacenará en la colección *Taxon_Poblaciones_Registros* de la base de datos. Dicha colección posee 521 documentos.

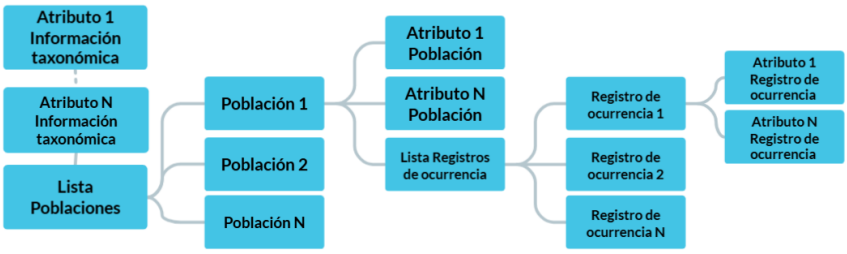

Figura 3.2: Estructura Taxon_Poblaciones_Registros

- **Estructura Poblacion_TaxonRegistros**
  - Estructura organizada por población. Con esta organización, cada uno de los documentos de la base de datos albergará en primera instancia

los atributos correspondientes a la entidad Población. Además, los documentos almacenarán la especie (taxón) asociada así como una lista con cada uno de los registros de ocurrencia de dicha población.

Esta estructura se almacenará en la colección *Poblacion_TaxonRegistros* de la base de datos. Dicha colección posee 611611 documentos.

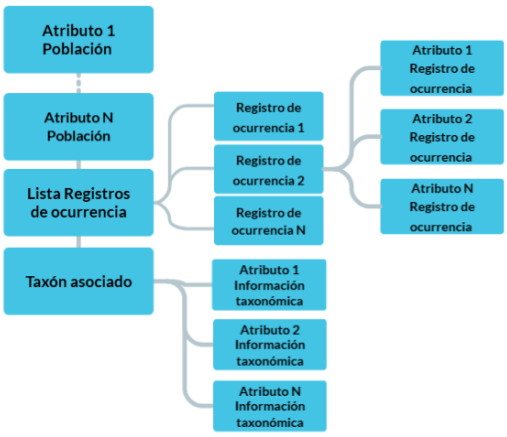

Figura 3.3: Estructura Poblacion_TaxonRegistros

- **Estructura Registro_TaxonPoblacion**

  - Estructura organizada por registro de ocurrencia. Cada uno de los documentos de la base de datos almacenará en primera instancia los atributos correspondientes a la entidad de Registro de ocurrencia. A su vez, se almacenará la información relativa al taxón y población asociada.

    Esta estructura se almacenará en la colección *Registro_TaxonPoblacion* de la base de datos. Dicha colección posee 1806711 documentos.

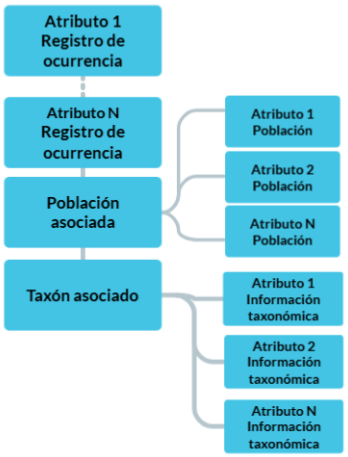

Figura 3.4: Estructura Registro_TaxonPoblacion

- **Estructura ColeccionesSeparadas**

  - En este caso, los documentos se almacenarán en tres colecciones distintas en base al área al que pertenezcan.

    El nombre de las colecciones donde se almacenarán los datos son *colSepInformacionTaxonomica*, que posee 521 documentos, *colSepPoblacion*, que posee 611611 y *colSepRegistroDeOcurrencia* que posee 1806711.

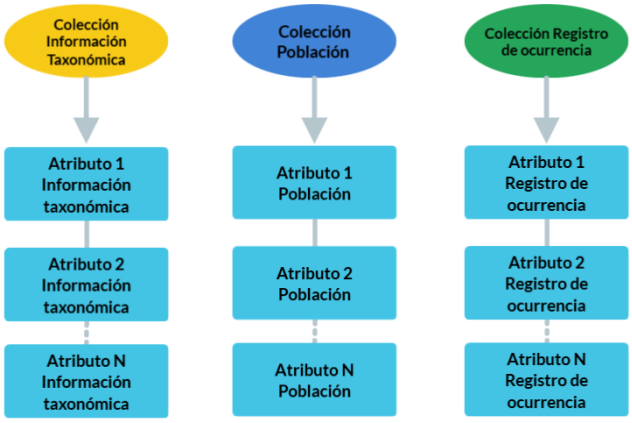

Figura 3.5: Estructura ColeccionesSeparadas

En cuanto a la nomenclatura que he utilizado para las estructuras, el guión bajo (_) representa el nivel de anidamiento de los datos. En otras palabras, la estructura *Taxon_Poblaciones_Registros* tiene dos guiones porque en los documentos asociados a esa colección, un taxón tiene un conjunto de poblaciones que a su vez, estas tienen un conjunto de registros. Por otro lado, la estructura *Poblacion_TaxonRegistros* solo tiene un guión porque una población tiene un taxón asociado así como un conjunto de registros de ocurrencia asociados.

Aparentemente la mejor estructura parece ser *Taxon_Poblaciones_Registros* ya que por cada taxón (en total hay 521) tenemos su información además de su lista de poblaciones que, a su vez, cada una de estas poblaciones tiene todos los registros de ocurrencia correspondientes. Parece ser una estructura ordenada y, en el fondo lo es. Junto con la estructura *ColeccionesSeparadas*, son las únicas que no presentan redundancia en los datos. Sin embargo, por tener una gran cantidad de información embebida en cada documento, en la base de datos nos encontraremos con documentos muy pesados, con una gran cantidad de información la cual nos costará más tratar y recuperar.

Con esto se concluye que hasta que no se realicen pruebas y validaciones de tiempo y espacio, no se podrá concluir con total certeza cuál es la mejor de las estructuras que se han implementado.

Por último, cabe destacar que se podría haber implementado también la colección Taxon_RegistrosPoblacion ya que el contexto lo permite. En esta colección se almacenarían documentos que guardan en primera instancia la información relativa a los taxones, seguido de su lista de registros de ocurrencia que a su vez cada uno de estos guarda su población correspondiente. No se ha implementado debido a que ya existe una colección en la que en primera instancia se guarda la información relativa a los taxones (*Taxon_Poblaciones_Registros*) y, comparando el rendimiento de ambas colecciones, en términos de espacio y redundancia de datos es peor. Se repetirían los datos de las poblaciones asociadas a los registros de ocurrencia, conllevando así a un aumento de tamaño en la base de datos.

3.1.4.2 Evolución en la creación de las estructuras

Uno de los principales objetivos a la hora de implementar las estructuras es tratar de minimizar al máximo el tiempo de ejecución en crearlas así como el espacio que ocuparán en la base de datos. Tratar de cumplir estos objetivos es lo que sin duda más tiempo me ha llevado.

Siendo más preciso en mi explicación, el tiempo de ejecución en crear una estructura se traduce como el tiempo que se demora en generar el dataframe en R que posteriormente se cargará a la base de datos. Este dataframe guardará los datos organizados en función de la estructura que se esté implementando.

En esta etapa de esta iteración se ha procedido a modelar cada uno de los diferentes scripts asociados a cada una de las estructuras que se van a implementar, lo que supone la etapa en la que más tiempo se ha invertido. Se han implementado de forma paralela ya que encontrar una solución para el modelado de un script, supone replicar esa solución en el resto, realizando pequeñas modificaciones en base a las peculiaridades específicas de cada estructura.

Aunque las diferentes estructuras implementadas organizan los datos de manera diferente, el punto clave es encontrar la manera óptima de embeber los datos dentro de otros. Poniendo un ejemplo, en la estructura *Taxon_Poblaciones_Registros* hay que intentar recopilar todas las poblaciones asociadas a cada uno de los taxones, y posteriormente todos los registros asociados a cada una de estas poblaciones. Puede parecer tarea fácil pero la dificultad se encuentra en realizar esto optimizando el tiempo y espacio invertido.

Comenzando con la explicación del desarrollo en la creación de estructuras, el primer paso es cargar cada uno de los datasets de información que se proporcionan. Como dije antes, cada uno corresponde con un área de información diferente (Taxón, Población y Registro de observaciones). Al estar estos datasets guardados en una computadora local, será necesario modificar los scripts de R para cambiar la ruta donde se encuentran. De esta manera, R podrá cargarlos correctamente. Esta modificación deberá hacerla la persona que vaya a ejecutar el script.

Una vez los datasets se hayan cargado a R, estos conjuntos de información tomarán el nombre de *dataframes*. Al fin y al cabo es la representación del dataset en R, donde las filas representan observaciones y las columnas los atributos.

Una vez hecho esto, se llevará a cabo el proceso de generación de entidades, en la cual se agruparán de forma adecuada los atributos de cada uno de los dataframes ya que como se comentaba anteriormente, los datasets guardan la información de todos los atributos de todas la entidades, no las entidades como tal. Además de generar las entidades, como pueden ser *Descripción de lugar*, *Inventario nacional* o *Rasgos ecológicos*, se han generado también las relaciones entre cada una de ellas, embebiendo la información sobre una única entidad que se situará en primer plano representando a cada área.

Destacar que muchos de los atributos que aparecen en el diseño conceptual, sobre todo en el área de Taxón, no están recogidos en su dataset correspondiente. Las entidades *Rasgos biológicos, Rasgos ecológicos, Mejora genética y Detalle mejora genética* no están completas. Además, la entidad *Referencias* ni siquiera se ha podido formar ya que ninguno de los atributos está contemplado en su dataset.

Este proceso de carga de la información y generación de entidades se lleva a cabo en todos los scripts implementados ya que la organización de las entidades son comunes en todas las estructuras. Fue algo con no mucha dificultad, bastó con aplicar los comandos adecuados para cargar información a R y realizar operaciones como agrupar o embeber datos utilizando dataframes. Esto estableció un punto de partida para relacionar las diferentes entidades principales de cada área, ya que en este momento las tenemos totalmente estructuradas, con las entidades secundarias correctamente integradas.

En este momento, tenemos 3 entidades principales, organizadas como dataframes. Por un lado tenemos la entidad *Información taxonómica*, la cual consta de 521 filas, cada una guarda la información de un taxón (especie). Por otro lado está la entidad *Población* que consta de 611611 filas, cada una guarda la información sobre una población (conjunto de registros de ocurrencia de la misma especie en un área reducida determinada). Por último contamos con la entidad *Registro de ocurrencia* con 1806711 filas, cada una correspondiente a una observación de una especie en un determinado momento. Como se puede apreciar, las tres entidades guardan una relación entre ellas. Para que quede más claro, a continuación se detallan cada una de las relaciones:

- Un taxón puede tener de 0 a N poblaciones. Sin embargo, una población pertenece únicamente a un taxón.
- Un taxón puede tener de 0 a N registros de ocurrencia. Sin embargo, un registro de ocurrencia pertenece únicamente a un taxón.
- Una población consta de 0 a N registros de ocurrencia. Sin embargo, un registro de ocurrencia pertenece únicamente a una población.

Estas relaciones se deben implementar en R y, durante gran parte del desarrollo he hecho uso de una versión reducida de los datasets proporcionados con el objetivo de que los tiempos en ejecutar haciendo pruebas sea mucho menor. Estas versiones las he ajustado yo, escogiendo 50 de 521 especies, así como un número poblaciones asociadas a estas especies y un número de registros asociados a estas poblaciones. Es decir, no he reducido aleatoriamente los datasets, escogiendo los 50 primeros elementos de cada uno de ellos ya que, de esa manera se puede dar el caso de estar cogiendo poblaciones que no están asociadas a ninguna especie así como registros que no están asociados a ninguna población. En resumen, los datasets fueron ajustados de tal manera que cualquier tipo de prueba tenga el suficiente contenido. El objetivo es que una vez que las pruebas funcionen para estos datasets reducidos, se puedan replicar en los datasets originales con toda la información.

Para establecer las relaciones entre las entidades principales lo primero que se me ocurrió fue lo que considero que se le ocurriría a cualquier persona que se enfrente a este problema. En el caso de relacionar los taxones con las poblaciones por ejemplo, pensé en recorrer cada una de las poblaciones y, en función del taxón al que pertenezca esa población (se sabe por el atributo *species_LP*), introducir esa población como parte de las poblaciones de ese taxón. Un ejemplo en pseudocódigo sería el siguiente:

```
for poblacion in dfPoblacion:
    # Obtener el nombre de la especie asociada a la poblacion
       actual
    especie_actual = poblacion.species_LP
    # Asignar la poblacion actual al conjunto de poblaciones del
       taxon correspondiente
    dfTaxon[especie_actual].listaPoblacionesAsociadas <-
       poblacion
```

Código 3.3: Pseudocódigo para asignar las poblaciones a un taxón.

Esto puede parecer una buena idea pero presenta varios problemas. Uno de ellos es que nos estamos recorriendo el dataframe *Poblacion* entero y, por cada fila estamos realizando una operación. A nivel tiempo de ejecución esto es muy costoso.

Otro problema es que a pesar de que parece totalmente intuitivo el poder asignar una población (que en este caso es una fila del dataframe *Poblacion*, y a su vez, un dataframe) a una lista, R solo permite realizar esta operación una vez. Es decir, no se permite tener una lista y añadir varios dataframes como elementos de la lista de forma que la lista resultante sea una lista de dataframes. R solo permite tener una lista con un único dataframe. La única manera de introducir varios dataframes a la lista, es envolviendo cada dataframe en una lista, dando como resultado una lista con un único elemento (el dataframe) y que sea esa lista la que se mete a la lista de poblaciones asociadas. Si se lleva a cabo esta solución, la lista de poblaciones asociadas para cada taxón será una lista de listas de dataframe. Esto trae un problema crítico y es que la base de datos aumentaría mucho su tamaño ya que estaríamos desaprovechando espacio de almacenamiento en guardar listas de 1 elemento para almacenar los dataframes. Además, si queremos recuperar la lista de poblaciones asociadas a un taxón, estaríamos recuperando una lista de listas de población. Obligando a invertir tiempo en desenvolver cada una de las listas, ya sea a nivel de consultas a la base de datos o en el back-end de la aplicación web que se implementará en un futuro (se debe acceder al primer elemento de cada lista de dataframe ya que estas listas guardan únicamente un dataframe, la población).

No cabe duda que esto es muy costoso en cuanto a espacio y tiempo de ejecución. Si nos tuviéramos que recorrer el dataframe *Registro de ocurrencia* para meter los registros en las poblaciones, nos estaríamos recorriendo 1806711 filas. El tiempo en recorrernos todo el dataframe y el tamaño final en la base de datos sería desproporcionado.

Para minimizar este problema recordé un comando de R que había aprendido durante mi formación en esta tecnología. Este es el comando *"split"*. Este comando es una función muy útil para dividir un dataframe en una lista de subdataframes en base a una columna específica. Esta función recibe 2 parámetros, el primero es el dataframe sobre el que se realizará la operación y el segundo es el nombre de la columna sobre la que se dividirá el dataframe. El objetivo de usar este método es dividir el dataframe *Poblacion* en listas de subdataframes en base a la columna *species_LP*. De esta manera obtendremos una lista de 521 subdataframes en los que cada uno de estos guardará los datos de las poblaciones correspondientes a una especie diferente. Por ejemplo, el primer subdataframe de la lista guardará todos los datos (todas las poblaciones) asociadas al taxón nº1, el segundo de la lista los datos (poblaciones) asociadas al taxón nº2, etc.

El pseudocódigo quedaría de la siguiente manera:

```
listaSubdataframes <- split(dfPoblacion,dfPoblacion.species_LP)
for subdataframe in listaSubdataframes:
    # Obtener el nombre de la especie asociada a ese subdataframe
    especie_actual = subdataframe[[1]].species_LP
    # Asignar el dataframe al taxon correspondiente
    dfTaxon[especie_actual].Poblacion <- subdataframe
```

Código 3.4: Pseudocódigo para asignar las poblaciones a un taxón usando la función de R *split*

Tal y como se comentaba antes, R no permite tener una lista de dataframes. Lo que en realidad el comando *split* devuelve es una lista donde cada elemento no es un dataframe, si no una lista con un único elemento (ese dataframe). Se puede comprobar esto observando el tipo de cada elemento de la lista usando el comando *class()*, accediendo a él con corchetes, por ejemplo: *class(listaSubdataframes[1])* devuelve que es de tipo lista. Sin embargo si accedemos al elemento que guarda esa lista con dobles corchetes de la siguiente manera: *class(listaSubdataframes[[1]])* nos devuelve que es de tipo dataframe. De esta manera verificamos que el comando *split* devuelve una lista de listas de dataframe.

R tampoco permite asociar un dataframe a un campo específico de otro dataframe ya que si se hace saltaría el siguiente aviso: *"Warning message: El número de items para para sustituir no es un múltiplo de la longitud del reemplazo"*. Al saltar este aviso, R inserta únicamente la primera columna del dataframe, cosa que no queremos. Esto no lo estamos haciendo ya que lo que estamos asignando es una lista con un único elemento (el dataframe con todas las poblaciones del taxón).

A pesar de presentar una lista de un único elemento donde ese elemento es el dataframe, MongoDB lo interpreta como si fuera un array donde cada fila de ese dataframe es un elemento en el array. Eso es justo lo que se busca, por cada taxón, tener la lista de poblaciones asociadas.

De esta manera hemos logrado solucionar el problema que anteriormente se planteaba, evitando guardar las poblaciones individualmente como listas de un único elemento. También hemos minimizado el problema del tiempo de ejecución ya que hemos evitado recorrernos las 611611 filas del dataframe *Poblacion* para recorrernos 521 elementos de la lista listaSubdataframes. Sin duda un gran avance ya que el numero de iteraciones que realiza nuestro bucle for ha disminuido bastante.

El problema de esto es que a pesar de que el número de iteraciones sea el menor de los dos dataframes que vamos a relacionar, si queremos relacionar los dataframes *Poblacion* que tiene 611611 filas y el de *Registro de ocurrencia* que posee 1806711 filas, igualmente el bucle for realizará 611611 iteraciones ya que la lista que el comando *split* genera tiene 611611 subdataframes, cada uno de estos

guarda la información de los registros de ocurrencia asociados a cada población.

3.1.5. Evaluación

Aunque el número de iteraciones sea elevado, se ha evaluado el tiempo que tardaría el ejecutarse el fragmento de código que introduce los registros de ocurrencia en las poblaciones dando lugar a los siguientes resultados:

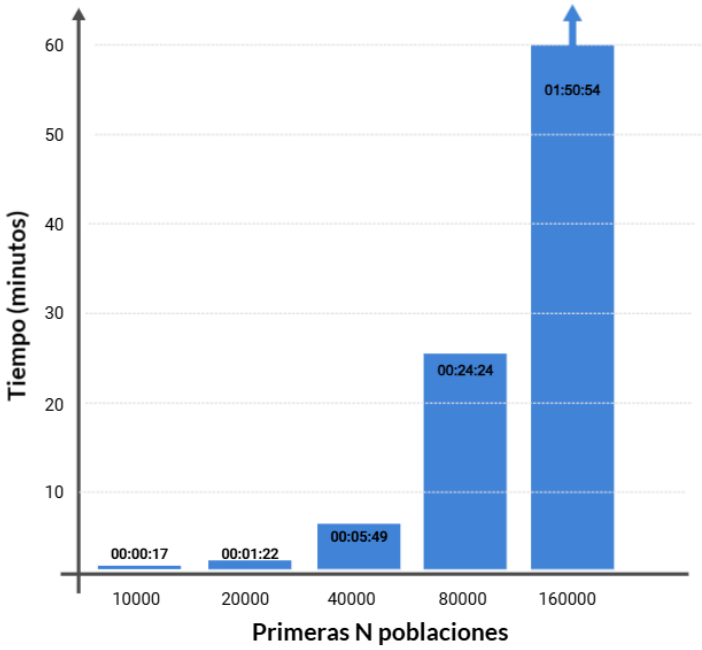

Figura 3.6: Tiempos en insertar los registros en las primeras N poblaciones

En este experimento se han hecho 5 pruebas diferentes. En cada una de ellas se ha escogido las primeras N poblaciones del dataframe *Poblacion* donde N comienza en 10000 y se va doblando su número en cada una de las pruebas. En cada una de ellas, se han asociado los registros correspondientes a esas poblaciones y los tiempos de ejecución que han dado como resultado son bastante diferentes de lo que pensaba.

Al doblar las poblaciones, es lógico pensar que el tiempo de ejecución se doblará a su vez. Como se puede observar esto no ocurre. En lugar de doblarse los tiempos, el tiempo de ejecución en cada prueba es aproximadamente 4.45 veces veces mayor que el de la prueba anterior. Esta relación se ha calculado de la siguiente manera:

Prueba	Tiempo	Tiempo (segundos)	Relación
Prueba 1	00:00:17	17	–
Prueba 2	00:01:22	82	$\frac{82}{17} \approx 4{,}82$
Prueba 3	00:05:49	349	$\frac{349}{82} \approx 4{,}26$
Prueba 4	00:24:24	1464	$\frac{1464}{349} \approx 4{,}19$
Prueba 5	01:50:54	6654	$\frac{6654}{1464} \approx 4{,}54$
		Relación Media:	$\frac{4{,}82+4{,}26+4{,}19+4{,}54}{4} \approx 4{,}45$

Tabla 3.1: Cálculo de la relación media entre pruebas

Investigando sobre las posibles causas del crecimiento exponencial del tiempo de ejecución en cada una de las pruebas, se llega a la conclusión de que realizar operaciones como la manipulación de dataframes o el tratamiento de listas de forma repetitiva conlleva a una sobrecarga del sistema debido al aumento en el uso de memoria. En este caso particular, en cada iteración del bucle se inserta una lista con un dataframe en el interior del dataframe *Poblacion*. Este hecho hace que se haya tenido que reservar memoria para crear la lista así como invertir recursos en modificar la estructura interna del dataframe donde se insertará dicha lista, acomodándola en una celda específica del mismo. La constante repetición de esta operación de reasignación de memoria incrementa consigo la demanda de recursos, lo que contribuye a una sobrecarga del sistema y se traduce así en un incremento en el tiempo de ejecución de forma exponencial.

Aunque se ha solucionado uno de los problemas, el otro persiste y considero de gran importancia encontrar una solución ya que haciendo cálculos basados en los tiempos exponenciales, se deberán invertir alrededor de 82 horas en generar la estructura. Esto es inviable.

Al ser los tiempos de ejecución tan desorbitados, no ha sido posible generar las estructuras con toda la información que se almacenará en la base de datos. Se ha logrado con las versiones de los datasets reducidos y se ha comprobado que funciona, pero será necesario volver a modelar para encontrar la solución óptima.

3.2. Segunda iteración

En esta iteración se han suprimido dos etapas, en concreto las etapas de *Entendimiento de los datos* y *Preparación de los datos*. Esto se debe a que los datos de los que disponemos no han cambiado, tampoco se ha visto la necesidad de modificar la forma en la que se representarán tras el desarrollo de la anterior iteración.

3.2.1. Entendimiento del contexto

Al igual que en la anterior iteración, esta etapa es útil para refrescar conceptos e investigar sobre temas de interés para la investigación que se está desarrollando. En todo proyecto, a medida que avanza, van surgiendo nuevas necesidades y retos que obligan a adaptarse a esta situación cambiante. Es por esto que me he visto en la necesidad de investigar sobre temas relacionados con la reducción del tiempo de ejecución en R.

Por un lado, he investigado sobre librerías de paralelización de tareas, librerías que permiten el uso de threads para así acelerar el proceso de generación de las estructuras. Es una de las opciones clásicas si se quiere agilizar un determinado proceso, dividiendo el trabajo entre diferentes hilos. Hay una gran cantidad de estas librerías que se pueden utilizar en R. Sabiendo de su existencia, se pondrán en funcionamiento en un futuro.

Por otro lado, se han ampliado conocimientos sobre R, descubriendo un tipo de operaciones clave para la investigación, las operaciones vectorizadas. Estas operaciones son capaces de realizar cálculos o ejecutar determinadas operaciones de manera simultánea sobre todos los elementos de un vector. Son operaciones muy simples, que dan como resultado un código limpio, fácil de entender y lo más importante, rápido.

3.2.2. Modelado

Tras la evaluación del modelo hasta la fecha implementado, se llega a la conclusión de que es totalmente necesario llevar a cabo otras estrategias diferentes, dejando a un lado el uso de bucles ya que esta solución implica una gran cantidad de tiempo en completarse. Fueron muchas las horas que invertí en encontrar la solución óptima, implementé scripts que reducían el tiempo en gran medida hasta que finalmente conseguí implementar la solución que reducía al máximo el tiempo en construir la estructura.

La solución final resultó ser mucho más simple de lo que imaginaba. Tras numerosos intentos en los que empleaba una gran cantidad de líneas de código para relacionar por ejemplo los registros de ocurrencia con las poblaciones, conseguí reducir la implementación a tan solo dos o tres líneas.

Es importante destacar que hay 2 tipos de soluciones, ambas usan operaciones vectoriales y cada una de ellas se empleará en función de lo que se quiera relacionar. Si se quiere relacionar entidades cuyas relaciones son 1:N se empleará la *Solución 1*, si se quiere relacionar entidades cuyas relaciones son 1:1 se empleará la *Solución 2*. A continuación se explicarán ambas implementaciones:

3.2.2.1 Solución 1 (Relaciones 1:N)

Esta primera solución se implementará cuando se quieran relacionar entidades cuyas relaciones son de 1 a N. Por ejemplo, es el caso de la relación entre poblaciones y registros de ocurrencia ya que una población consta de N registros de ocurrencia. También ocurre con la relación entre taxones y poblaciones ya que a un taxón le corresponden N poblaciones.

Comenzando con la explicación de la implementación, esta hace uso del comando *split*, comando que había sentido necesario desde que descubrí su potencial. Tras muchas pruebas, mi intuición resultó acertada. También se usa el comando *match*, función que busca coincidencias entre los elementos de dos vectores (en este caso los vectores serán ciertas columnas de ciertos dataframes) y devuelve un vector con los índices de las coincidencias en el segundo de los vectores. Más adelante se explicará con más detalle.

La combinación de los valores devueltos por ambas funciones han culminado en la aplicación de una operación vectorizada, operación mucho más eficiente que los bucles ya que en lugar de iterar sobre cada elemento de un dataframe de forma individual y aplicar una operación (en este caso de asignación), las asignaciones se realizan a todas las filas del dataframe simultáneamente, reduciendo al máximo el tiempo de procesamiento.

A continuación se muestra el código empleado para asignar los registros de ocurrencia a las poblaciones y posteriormente se explicará linea por línea su funcionamiento:

```
1 listaRegistrosPoblaciones <- split(RegistroDeOcurrencia,
    RegistroDeOcurrencia$POPID)
2 indices <- match(Poblacion$POPID, names(listaRegistrosPoblaciones
    ))
3 Poblacion$RegistroDeOcurrencia <- listaRegistrosPoblaciones[
    indices]
```

Código 3.5: Código para asignar los registros de ocurrencia a las poblaciones (1:N)

- **Primera línea:** Tras ejecutarse esta línea, *listaRegistrosPoblaciones* será una lista de 611611 elementos donde cada elemento es una lista con un dataframe que guarda tantas filas como registros de ocurrencia tenga esa población. Esta lista es una lista nombrada por el identificador *POPID* ya que cada dataframe representa todos los registros de ocurrencia de una misma una población, esto nos facilitará la asociación con las poblaciones correspondientes más adelante. Recordando lo que se explicaba anteriormente, *split* devuelve una lista de listas de dataframe ya que R no permite guardar una lista de dataframes. Un ejemplo visual simplificado de lo que guarda *listaRegistrosPoblaciones* sería el siguiente:

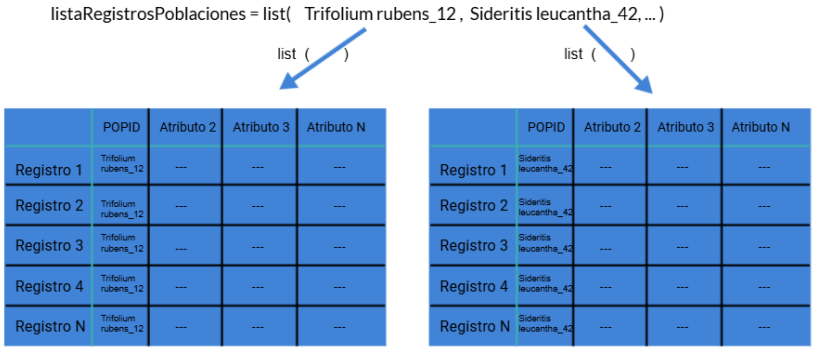

Figura 3.7: Ejemplo visual correspondiente a la primera línea de código de la solución final (1:N)

- **Segunda línea:** Tras ejecutarse esta línea, la variable *indices* será un vector de 611611 elementos que guardará la posición de la lista *listaRegistrosPoblaciones* donde se encuentra cada una de las poblaciones del dataframe *Poblacion*. Esto es posible porque la lista *listaRegistrosPoblaciones* es una lista nombrada, donde cada nombre representa el POPID de los registros de ocurrencia que alberga. Un ejemplo representativo es el siguiente:

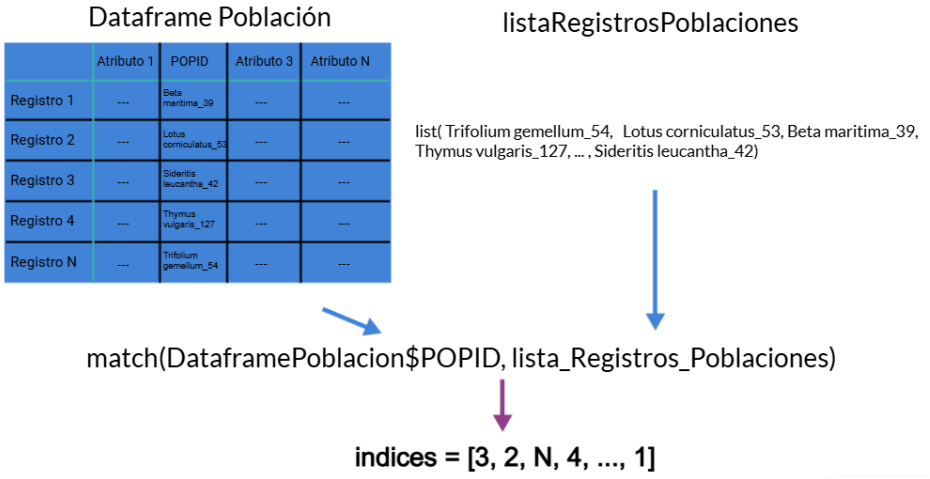

Figura 3.8: Ejemplo visual correspondiente a la segunda línea de código de la solución final (1:N)

- **Tercera línea:** Tras ejecutarse esta línea, el dataframe *Poblacion* tendrá una nueva columna llamada *RegistroDeOcurrencia* que contendrá el dataframe correspondiente de la lista *listaRegistrosPoblaciones*. Como se decía antes, esta asignación es eficiente porque es vectorizada ya que se aplica de forma simultánea a todas las filas aprovechando el vector *indices*.

  Como se comentaba, R no permite asignar un dataframe a una celda específica de un dataframe. En realidad no se está asignando un dataframe

ya que si de verdad queremos asignarlo (cosa que no se puede) deberíamos acceder a él poniendo doble corchete ([[indices]]) de la siguiente manera:

```
Poblacion$RegistroDeOcurrencia <- listaRegistrosPoblaciones[[
    indices]]
```

Código 3.6: Ejemplo (erróneo) de asignación de un dataframe a una celda específica

Accediendo al elemento de la lista sin doble corchete, estamos recuperando lo que en realidad guarda la lista, una lista con un único elemento, el dataframe. De esta manera R si permite asignar esta lista de un elemento a la posición correspondiente del dataframe *Poblacion* y, como se explicaba anteriormente, MongoDB lo interpreta como si fuera un array donde cada elemento es una fila del dataframe. Justo lo que se busca.

3.2.2.2 Solución 2 (Relaciones 1:1)

Si bien con anterioridad hemos hablado de la implementación para tratar las relaciones 1:N. A continuación veremos la implementación para tratar las relaciones 1:1.

Estas relaciones se dan por ejemplo para relacionar una población a un registro de ocurrencia. Justo en el sentido contrario al que teníamos antes. De la manera anterior relacionábamos registros de ocurrencia a una población, ya que una población tiene varios registros de ocurrencia pero en este caso, si se le da la vuelta, un registro de ocurrencia pertenece únicamente a una población. Otro ejemplo más sería la relación entre un registro de ocurrencia y un taxón, ya que una registro pertenece únicamente a una especie (taxón).

Si bien se podrían haber implementado sin ningún problema estas relaciones con la *Solución 1*, la implementación de una nueva solución se ha visto motivada por el elevado tamaño del dataframe resultante tras crear la estructura manejando las relaciones 1:1 con esa solución. Dando lugar a una base de datos mucho más pesada.

Este gran tamaño se debe a que en la anterior solución, las listas que se creaban tenían sentido ya que en su interior había más de un elemento. En otras palabras, una población tenía su lista de registros de ocurrencia asociados como debe ser. El problema es que si estamos trabajando con relaciones 1:1, no tiene nada de sentido generar listas para almacenar un único elemento. Si seguimos la *Solución 1* para implementar estas relaciones, nos encontraríamos con que un registro de ocurrencia (el cual siempre está asociado de manera 1:1 a una población) tiene una lista de un elemento donde ese elemento es la población.

Si extrapolamos esto a todos y cada uno de los registros de ocurrencia, estamos

malgastando espacio en crear una lista con un único elemento, donde ese elemento es su población correspondiente. Lo mismo ocurre con las demás relaciones 1:1 que se pueden implementar. Lo suyo es que en vez de guardar una lista con un elemento, guardar directamente el elemento.

La implementación para tratar las relaciones 1:1, con un ejemplo en el que se relaciona las poblaciones a los registros de ocurrencia es la siguiente:

```
indices <- match(RegistroDeOcurrencia$POPID,Poblacion$POPID)
RegistroDeOcurrencia$Poblacion <- Poblacion[indices, ]
```

Código 3.7: Código para asignar las poblaciones a los registros de ocurrencia (1:1)

Esta implementación está claramente influenciada por la propuesta en la *Solución 1*. Es prácticamente igual salvando que en esta no se hace uso del comando *split*. Lógico, ya que no queremos listas en la estructura, queremos directamente el elemento que la contendría.

De igual manera que la solución para las relaciones 1:N, esta hace uso de una operación vectorizada, apoyándose en el comando *match* para obtener un vector de 1806711 elementos donde cada uno de los elementos representa el índice de la fila en la que se encuentra la población asociada a cada uno de los registros de ocurrencia.

Una imagen representativa del funcionamiento del comando *match* para esta ocasión es el siguiente:

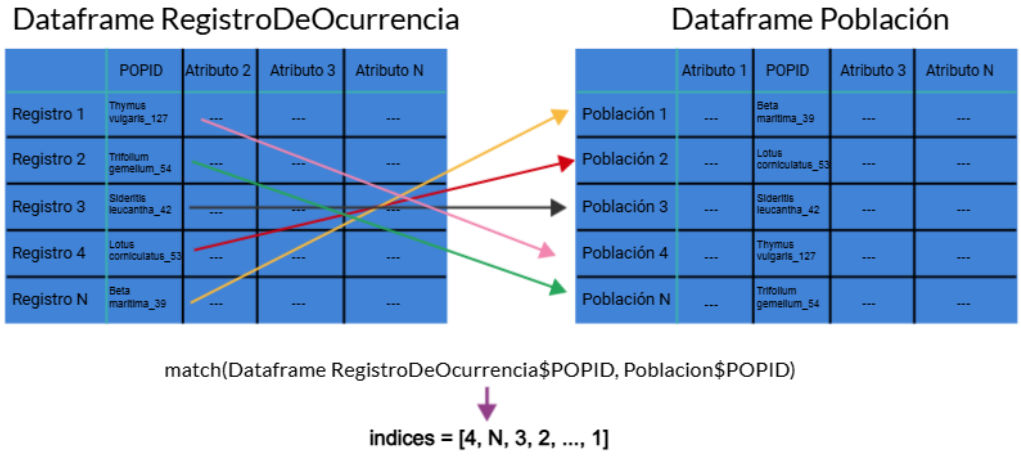

Figura 3.9: Ejemplo visual del funcionamiento del comando match entre 2 dataframes

Una vez obtenidos los índices, se lleva a cabo la operación vectorial, donde se asignan las filas correspondientes del dataframe *Poblacion* al dataframe *RegistroDeOcurrencia*.

Tal y como se comentaba con anterioridad, R no permite la asignación individual de un dataframe a una celda específica de otro dataframe ya que nos salta el siguiente aviso: *"Warning message: El número de items para para sustituir no es un múltiplo de la longitud del reemplazo"*. En este caso, no se está haciendo esto. En realidad se está asignando el dataframe de manera embebida, añadiendo las columnas correspondientes de la entidad *Poblacion* al dataframe *Registro de ocurrencia*. Estas nuevas columnas se llamarán como se llamaban en la entidad *Poblacion* pero empezando su nombre por *Poblacion$*, representando así que esa información se guarda de forma embebida. Para entender esto mejor, a continuación se presenta un ejemplo visual del resultado tras ejecutarse esta operación:

Dataframe RegistroDeOcurrencia

	POPID	Atributo 2	Atributo N	Poblacion$Atributo1	Poblacion$POPID	Poblacion$Atributo3
Registro 1	Thymus vulgaris_127	---	---	---	Thymus vulgaris_127	---
Registro 2	Sideritis leucantha_42	---	---	---	Sideritis leucantha_42	---
Registro 3	Trifolium gemellum_54	---	---	---	Trifolium gemellum_54	---
Registro 4	Lotus corniculatus_53	---	---	---	Lotus corniculatus_53	---
Registro N	Beta maritima_39	---	---	---	Beta maritima_39	---

Figura 3.10: Ejemplo visual que muestra de manera simplificada la estructura resultante del dataframe RegistroDeOcurrencia

De esta manera se observa que al dataframe *RegistroDeOcurrencia* se han añadido todos los atributos del dataframe *Poblacion* y además se han almacenado los datos de las poblaciones correspondientes en base al valor del vector *indices* calculado con anterioridad, que indica en qué posición del dataframe *Poblacion* se encuentra la población a insertar. Se sabe que la operación se ha completado correctamente si los valores de los atributos *POPID* en la misma fila coinciden.

Con todo lo anteriormente mencionado y presentados los dos tipos de soluciones, se puede decir que el modelado de las estructuras se ha completado. Se ha logrado optimizar el tiempo en generarlas al máximo.

Basta con replicar estas soluciones en el resto de scripts para generar las relaciones 1:1 o 1:N que se manejen. En el caso de la estructura *ColeccionesSepradas* no se aplicarán estas relaciones entre entidades ya que la información sobre cada una de las áreas se guarda en colecciones diferentes.

3.2.3. Evaluación

1. Evaluación de tamaños en las relaciones 1:1 implementando ambas soluciones:

Para demostrar que la *Solución 1* no es adecuada para tratar las relaciones 1:1, se han realizado diversas pruebas con el fin de demostrar que el espacio ocupado por el dataframe generado, así como el tamaño en JSON, que será el espacio que aproximadamente ocupará en la base de datos, es significativamente mayor en comparación con la estructura generada por la *Solución 2*.

Los tamaños del dataframe resultante (calculados con el comando *object.size()*) y del JSON [18] generado (calculado con el comando *toJSON()*) para tratar las relaciones 1:1 en la estructura *Registro_TaxonPoblacion* implementando ambas soluciones se recogen en la siguiente tabla:

	Tamaño dataframe	Tamaño JSON
Solución 1	34.15 GB	5.1 GB
Solución 2	5.17 GB	3.75 GB

Tabla 3.2: Comparación de tamaños de dataframe y JSON para las dos soluciones implementando relaciones 1:1 en la estructura Registro_ TaxonPoblacion.

De esta manera se confirma que para tratar las relaciones 1:1, la mejor idea es implementar la *Solución 2* ya que la *Solución 1* resultaría en un malgasto de espacio de almacenamiento además de la futura necesidad de desenvolver las listas para obtener el dataframe que se quiere recuperar.

Es curioso observar que a pesar de que el tamaño del dataframe generado por la *Solución 1* es 6.6 veces mayor que el generado en la *Solución 2*, este último no es 6.6 menor en términos de espacio en JSON.

2. Cargar las estructuras a MongoDB y evaluar resultados

Con el dataframe totalmente estructurado en R, el hecho de cargar su información a la base de datos se puede llevar a cabo de diferentes maneras. Primero de todo, recordar que en un dataframe, cada una de sus filas representa un documento en la base de datos y, cada una de las columnas representa un atributo que puede tener dicho documento.

Durante la mayor parte del desarrollo, se ha usado la librería *mongolite* [19]. Sin duda es la más común, la más utilizada si se quiere interactuar entre R y MongoDB debido a su facilidad de uso. Permite de manera cómoda realizar operaciones de inserción, eliminación, consulta y actualización desde el propio entorno de R. Aunque no he indagado sobre las operaciones de consulta y eliminación, puedo decir que la inserción y la modificación se realizan de manera

sencilla. Entre ellas dos, supone algo más de complejidad la modificación ya que se debe escribir la *query* (consulta) en el código.

Uno de los puntos fuertes de esta librería es que no necesita de la conversión previa del dataframe a formato JSON para insertar los datos, este proceso lo hace automáticamente. Sin embargo, posee aspectos negativos que en un futuro me hicieron decantar por otras alternativas. Uno de ellos es que se debe introducir la URL de mongo directamente en el código. Esta URL tiene el siguiente formato:

```
1 mongodb+srv://usernameMongo:passwordMongo@cluster0.xlav1ka.
    mongodb.net
```

Código 3.8: Ejemplo de URL de conexión a MongoDB

Como se puede apreciar, el nombre de usuario y la contraseña se escriben directamente en el código, lo que puede conllevar a problemas de seguridad debido a esta mala práctica. Además de la URL de mongo, se debe especificar directamente en el código el nombre de la base de datos y la colección en la que se introducirá la información. Esto supone que si se quiere cambiar de URL, base de datos o nombre de colección, la persona que vaya a ejecutar el script, no solo tendrá que ejecutarlo, sino también modificarlo.

A pesar de que en un futuro no se usará *mongolite* para cargar los datos a la base de datos, esta etapa de esta iteración sirvió para darse cuenta de los problemas que acarreaba esta librería.

La librería *mongolite* coge el dataframe que quieras insertar a la base de datos y lo inserta en ella, no es capaz de realizar ningún proceso de limpieza previo con el objetivo de no insertar información irrelevante a la base de datos. Esto supone un problema ya que nuestra base de datos ocupará más espacio y almacenará información que no aporta ningún valor.

En concreto, se almacenará información como cadenas vacías, problema que se comentaba con anterioridad. También se almacenarán listas con un elemento donde ese elemento es una cadena vacía o *None*, fruto de la función *ConvertirEnLista* aplicada en la preparación de los datos de la etapa anterior. Además, se insertarán objetos vacíos cuando todos los atributos de una determinada entidad estén a "NA". Todo esto es información que ensucia la base de datos.

Gracias a la evaluación de estos aspectos, se tendrá en cuenta las conclusiones obtenidas para tratar de proporcionar solución a los problemas. En concreto, se desarrollará un script de Python que hace de intermediario entre R y MongoDB para cargar la información a la base de datos.

3. Dificultades con la estructura *Taxon_Poblaciones_Registros*

Esta estructura se diferencia del resto porque posee documentos muy pesados, con una gran cantidad de información embebida. Esto trae problemas a la hora

de insertar la información a MongoDB.

Una de las limitaciones que presenta Mongo es que no permite la inserción de documentos que pesan más de 16 MB [20]. Si se trata de insertar un documento que exceda este límite, R nos avisará con el siguiente error en el proceso de inserción:

Error: Document 0 is too large for the cluster. Document is 22784361 bytes, max is 16777216.

Este error nos expresa que el documento que queremos insertar pesa 22.78 MB y lo máximo permitido es 16.77 MB. Por lo tanto, se aborta el proceso de inserción.

Una de las principales causas por las que MongoDB hace esto es porque los documentos muy grandes afectan al rendimiento y eficiencia de la base de datos. Cuanto más grande sea un documento, más tardará en procesarse, lo que conlleva a un consumo más elevado de memoria y una ralentización de las operaciones de la base de datos.

Hay que destacar que este problema solo ocurre en los documentos que se almacenan en la colección correspondiente a la estructura *Taxon_Poblaciones_Registros* porque los documentos generados son demasiado largos. Un taxón puede tener por ejemplo 6000 poblaciones y, cada una de esas poblaciones tener a su vez una lista con 10 registros de ocurrencia. Sin duda, queda un documento muy largo y pesado.

Este problema se solventará en la etapa de modelado de la siguiente iteración.

3.3. Tercera iteración

Al igual que en la anterior iteración, se han suprimido las etapas de *Entendimiento de los datos* y *Preparación de los datos* debido a que los datos siguen siendo los mismos. Tampoco surge la necesidad de modificar su naturaleza.

3.3.1. Entendimiento del contexto

Sabiendo que se va prescindir de la librería *mongolite*, se han explorado otras alternativas para insertar datos en MongoDB. Existen otras librerías como RMongo pero no ha recibido actualizaciones significativas en los últimos años, está obsoleta.

Sabiendo esto, me he visto en la necesidad de recopilar información para implementar otras estrategias ajenas al uso de librerías directamente en R. Estas

estrategias se basan en el uso de Python, implementando un script que, apoyándose en las librerías adecuadas, sea capaz de recibir la información estructurada de R, limpiar los datos irrelevantes y, por último, insertar la información en MongoDB. Barajando diferentes posibilidades, he logrado implementar un script que realiza este proceso a la perfección. Se explicará su funcionamiento en la etapa de modelado.

También se ha investigado sobre las posibles formas de almacenar documentos que superen los 16 MB en la base de datos con el objetivo de lograr almacenar toda la información de la colección *Taxon_Poblaciones_Registros*.

Ya que se realizarán consultas a la base de datos una vez se haya cargado correctamente toda la información a las colecciones, ha sido necesario investigar sobre el lenguaje propio de consultas de MongoDB (MQL, MongoDB Query Language) para así generarlas correctamente, haciendo uso de la menor cantidad de líneas posibles.

3.3.2. Modelado

En esta etapa se han solucionado problemas encontrados en la anterior iteración, se ha desarrollado el script de Python necesario para insertar correctamente la información en MongoDB y se han solucionado los problemas surgidos en esta etapa. A continuación se explica con más detalle cada uno de estos apartados.

3.3.2.1 Solución al límite de 16 MB por documento en MongoDB

Sabiendo que MongoDB no permite almacenar documentos que ocupan más de 16 MB porque afecta directamente al rendimiento y eficiencia de la base de datos, ha sido necesario ajustar el dataframe generado tras implementar la estructura. Este ajuste solo se ha implementado en el script correspondiente a la estructura *Taxon_Poblaciones_Registros* ya que es la única que presenta este problema. Como consecuencia, ha aumentado notablemente el tiempo de ejecución para generar dicha estructura.

En primer lugar, se han identificado los documentos que en la base de datos ocuparán más de 16 MB, es decir, las filas del dataframe que ocupan más del límite permitido. Hay que destacar que el peso de cada fila no es el peso en R a nivel de dataframe. Ha sido necesario transformar cada una de las filas del dataframe a JSON para así calcular el espacio real que ocupará esa fila en MongoDB.

MongoDB guarda la información en formato BSON (Binary JSON) [21], pero se ha calculado el peso en JSON ya que R no tiene ninguna librería para transformar los datos a ese formato. Esto no nos supone ningún problema ya que el peso en los diferentes formatos no difiere apenas. Es una muy buena aproximación. Al

ser esta una operación que demora más tiempo del esperado, se han empleado threads para reducir al máximo el tiempo de ejecución.

Una vez identificados los pesos de cada fila, el dataframe original se ha dividido en dos. Uno almacena las filas que representan los documentos que superan el límite y otro las que no lo superan. Aunque el límite son 16 MB, he establecido manualmente en 10 MB el tamaño máximo que pueden tener los documentos tras implementar el ajuste. Esto lo he hecho ya que contra más grande sean los documentos, más lento funcionan las operaciones de la base de datos. Simplemente bajando el límite, la base de datos funciona de una manera mucho más rápida y eficiente.

El dataframe en el que se encuentran los documentos que pesan menos de 10 MB no se modificará. Sin embargo, para el otro dataframe se ha implementado un algoritmo que divide el documento tantas veces como sea necesario para que se pueda almacenar la misma información sin que se superen los 10 MB de espacio en cada documento. Esta división se ha implementado en base a la división de la lista de poblaciones de un taxón.

Explicado con un ejemplo: Si un documento pesa 25 MB, se deberán hacer $25/10 = 2.5$, es decir, 3 particiones. Por lo tanto, donde antes había un documento que representaba un taxón con 1500 poblaciones, ahora habrá 3 documentos donde se guardará la misma información del taxón excepto la información de las poblaciones, ya que en el primer documento se encontrarán las poblaciones de la 1 a la 500, en el segundo de la 501 a la 1000 y en el tercero de la 1001 a la 1500. El único punto negativo es que en los 3 documentos, la información del taxón se repetirá y, por ejemplo, si se quiere realizar una consulta en la que se quiera obtener un determinado taxón por su identificador, si ese taxón ha sido ajustado para no superar el límite, la consulta devolverá N documentos en lugar de 1, donde N es el numero de divisiones que se han hecho.

Aunque anteriormente comenté que MongoDB no permite almacenar documentos que ocupen más de 16 MB, en realidad se puede superar este límite usando GridFS [22]. Esta es una tecnología desarrollada por MongoDB que permite dividir un archivo grande en pequeños fragmentos llamados *chunks* y almacenar así documentos que excedan ese límite. Esta solución se suele implementar para almacenar imágenes, vídeos o documentos pesados en la base de datos y no es adecuado para almacenar datos estructurados (como los de un dataframe). Tampoco se permite realizar consultas, por lo tanto, a pesar de su existencia, GridFS no será de utilidad para lo que se desea hacer.

3.3.2.2 Script en Python

El script implementado en Python hará de intermediario entre R y MongoDB para limpiar los datos irrelevantes y almacenar la información en la base de datos. Este script debe estar en ejecución en el momento en el que se vaya a insertar la

información ya que de lo contrario no se insertará nada. Para configurar el entorno correctamente antes de ejecutarlo, se deberán seguir los pasos que se muestran en el apéndice B de este documento. A continuación se detalla el funcionamiento del script implementado:

En primer lugar, para ejecutarlo se deberán especificar tres parámetros de forma obligatoria: La URL de conexión a MongoDB, el nombre de la base de datos y el nombre de la colección en la base de datos. Ante la ausencia de alguno de ellos, el script no se ejecutará y mostrará el mensaje de error correspondiente. De esta manera, no se tendrá directamente en el código datos críticos a nivel de seguridad como la URL de conexión a Mongo.

En cuanto al nombre de las colecciones, se han establecido 4 posibles nombres de colección, una para cada estructura que se ha implementado. De esta manera, el usuario que ejecute el script estará obligado a que su colección se llame de una determinada manera a no ser que modifique los posibles valores directamente en el código. Esto puede parecer algo sin sentido pero se ha hecho para controlar la forma en la que se trata la estructura *ColeccionesSeparadas*. Este es un caso especial ya que con esa estructura, tenemos 3 colecciones diferentes, cada una de ellas con un nombre y, para evitar ejecutar el script 3 veces, especificando 3 nombres diferentes en cada una de las ejecuciones, se ejecutará una única vez ya que los nombres de las colecciones viajarán en el cuerpo del JSON desde R. Por lo tanto, si se ejecuta el script con el parámetro correspondiente al nombre de la colección con valor de *ColeccionesSeparadas*, el script mirará en el cuerpo del JSON para determinar el nombre de la colección. Este es el único caso en el que el nombre de la colección viaja en el cuerpo del JSON. Para las otras estructuras, el nombre de la colección se establece directamente a la hora de ejecutar el script.

En cuanto al funcionamiento del script, a parte de utilizar la librería *argparse* [23] para permitir el uso de parámetros, se han utilizado las librerías *Flask* [24] y *pymongo* [25] .

La librería *Flask* nos será de gran utilidad para crear un servidor web que se mantendrá en escucha en un puerto determinado esperando a recibir los datos en formato JSON. Desde Python, se define la ruta en la que el servidor acepta solicitudes POST y en cuanto se ejecuta el script, se pondrá en escucha esperando a recibir el contenido. El contenido se calculará en R, transformando en JSON el dataframe que contiene la estructura con los datos que se cargarán a la base de datos. Una vez se ha calculado el JSON, se enviará ese contenido a la ruta del servidor en escucha de la siguiente manera:

```
1 POST("http://localhost:5000/insert", body = json, encode = "json"
    )
```

Código 3.9: Código para enviar la información en JSON al servidor en escucha

De esta manera, el JSON con la información que se quiere insertar lo tenemos en Python. Con él podemos hacer lo oportuno para perfeccionar la información que se insertará en MongoDB.

Para limpiar los valores indeseados, se ha implementado el método *limpiarDatos*, que recibe el json y, de manera recursiva elimina toda aquella información que carezca de utilidad en la base de datos. La información que se ha limpiado es la siguiente:

- Valores con cadenas vacías. De esta manera se soluciona el problema nombrado con anterioridad, en el que disponíamos de datos cuyo valor era la cadena vacía.
- Listas vacías.
- Claves cuyo valor es None. Se da si todos los atributos de una clave no tienen valor. Es decir, si por ejemplo una entidad embebida de la entidad *InformaciónTaxonómica* tiene atributos y ninguno de ellos tiene valor, el json que se genera para esa entidad es con un valor de None.
- Todos los objetos sin contenido ({}). Pueden surgir fruto de la limpieza, es decir si se han eliminado todas las claves de un determinado objeto.

Gracias a esta limpieza, la base de datos no contendrá valores que no aportan nada, al revés, todo valor que se almacena es útil. Además, se reduce en gran cantidad el espacio final de la base de datos, por lo que el ahorro económico en el plan de MongoDB Atlas es el máximo.

Una vez disponemos del JSON limpio, tenemos todo lo necesario para proceder a insertar la información en la base de datos. Para ello haremos uso de la librería *pymongo*, útil para conectar y operar con MongoDB. En el script se establecerán los valores de la URL de conexión a Mongo, así como el nombre de la base de datos y la colección que el usuario ha especificado en los parámetros, en el caso de seleccionar *ColeccionesSeparadas* para el nombre de colección, se elegirá como nombre la recibida en el cuerpo del JSON. Una vez se haya configurado todo correctamente, se utilizará el método *insert_many* de la librería *pymongo* para cargar la información a MongoDB.

Gracias a que el script se ejecuta en modo de depuración (*app.run(debug=True)*), se podrá ver si la inserción se ha realizado correctamente ya que se mostrará por pantalla los mensajes de error en caso de que algo falle o los mensajes de éxito en caso de que todo vaya bien. Además se mostrarán sus códigos de estado asociados.

3.3.2.3 Atributo _row creado inesperadamente

La información que recibe Python desde R con ayuda del servidor Flask está en formato JSON. Esta información se ha generado en R a partir del dataframe

estructurado haciendo uso del método *toJSON* de la librería de R *jsonlite* [26].

Sin embargo, al hacer esta conversión se observa que los objetos introducidos fruto de las relaciones 1:1 contienen un atributo adicional llamado _row. Este atributo se añade en todas las anidaciones que presenta el objeto introducido, además de su nivel de anidación primario. Un ejemplo simplificado en código JSON donde se aprecian los atributos _row en un documento de la estructura *Registro_Taxon_Poblacion* es el siguiente:

```
{
    "_id": {
        "$oid": "6667286ff8b35d2d48c7bc5d"
    },
    "Atributo1_RegistroDeOcurrencia1": 9050,
    "Atributo2_RegistroDeOcurrencia1": "Trifolium rubens",

    "InformacionTaxonomica": {
        "Atributo1_InformacionTaxonomicaAsociada": "Trifolium rubens",
        "Atributo2_InformacionTaxonomicaAsociada": 9050,
        "Usos": {
            "USE_VALUE": "CWR",
            "_row": "1"
        },
        "RasgosEcologicos": {
            "TAXONID_LP": 9050,
            "_row": "1"
        },
        "MejoraGenetica": {
            "TAXONID_LP": 9050,
            "DetalleMejoraGenetica": {
                "TAXONID_LP": 9050,
                "RELATEDCROP": "Trifolium pratense L.",
                "_row": "1"
            },
            "_row": "1"
        },
        "_row": "1"
    }
}
```


Figura 3.11: Ejemplo de código en JSON de un documento simplificado con los atributos _row generados

Como se aprecia en la Figura 3.11, el método *toJSON* crea automáticamente el atributo _row y le pone en este caso el valor de 1. En el siguiente documento, el valor de este atributo es 1.1, en el siguiente 1.2 y así sucesivamente con el resto de los documentos de la colección.

El documento que aparece en la imagen se ha simplificado ya que se han eliminado muchos de los atributos así como el objeto *Poblacion* el cual presenta el mismo problema que el objeto *InformacionTaxonomica*. Tal y como se aprecia, se añade el atributo *_row* en el nivel de anidación principal (a este le corresponde el _row de la línea 28) así como al resto de anidaciones que se presentan en el interior de dicho objeto. Por ejemplo, para el objeto *MejoraGenetica* se le corresponde el _row de la línea 26 mientras que para el objeto *DetalleMejoraGenetica* se le corresponde el _row de la línea 24.

Investigando sobre las causas de este fenómeno, no he encontrado ninguna explicación concreta en la documentación oficial que de respuesta a mis preguntas. Si que puedo intuir que el método *toJSON* añade los atributos _row de forma automática con el objetivo de proporcionar una referencia única a cada fila del dataframe que se quiere transformar a JSON. Considero que ese es el motivo ya que si se quiere transformar únicamente una fila de un dataframe a JSON, el método *toJSON* no añade estos atributos. Esto se debe a que no hace falta añadir un elemento que distinga esa fila de otra porque solo hay una.

Al ser algo que ensucia mucho la base de datos y hace aumentar su tamaño. La solución propuesta se basa en la modificación del método *limpiarDatos* del script de Pyhton, encargado de limpiar datos irrelevantes. Se ha añadido una condición de limpieza a este método para que elimine todas las claves cuyo nombre sea _row.

3.3.2.4 Conversión de dataframe a JSON

Como se comentaba anteriormente, para insertar los datos a la base de datos hay que transformar en un paso previo el dataframe estructurado a formato JSON. Esto se realiza empleando el método *toJSON* de la librería de R *jsonlite*.

Al manejar una gran cantidad de datos, este proceso demora mucho tiempo, más incluso que la generación de las estructuras. Al ser esta operación algo inevitable, se han adoptado medidas para reducir al máximo los tiempos. Estas medidas se basan en el uso de paralelización mediante threads.

La paralelización fue algo que desde momentos tempranos de la investigación se tuvo en cuenta. De hecho, invertí tiempo de estudio e investigación con el objetivo de reducir los tiempos en generar las estructuras. Cosa que finalmente no se llevó a cabo gracias al uso de las operaciones vectorizadas.

Se han probado diversas librerías que ofrece R para la paralelización pero ninguna ha ofrecido tan buenos resultados como la librería *furrr* [27], la cual posee el método *future_map* para paralelizar el cálculo.

Tras numerosos intentos, se ha concluido que la configuración implementada (número de división de los datos y número de procesadores lógicos utilizados) es de gran importancia en los tiempos que se demora en completar la operación.

A continuación se muestra una imagen en la que se exponen las diferentes mediciones de tiempo obtenidas en convertir a JSON el dataframe generado correspondiente a la estructura *Poblacion_TaxonRegistros*. El equipo en el que se han desarrollado las pruebas tiene 8 procesadores lógicos, es decir, se pueden utilizar como máximo 8 cores para completar la operación

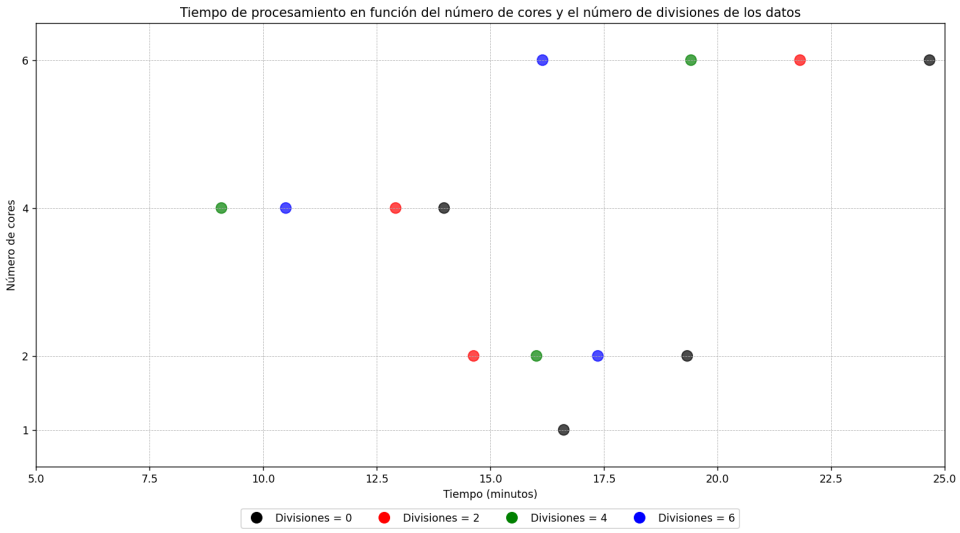

Figura 3.12: Comparación de tiempos en función de el número de cores y división de los datos

Con esto se concluye que la mejor configuración es aquella que hace uso de 4 cores y 4 divisiones de los datos, pasando de 17 minutos de procesamiento sin paralelización (1 core, 0 divisiones) a tan solo 9, una mejora notoria.

Este experimento se ha replicado de forma simplificada en otro equipo el cual solo dispone de 4 cores. Aunque los tiempos de procesamiento son peores, el mejor de todos ellos es aquel en el que se emplean 2 cores y 2 divisiones. Por lo tanto, la configuración final que se ha adoptado es aquella que hace uso de N/2 cores donde N es el número de procesadores lógicos que dispone el equipo (se conoce con la función *availableCores* de la librería *furrr*) y tantas divisiones como número de cores se empleen.

Otra conclusión relevante obtenida es que no por emplear más cores, el tiempo de procesamiento disminuirá más.

3.3.2.5 Modelado de las consultas

Para generar las consultas, se dispone de una lista de consultas frecuentes las cuales se dividen en 3 campos: consultas frecuentes a nivel de taxón (se consulta por información relativa a los taxones), consultas frecuentes a nivel de población (se consulta por información relativa a las poblaciones), y consultas frecuentes a nivel de registro de ocurrencia (se consulta por información relativa a los registros de ocurrencia).

Dada esta lista de consultas frecuentes se han implementado las *queries* pertenecientes a cada consulta en cada una de las colecciones implementadas. Se ha hecho esto ya que es distinta la manera de acceder a la misma información en

las diferentes colecciones. Posteriormente se han medido los tiempos de respuesta obtenidos con el objetivo de determinar en qué colecciones funcionan mejor cada una de las consultas, estos tiempos se evaluarán en la siguiente etapa de esta iteración.

El el apéndice de este documento se encuentran las consultas más frecuentes que se han tenido en cuenta, así como las diferentes *queries* sobre estas consultas correspondientes a cada una de las estructuras en las que se han probado. Estas *queries* se pueden probar directamente desde la shell integrada de MongoDB Compass (*mongosh*). Algunas de las consultas pueden generarse en una única línea, de manera directa. Otras sin embargo requieren de operaciones más complejas (*aggregates*).

Antes de nada, comentar que MongoDB Query Language presenta una gran cantidad de herramientas para realizar consultas y acceder así a los datos, pero no todas funcionan de la misma manera. Hay que diferenciar entre consultas que devuelven un cursor, como *find* o *aggregate* y otras que devuelven directamente la información en forma de resultado como *distinct* o *count*. Aquellas que devuelven un cursor son más rápidas y eficientes ya que no se recupera la información como tal, se recupera un cursor, objeto que permite iterar sobre los documentos de una consulta. Por lo tanto, para medir los tiempos se ha tenido en cuenta el tiempo que se tarda en obtener todos los elementos del cursor. Por otro lado, operaciones como *distinct* o *count* procesan todos los documentos de la consulta para proporcionar el resultado.

3.3.3. Evaluación

Ahora todo está listo para cargar la información a la base de datos. Tras los ajustes y soluciones implementadas, podemos estar seguros de que la base de datos guardará información relevante y bien organizada.

Cualquier persona puede almacenar datos de manera gratuita en MongoDB Atlas, la cual ofrece hasta 512 MB de almacenamiento. Para aumentar el espacio de almacenamiento de la base de datos, se deberá contratar uno de los planes que ofrece MongoDB Atlas acorde con las necesidades que se presenten.

No todas las estructuras ocupan lo mismo ya que en unas se presenta más repetición de datos que en otras. Por lo tanto, será diferente la inversión económica que se deba hacer para conseguir que la información para las diferentes estructuras se logre almacenar en la base de datos. Esto es algo que más adelante se valorará.

En mi caso, no ha sido necesario invertir económicamente para cargar los datos a MongoDB ya que se me ha proporcionado acceso a un servidor de base de datos con el suficiente almacenamiento para concluir mi investigación.

En primer lugar, se evaluarán los tiempos empleados en generar las estructuras en R, convertirlas a JSON y cargarlas a MongoDB. Posteriormente se expondrán los resultados y conclusiones obtenidas sobre cada uno de los tipos de consultas a las colecciones de la base de datos.

3.3.3.1 Evaluación de los tiempos en generar y cargar las estructuras a MongoDB

Los tiempos en generar y cargar la información a MongoDB se han calculado gracias a la función *Sys.time()*. No pertenece a ninguna librería y su propósito es obtener el tiempo del sistema en un momento específico. Calculando ese tiempo al principio de la ejecución y al final para posteriormente restar los resultados, apoyándonos en la librería *hms* de R [28], se obtiene el tiempo que se ha demorado en ejecutar cierta operación.

Los tiempos obtenidos dependerán del equipo en el cual se esté ejecutando el script. La memoria RAM, el procesador y el número de procesadores lógicos influyen en los tiempos obtenidos. Las pruebas se han hecho desde un equipo con 16 GB de RAM, procesador Intel i7-1165G7 (11th Gen) y 8 procesadores lógicos. También influye el plan de MongoDB Atlas que se haya elegido ya que unos ofrecen más velocidad de lectura y escritura que otros.

Se recuerda que el tiempo en generar la estructura es la suma del tiempo en cargar los datasets a R, organizar las entidades por sus atributos y establecer los diferentes tipos de relaciones entre las entidades principales de cada área. También se tiene en cuenta el posible tiempo empleado en ajustar la solución.

A continuación se muestra un gráfico de barras donde se distinguen los tiempos empleados para realizar las 3 diferentes tareas en las 4 diferentes estructuras implementadas

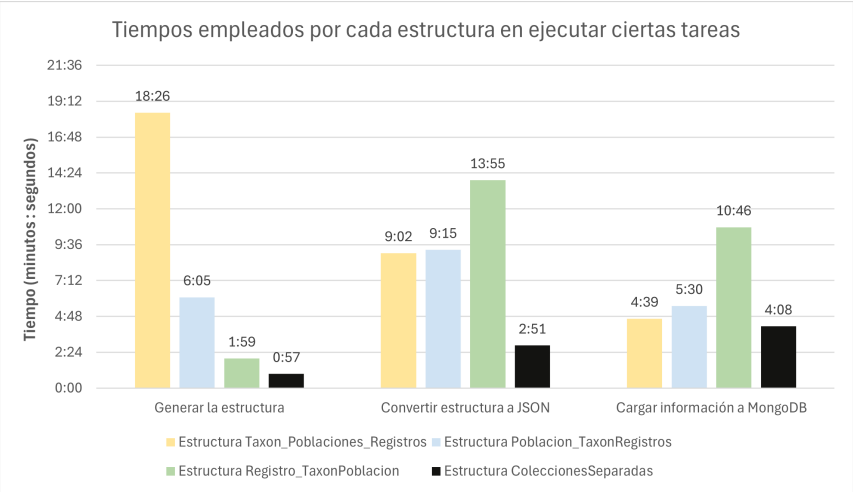

Figura 3.13: Medición de tiempos para las diferentes tareas de cada estructura

Para generar la estructura, se aprecia una gran diferencia de tiempos en la estructura *Taxon_Poblaciones_Registros* con respecto al resto. Esto se debe al proceso de división de aquellos documentos que en la base de datos pesarían más de 16 MB. También destacar que el tiempo es mayor en aquellas estructuras en las que se implementan relaciones 1:N debido al uso del comando *split*. Por esto, generar la estructura *Poblacion_TaxonRegistros* demora más tiempo que generar la estructura *Registro_TaxonPoblacion* a pesar de que esta última presenta más información debido a la redundancia de sus datos. Generar la estructura para *ColeccionesSeparadas* demora apenas un minuto, tiempo que únicamente se dedica a cargar los datasets y generar las entidades. No hay que implementar relaciones entre ellas.

Tal y como se comentaba con anterioridad, transformar las estructuras a JSON requiere de paralelización. De lo contrario, los tiempos se verían disparados. La estructura que más tiempo demora es *Registro_TaxonPoblacion*, principalmente por su gran tamaño debido a la gran cantidad de redundancia en sus datos. La estructura *ColeccionesSeparadas* de nuevo vuelve a ser la que mejores resultados presenta.

Para cargar la información a la base de datos, de nuevo la estructura *Registro_TaxonPoblacion* vuelve a ser la que peores resultados ofrece. El resto presenta tiempos similares siendo de entre estas la peor *Poblacion_TaxonRegistros*, debido principalmente a que tiene que insertar más información que las otras 2 porque también presenta redundancia en los datos.

En resumen, sumando los tiempos de cada tarea en cada estructura se consigue el tiempo total en ejecutar cada script. Tras ejecutar el script, se tendrá en la base de datos la información cargada, organizada según la estructura correspondiente. Los tiempos totales en ejecutar cada script se recogen en la siguiente tabla:

Estructura	**Tiempo Total (hh:mm:ss)**
Taxon_Poblaciones_Registros	00:32:07
Poblacion_TaxonRegistros	00:20:50
Registro_TaxonPoblacion	00:26:40
ColeccionesSeparadas	00:07:56

Tabla 3.3: Tiempos totales de ejecución para diferentes estructuras

Comparando los tiempos, la estructura *ColeccionesSeparadas* es la que mejores ofrece. Falta por evaluar los tiempos de consulta y el espacio resultante en la base de datos para elegir cual de las 4 estructuras implementadas es la mejor

3.3.3.2 Evaluación del espacio resultante en la base de datos

A continuación se muestra una tabla con el tamaño correspondiente a cada colección almacenada en la base de datos. Para calcular dichos tamaños se ha empleado la shell Mongosh, usando los siguientes comando:

Calcular espacio total en disco: *db.NombreColección.dataSize() / (1024*1024)*

Calcular espacio comprimido: *db.NombreColección.stats().totalSize / (1024*1024)*

Colección	**Tamaño total(MB)**	**Tamaño comprimido(MB)**
Taxon_Poblaciones_Registros	1449.1605	299.1758
Poblacion_TaxonRegistros	1860.4491	387.3008
Registro_TaxonPoblacion	3431.1258	694.2500
colSepInformacionTaxonomica	0.3441	0.1289
colSepPoblacion	313.9293	80.7695
colSepRegistroDeOcurrencia	1149.5453	331.9766

Tabla 3.4: Espacio de almacenamiento de las diferentes colecciones en MongoDB.

Como se aprecia en la tabla, se distinguen 2 tipos de tamaños. Por un lado el tamaño total que ocupan los datos de cada colección en disco, sin considerar la compresión de los datos. Por otro lado, el tamaño de esa colección una vez se han empleado las técnicas de compresión de los datos. Este tamaño se tiene en cuenta a la hora de almacenar la información si se utiliza MongoDB Atlas o incluso un servidor de MongoDB configurado para utilizar compresión.

Analizando los datos obtenidos, se concluye que la peor en términos de espacio de almacenamiento es la colección *Registro_TaxonPoblacion*. Esto se debe principalmente a la gran cantidad de repetición de los datos. Se recuerda que en esta colección, los datos sobre las poblaciones y los taxones se encuentran repetidos en los documentos, sobre todo estos últimos.

Teniendo en cuenta el espacio total en disco, se observa que la colección *Taxon_Poblaciones_Registros* ocupa aproximadamente la suma de los pesos totales en las colecciones *colSepInformacionTaxonomica*, *colSepPoblacion* y *colSepRegistroDeOcurrencia* ya que no se presenta nada de redundancia en los datos. Sin embargo, llama la atención que tras realizar la compresión de la información, este patrón no se cumple.

Sin duda esto tiene que ver con la forma en la que Mongo comprime la información para almacenarla en la base de datos. Investigando sobre el tema en la documentación oficial, se habla de que el motor de almacenamiento predeterminado empleado es *WiredTiger* [29]. El motor tiene la responsabilidad de administrar

la forma en la que se almacenan los datos. Este motor hace uso de manera predeterminada de la librería *Snappy* [30] para comprimir y descomprimir los datos que se almacenan en la base de datos.

No se ha llegado a una conclusión certera de los motivos por los que este patrón no se cumple, pero todo apunta a cómo *Snappy* comprime de manera eficiente este tipo de documentos, documentos grandes con estructuras largas (listas) con patrones repetitivos (lista de poblaciones, las cuales tienen una estructura similar). En el caso de la colección *colSepRegistroDeOcurrencia*, esta ocupa más incluso que *Taxon_Poblaciones_Registros* tras la compresión de los datos. En este caso, *Snappy* no es tan eficiente comprimiendo la información.

En resumen, la colección *Taxon_Poblaciones_Registros* es la que menos espacio ocupará si se emplea MongoDB Atlas o un servidor configurado para la compresión de los datos. En caso contrario, deberíamos fijarnos en el tamaño total ocupado por los datos en disco.

3.3.3.3 Evaluación de las consultas a la base de datos

Para evaluar los tiempos de respuesta ante determinadas consultas, se ha tenido en cuenta una lista de consultas frecuentes proporcionada de antemano a la realización de este trabajo de fin de grado. En ella se distinguen 3 tipos de consultas, a nivel de taxón, población y registro de ocurrencia. Para cada una de las consultas, las cuales están a su vez divididas en varias preguntas, se ha implementado su *query* correspondiente en MongoDB. Las consultas y su implementación correspondiente se pueden encontrar en el documento *ListaConsultas.pdf* adjunto a esta memoria.

Cada uno de los gráficos de barras que se presentarán a continuación muestran el tiempo empleado en recibir respuesta ante las consultas implementadas en las diferentes colecciones. Los tiempos difieren entre las colecciones porque la forma de acceder a la misma información es distinta. Recordar tener en cuenta que dentro de una misma consulta se pueden distinguir diferentes preguntas, por eso la notación de "C" para consulta y "P" para pregunta.

1. Consultas frecuentes a nivel de taxón

Se dice que es a nivel de taxón porque se pregunta por información que guardan en primera instancia los taxones. Un ejemplo de consulta sería la siguiente:

¿Qué especies son los parientes silvestres del trébol rojo (nombre científico: Trifolium pratense)? ¿A qué genepool pertenece cada uno de ellos?

Los tiempos de respuesta ante las consultas a nivel de taxón son los siguientes:

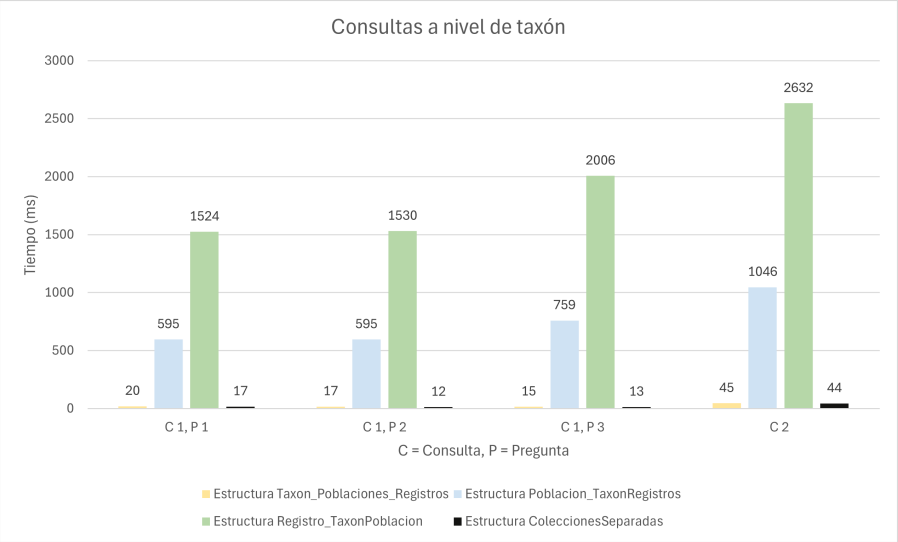

Figura 3.14: Tiempos de respuesta ante las consultas a nivel de taxón

Tras los resultados obtenidos se aprecia que las colecciones *Taxon_Poblaciones_Registros* y *colSeparadaInformacionTaxonomica* son las que mejores tiempos de respuesta ofrecen. La rapidez de este tipo de consultas en estas 2 colecciones con respecto al resto se debe principalmente a dos factores. El primero es que la información está a primera mano y el segundo es que para buscar información sobre un taxón por ejemplo en la colección *Poblacion_TaxonRegistros* tenemos que buscar dentro de cada uno de los 611611 documentos. Sin duda, no es nada favorable buscar información sobre taxones en este tipo de colecciones ya que, además de estar buscando entre muchos más documentos, la repetición de los datos nos dificulta obtener la respuesta esperada.

De entre estas dos, hay una de ellas presenta mejores resultados a nivel práctico. Como se comentaba con anterioridad, la colección *Taxon_Poblaciones_Registros* presenta documentos con información repetida fruto de las divisiones que se tuvieron que realizar para evitar que ningún documento de la base de datos superara los 16 MB en tamaño. Esto supone que también se deba tener en cuenta este factor a la hora de realizar consultas, invirtiendo esfuerzos en controlar la forma en la que se recuperan los datos.

En conclusión, la mejor colección para realizar las consultas a nivel de taxón es a *colSeparadaInformacionTaxonomica*, colección que almacena todos los taxones en la estructura *coleccionesSeparadas*.

2. Consultas frecuentes a nivel de población

Se dice que es a nivel de población porque se pregunta por información que guardan en primera instancia las poblaciones. Un ejemplo de consulta sería la siguiente:

¿Cuántas poblaciones de Lupinus angustifolius están recogidas en la base de datos? ¿Cuántas de ellas se encuentran al norte del paralelo 40?

Los tiempos de respuesta ante las consultas a nivel de población son los siguientes:

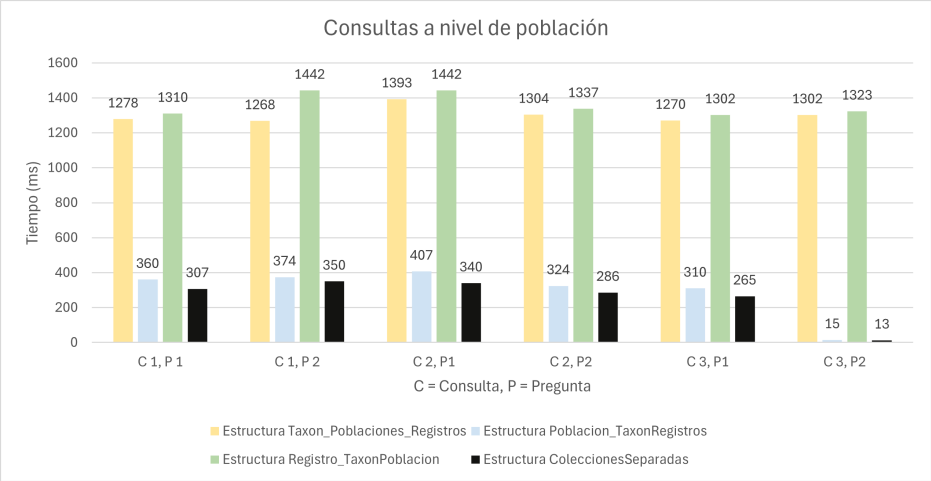

Figura 3.15: Tiempos de respuesta ante las consultas a nivel de población

Las colecciones que ofrecen mejores resultados son *Poblacion_TaxonRegistros* y *colSeparadaPoblacion*, correspondiente a la colección que almacena todas las poblaciones en la estructura *coleccionesSeparadas*. Esto, al igual que antes se debe a que la información se encuentra a primera mano, la *query* que hay que implementar es muy sencilla. En las colecciones *Registro_TaxonPoblacion* y *Taxon_Poblaciones_Registros* es necesario indagar entre los objetos *Poblacion* de cada documento, implementando así una *query* más compleja (haciendo uso de *aggregates*), lo que conlleva un aumento de tiempo notable. Además, en la colección *Registro_TaxonPoblacion* hay que tener en cuenta la repetición de los datos, siendo esta la que peores resultados ofrece.

3. Consultas frecuentes a nivel de registro de ocurrencia

Se dice que es a nivel de registro de ocurrencia porque se pregunta por información que guardan en primera instancia los registros de ocurrencia. Un ejemplo de consulta sería la siguiente:

¿En qué herbarios puedo encontrar registros de Lupinus angustifolius? ¿Cuáles son los identificadores de especímenes de herbario que nos permitirían localizarlos en dichos herbarios?

Los tiempos de respuesta ante las consultas a nivel de registro de ocurrencia son los siguientes:

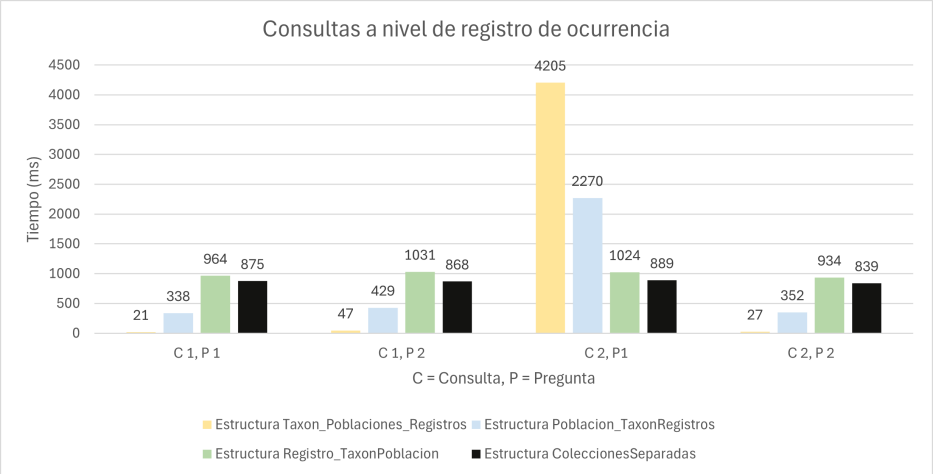

Figura 3.16: Tiempos de respuesta ante las consultas a nivel de registro de ocurrencia

Siguiendo el patrón que estamos viendo con los otros tipos de consulta, se puede pensar que las mejores colecciones para realizar consultas a nivel de registro de ocurrencia son a *Registro_TaxonPoblacion* y a *colSeparadasRegistroDeOcurrencia* ya que presentan la información de los registros de ocurrencia en primera instancia. En parte esto es así, ya que esta última presenta resultados aceptables (no tan buenos como en las otras colecciones con información de área separadas ya que esta posee 1806711 documentos). Sin embargo, para la colección *Registro_TaxonPoblacion* los tiempos de respuesta para consultas simples son peores incluso que en el resto de colecciones, algo que hasta entonces no pasaba. Para consultas complejas (las que usan *aggregates*), esto cambia y la colección *Registro_TaxonPoblacion* presenta resultados mejores, devolviendo la información en algunas de las pruebas en un tiempo de 1024 ms con respecto a los 4205 ms que se demora la misma consulta en la colección *Taxon_Poblaciones_Registros*.

Dicho todo esto, observando y comparando los tiempos de respuesta para las diferentes colecciones, se concluye que las mejores colecciones a las que consultar información relacionada con los registros de ocurrencia son a *Registro_TaxonPoblacion* y a *colSeparadasRegistroDeOcurrencia* ya que los tiempos de respuesta se mantienen estables, independientemente de si se están implementando consultas simples o complejas.

Conclusión

Viendo los resultados obtenidos para los diferentes tipos de consultas, en todas ellas se llega a la conclusión de que una de las mejores colecciones a las que consultar información es aquella perteneciente a la estructura *coleccionesSeparadas*. En todas las ocasiones representa la opción que menos tiempo de respuesta proporciona, junto con la colección que guarda en primera instancia la información relativa a su área. Esto se debe a que la información que se quiere obtener se

presenta en primera instancia en aquella colección en la que se encuentre, no se presenta repetición en los datos ni se debe indagar entre los documentos y listas embebidas que pueda poseer el documento en el que se está buscando. Además, los documentos de las colecciones no son pesados, por lo que la base de datos funcionará de una manera más fluida.

En resumen, esta es la mejor estructura a implementar para realizar todo tipo de consultas. Además, gracias a cómo está estructurada la información en cada una de las colecciones, se pueden realizar consultas que involucren las diferentes colecciones. Esto se debe a que las distintas colecciones poseen atributos que referencian la información en el resto de colecciones. A continuación se muestra un ejemplo visual de lo que se comenta.

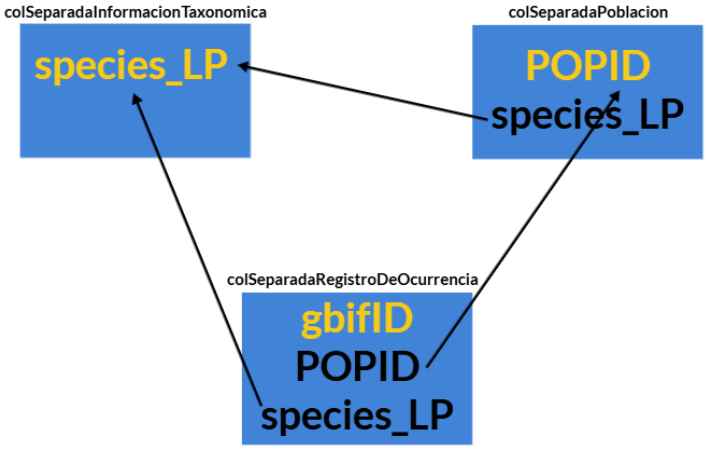

Figura 3.17: Referencias entre colecciones

A pesar de que en el diseño conceptual de la base de datos, las entidad *Poblacion* no tiene información sobre su taxón correspondiente y la entidad *Registro de ocurrencia* no tiene tampoco información sobre su taxón y población correspondientes, en los datasets proporcionados, estos atributos se tienen en cuenta. Aprovechando esto, a la hora de construir las entidades en el proceso de generación de estructuras, se han añadido dichos atributos para disponer así de estas referencias.

Como un taxón puede pertenecer a N poblaciones o a N registros de ocurrencia, la única manera que hay de tener una referencia desde la colección *colSeparadaInformacionTaxonomica* a las demás es teniendo un atributo de tipo lista que almacena los identificadores de las poblaciones o registros a los que está asociado ese taxón. Esto es algo que no se ha implementado ya que se puede obtener la misma información consultando a las otras colecciones incluso de forma más sencilla y útil.

Un ejemplo de este tipo de consultas es el siguiente: *¿Cuáles son las poblacio-*

nes asociadas a la especie Lupinus angustifolius?

```
db.colSepPoblacion.find({ species_LP: "Lupinus angustifolius" });
```

Código 3.10: Consulta en MongoDB para recuperar todas las poblaciones asociadas a una especie

Accediendo directamente a la colección *colSeparadaPoblacion* se obtiene la información de manera rápida y sencilla. Si tuviéramos en la colección *colSeparadaInformacionTaxonomica* una lista con las referencias a las poblaciones, en primer lugar tendríamos que consultar esa lista y, por cada elemento (identificador de población, POPID) tendríamos que consultar la colección *colSeparadaPoblacion* para obtener así cada una de las poblaciones. De esta manera resultarían consultas más complejas, por lo que se ha decidido no implementar ese tipo de referencias.

Por último, veo necesario comentar que no se ha implementado un mecanismo de indexación de los documentos más que el que por defecto MongoDB introduce ("_id") al insertar documentos en la base de datos. Esto se debe a que la lista de consultas frecuentes no proporciona una visión clara sobre cuál o cuáles son los atributos por los que más se busca. El único atributo al que se recurre en numerables consultas es a *species_LP*, candidato a implementar su indexación. Para trabajos futuros, se abre la puerta a la creación de índices con el objetivo de disminuir los tiempos de respuesta para las consultas.

3.3.4. Despliegue

Con las conclusiones obtenidas en la etapa de evaluación, se llega a la conclusión de que la mejor estructura a implementar es *coleccionesSeparadas* a pesar de que no sea la que menos espacio de almacenamiento (comprimido) ocupa. Sabiendo esto, tenemos todas las herramientas para pasar a producción dicha configuración.

Se recuerda que para pasar a producción, es necesario disponer de una cuenta en MongoDB Atlas si se quieren alojar los datos en la nube de Mongo. Además se debe elegir el plan que más se amolde a nuestras necesidades, teniendo en cuenta la velocidad de lectura y escritura que se necesita. A nivel de espacio, con la versión gratuita es suficiente para almacenar toda la información ya que se dispone de 512 MB gratuitos de almacenamiento. En caso de no querer almacenar los datos en MongoDB Atlas, se puede configurar un servidor dedicado privado, lo cual permite tener un control completo sobre la base de datos y su seguridad.

Una vez se disponga de un lugar en el que almacenar los datos, en primer lugar hay que ejecutar el script de Python con los parámetros adecuados: URL de conexión a Mongo, nombre de la base de datos y nombre de la colección. En este

caso, hay que especificar el nombre de la colección como *ColeccionesSeparadas*.

Una vez el script de Python está en ejecución, se debe ejecutar el script de R *ColeccionesSeparadas*. Recordar que para ejecutar el script, se debe modificar la ruta que referencia la carpeta en las que se alojan los datasets a leer. Bastará con abrir el archivo y realizar una pequeña modificación donde se indica (con comentarios). Se puede modificar también el nombre de las colecciones de la base de datos modificando este fichero, justo en las 3 primeras líneas. Si no se modifica, las colecciones se guardarán con los nombres de *colSepInformacionTaxonomica*, *colSepPoblacion* y *colSepRegistroDeOcurrencia*. Para ejecutar el script, se puede hacer de diferentes maneras, se puede ejecutar desde un entorno de desarrollo integrado (IDE) como RStudio o directamente desde la terminal con el siguiente comando: *Rscript nombreScript.R*.

Tras ejecutarse el script de R, la información estará cargada en MongoDB y lista para su lectura. Se recuerda que en el apéndice B de este documento se encuentra una guía completa para configurar el entorno y cargar la información a la base de datos.

4

Conclusiones y trabajos futuros

Tras la finalización del trabajo se puede concluir que se ha cumplido el objetivo principal del proyecto, implementar una base de datos en la que se almacenará información sobre el Inventario Nacional de Parientes Silvestres de los Cultivos (PSC) y Plantas Silvestres de Uso Alimentario (PSUA) de España. Una de las condiciones principales es, que tanto la construcción como el propio funcionamiento de la base de datos sea óptimo. Esto sin duda se ha cumplido ya que los tiempos empleados en generar, cargar a Mongo y realizar consultas son razonablemente buenos.

Además, tras concluir que la estructura *ColeccionesSeparadas* es la mejor de todas las posibles a implementar, disponemos de una base de datos rápida, ya que el tamaño de cada documento no es grande y los tiempos de respuesta para las consultas son bajos. También disponemos de una base de datos sin redundancia en los datos, no hay ni un solo dato que se repita.

Gracias al proceso de refinamiento de los datos y limpieza en el proceso de carga a MongoDB, toda la información que se almacenará en la base de datos será relevante. Por otra parte, las modificaciones realizadas sobre ciertos atributos han aumentado la funcionalidad de la base de datos, pudiendo realizar consultas que anteriormente no se podían realizar.

Si bien optimizar los tiempos empleados en la generación de estructuras en R o la carga de la información a MongoDB es algo a tener en cuenta, esto es un proceso que se realiza una única vez. Una vez se haya realizado esto y los datos estén en la base de datos, es de vital importancia que las consultas que se hagan a ella consuman el menor tiempo posible, consiguiendo así una base de datos

rápida y eficaz. Por lo tanto, para elegir la estructura de la base de datos óptima se ha priorizado por encima de todo los tiempos de respuesta de las consultas.

Para un trabajo de investigación futuro, sería buena idea implantar un sistema de indexación con el objetivo de mejorar los tiempos de respuesta ante las consultas. Debido a que no se pueden indexar todos los campos por problemas de espacio, será necesario una lista de consultas frecuentes más específica o solicitudes directas para implementar índices en los atributos que más se utilicen para las búsquedas. Uno de ellos podría ser el atributo *species_LP* ya que se recurre a él en numerosas ocasiones. Dado el script de Python implementado en este proyecto, bastará con añadir una condición que, dependiendo de la colección elegida, cree los índices necesarios con el comando *db.nombreColeccion.create_index()*.

El contexto del proyecto requería la creación de una base de datos que siguiera los principios FAIR (Localizable, Accesible, Interoperable y Reutilizable). Esto se ha logrado en la medida de lo posible y abre las puertas a trabajos futuros para el correcto y completo cumplimiento de estos principios. En primer lugar, se podría completar el cumplimiento del primer principio (Findable) introduciendo metadatos descriptivos sobre los datos. En cuanto al segundo principio (Accessible), una vez se implemente la interfaz web para que los usuarios puedan acceder a la información del Inventario Nacional, se cumplirá. Sobre el tercero (Interoperable), la adición de metadatos con un lenguaje formal, accesible y compartible contribuirá a su cumplimiento. Por último, (Reusable) estableciendo políticas que definan las condiciones bajo las cuales los datos pueden ser utilizados y compartidos. Todos estos principios van más allá de las sugerencias comentadas, por lo que en trabajos futuros se pueden invertir esfuerzos en cumplir cada uno de ellos.

Otra de las posibles mejoras a implementar en trabajos futuros es dar la capacidad al usuario de elegir el nombre que tendrá la colección o colecciones que guardarán la información en la base de datos. Ya sea a través de la ejecución del script de R o Python.

En resumen, los objetivos establecidos se han cumplido y se ha proporcionado una buena solución para colaborar en la Estrategia Nacional de conservación y utilización de parientes silvestres de los cultivos (PSC) y plantas silvestres de uso alimentario (PSUA).

Bibliografía

[1] "Sitio oficial de r," https://www.r-project.org/.

[2] "Sitio oficial de mongodb," https://www.mongodb.com/.

[3] T. van Hintum and J. Iriondo, "Principles for the inclusion of cwr data in eurisco," p. 30, 2022. [Online]. Available: https://www.ecpgr.org/fileadmin/bioversity/publications/pdfs/Principles_for_the_inclusion_of_CWR_data_into_EURISCO_final_17.05.2022.pdf

[4] "Principios fair," https://datos.gob.es/es/noticia/principios-fair-buenas-practicas-para-la-gestion-y-adminis 2017.

[5] P. y. A. Ministerio de Agricultura, "Estrategia nacional de conservación y utilización de parientes silvestres de los cultivos y plantas silvestres de uso alimentario," https://www.mapa.gob.es/es/agricultura/temas/medios-de-produccion/semillas-y-plantas-de-vivero/fitogeneticos_agricultura_alimentacion/Estrategia_PSC_PSUA.aspx, 2022.

[6] EURISCO, https://eurisco.ipk-gatersleben.de/apex/eurisco_ws/r/eurisco/home.

[7] FAO, https://www.fao.org/plant-treaty/areas-of-work/global-information-system/es/.

[8] G. de España, "Ministerio de agricultura, pesca y alimentación," https://www.mapa.gob.es/es/.

[9] ECPGR, https://www.ecpgr.org/.

[10] URJC, "Etsii," https://www.urjc.es/etsii.

[11] "Documentación oficial de r," https://www.rdocumentation.org/.

[12] "Documentación oficial de mongodb," https://www.mongodb.com/docs/.

[13] Arvindpdmn, "Guía para el uso de mql," https://devopedia.org/mongodb-query-language, 2021.

[14] "Sitio oficial de mongodb atlas," https://www.mongodb.com/es/atlas.

[15] "Sitio oficial de mongodb compass," https://www.mongodb.com/products/tools/compass.

[16] "Documentación oficial de mongodb shell," https://www.mongodb.com/docs/mongodb-shell/.

[17] "Sitio oficial de vertabelo," https://vertabelo.com/.

[18] "Definición de json," https://es.wikipedia.org/wiki/JSON.

[19] "Documentación sobre la librería mongolite," p. 10, 2024. [Online]. Available: https://cran.r-project.org/web/packages/mongolite/mongolite.pdf

[20] "Capacidad máxima de 16 mb por documento en mongodb," https://www.mongodb.com/docs/manual/reference/limits/#bson-documents.

[21] "Definición de bson," https://es.wikipedia.org/wiki/BSON.

[22] "Documentación oficial de gridfs," https://www.mongodb.com/docs/manual/core/gridfs/.

[23] "Documentación oficial de la librería argparse," https://docs.python.org/3/library/argparse.html.

[24] "Documentación oficial de la librería flask," https://flask.palletsprojects.com/en/3.0.x/.

[25] "Documentación oficial de la librería pymongo," https://pymongo.readthedocs.io/en/stable/index.html.

[26] "Documentación oficial de la librería jsonlite," https://www.rdocumentation.org/packages/jsonlite/versions/1.8.8.

[27] "Documentación oficial de la librería furrr," https://www.rdocumentation.org/packages/furrr/versions/0.3.1.

[28] "Documentación oficial de la librería hms," https://www.rdocumentation.org/packages/hms/versions/1.1.3.

[29] "Documentación oficial sobre el motor wiredtiger," https://source.wiredtiger.com/mongodb-5.0/compression.html.

[30] "Documentación oficial de snappy," https://google.github.io/snappy/.

Apéndices

A

Adjuntos a este documento

Debido al considerable tamaño de los archivos desarrollados durante el proyecto, se adjuntarán por separado junto a este documento. Los adjuntos serán los siguientes:

A.1. Scripts de R implementados

Serán 4, cada uno corresponde a una estructura implementada. Reciben los nombres de *Taxon_Poblaciones_Registros*, *Poblacion_TaxonRegistros*, *Registro_TaxonPoblacion* y *ColeccionesSeparadas*.

A.2. Script de Python implementado

Script encargado de recibir el JSON generado por el script de R, eliminar la información irrelevante y cargar el resultado a MongoDB. Recibe el nombre de *main.py*.

A.3. Listado de consultas a la base de datos

Para cada tipo de consulta de la lista de consultas frecuentes se distinguen diferentes consultas, cada una de ellas ha sido implementada en las diferentes colecciones generadas ya que es diferente la forma de acceder a la misma información en cada una de ellas. Este documento recibe el nombre de *ListaConsultas.pdf*.

B

Guía para la configuración y carga de información a MongoDB

A continuación se muestra una guía completa en la que se explica paso por paso cómo generar las estructuras y cargarlas a MongoDB. Se tomará como ejemplo la estructura *ColeccionesSeparadas*.

B.1. Descargar y preparar los adjuntos

- Descargar el adjunto *main.py*
- Descargar el adjunto *ColeccionesSeparadas*. Acto seguido se debe acceder a él y modificar la ruta en la que se encuentran los 3 datasets que guardan la información que se va a utilizar. También (voluntario), se puede cambiar el nombre de las colecciones que se crearán en la base de datos. Estas modificaciones se hacen al principio del documento, está explicado con comentarios dónde hay que realizar estos cambios. Por último, se debe guardar el archivo y cerrarlo.

B.2. Preparar el entorno

Para ejecutar el script de Python, será necesario configurar el entorno.

- Instalar Python. Se puede instalar desde el sitio oficial. Pulsando aquí.
- Instalar las dependencias necesarias:
  Abrir la terminal y ejecutar el siguiente comando: *pip install flask pymongo*

B.3. Ejecutar

- Lo primero que se debe ejecutar es el script de Python. Los pasos son los siguientes:
  - Abrir la terminal
  - Navegar a la carpeta donde se encuentre el script
  - Ejecutar el script con los parámetros correctos, un ejemplo de ejecución es el siguiente:
    python main.py –mongo_url mongodb+srv://usernameMongo:passwordMongo@cluster0.mtop1ql.mongodb.net/ –db nombreDB –collection ColeccionesSeparadas
    Ante cualquier duda, se puede ejecutar el comando de ayuda:
    python main.py -h
- Con el script de Python en ejecución, se debe ejecutar el script de R, hay diferentes maneras:
  - Ejecutar desde un IDE como RStudio. Abrir el archivo en dicho entorno de desarrollo y pulsar el botón *Source* en la parte superior.
  - Ejecutarlo directamente desde la terminal, usando el comando:
    Rscript ColeccionesSeparadas.R.

Seguidos estos pasos, solo toca esperar a que el proceso termine. Tras su finalización, la información estará organizada y cargada correctamente en la base de datos. En este caso, se habrán cargado 3 colecciones diferentes.