Escuela Politécnica Superior de Jaén
Máster Universitario en Ingeniería Informática

UNIVERSIDAD DE JAÉN
Escuela Politécnica Superior (Jaén)

Trabajo Fin de Máster

SERVIDOR PARA ALMACENAMIENTO DE MODELOS DE NUBES DE PUNTOS EN TERAESCALA CON DISTRIBUCIÓN ARBITRARIA EN EL ESPACIO Y TIEMPO

Alumno
Juan Antonio Béjar Martos

Tutores
Prof. D. Antonio Jesús Rueda Ruiz
(Departamento de Informática)

Prof. D. Carlos Javier Ogayar Anguita
(Departamento de Informática)

Junio, 2021

Universidad de Jaén

Escuela Politécnica Superior de Jaén

Departamento de Informática

Don Antonio Jesús Rueda Ruiz y Don Carlos Javier Ogayar Anguita , tutores del Trabajo de Fin de Máster titulado: ‘Servidor para almacenamiento de modelos de nubes de puntos en teraescala con distribución arbitraria en el espacio y tiempo’, que presenta Juan Antonio Béjar Martos, autoriza su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, junio de 2021

El alumno:

BEJAR MARTOS JUAN ANTONIO - 77646082Z

Firmado digitalmente por BEJAR MARTOS JUAN ANTONIO - 77646082Z
Fecha: 2021.06.03 11:05:22 +02'00'

Juan Antonio Béjar Martos

Los tutores:

Firmado digitalmente por RUEDA RUIZ ANTONIO JESUS - 33395227D
Fecha: 2021.06.03 11:43:57 +02'00'

OGAYAR ANGUITA CARLOS JAVIER - 26039170L

Firmado digitalmente por OGAYAR ANGUITA CARLOS JAVIER - 26039170L
Fecha: 2021.06.03 11:35:48 +02'00'

Antonio Jesús Rueda Ruiz

Carlos Javier Ogayar Anguita

Contenidos

Índice de ilustraciones

Índice de tablas

1- Introducción

El siguiente documento constituye la documentación del Trabajo de Fin de Máster “Servidor para almacenamiento de modelos de nubes de puntos en teraescala con distribución arbitraria en el espacio y tiempo”. La estructura de capítulos y apartados de este documento se ha realizado partiendo de múltiples de las bases y recomendaciones que establece la norma UNE 157801:2007 – “Criterios Generales para la elaboración de proyectos de Sistemas de Información”.

2- Objetivo del proyecto

El objetivo principal del proyecto es la implementación de un repositorio de carácter general para modelos basados en nubes de puntos y el diseño de un modelo de datos y una API que definan su comportamiento. Este objetivo principal se ha desglosado en los siguientes objetivos secundarios:

1. Diseño de una API genérica y flexible que defina operaciones para la inserción y consulta de modelos basados en nubes de puntos. La API debe permitir acceder a distintos niveles de detalle dentro de un modelo y ser capaz de organizar modelos en cualquier parte del territorio, teniendo en cuenta consideraciones de solapamiento espacial y permitir diferenciar distintas nubes de puntos en base a la dimensión temporal.

2. Desarrollo de un prototipo de aplicación de tipo backend que implemente dicha interfaz, siendo capaz de procesar modelos de nubes de puntos de distintos tamaños y con una capacidad de almacenamiento total en teraescala.

3. Diseño de una estructura de bases de datos que almacene la información provista por el servidor de forma eficiente y con capacidad para escalado horizontal (en un sistema distribuido).

4. Experimentación y adquisición de resultados a través de la implementación de un cliente sencillo que permita interactuar con el servidor.

3- Antecedentes y estado del arte

En la siguiente sección se pretende presentar los aspectos más relevantes que sirven de base para el trabajo realizado en este proyecto, introduciendo algunos de los fundamentos teóricos de los modelos de nubes de puntos, ámbitos en los que se emplean, tecnologías utilizadas para su captura, estándares y formatos empleados para su registro en ficheros, estructura de datos y enfoques empleados en su procesamiento así como su relación con el Big Data y algunas propuestas existentes para su almacenamiento.

3.1 - Nubes de puntos

Una nube de puntos es un conjunto de puntos distribuidos en un espacio tridimensional, sin un orden o relación entre ellos, identificados por su posición (X, Y, Z) y una serie de atributos adicionales, como el color (R, G, B), vectores normales, la intensidad o la fecha de adquisición. (Cao 2019)

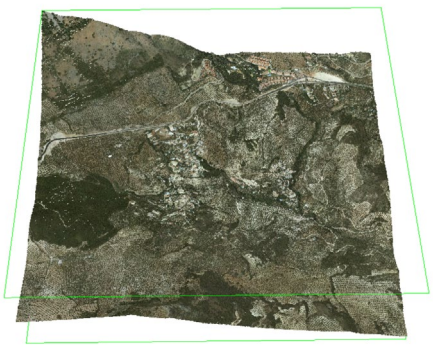

Ilustración 1. Nube de puntos adquirida en la provincia de Jaén

Las nubes de puntos se han consolidado como una herramienta de gran utilidad en múltiples disciplinas. Pueden emplearse tanto para obtener información de elementos de muy reducido tamaño hasta grandes masas territoriales. Dentro de este trabajo, cabe destacar su aplicación en aquellos ámbitos donde se pretende utilizar la nube de puntos como un método de representación de información geográfica. (Florent, 2019)

1. Ingeniería civil: orientada a labores de construcción de edificios y obras públicas, estudios demográficos y detección de cambios en entornos urbanos.

2. Ingeniería forestal y ambiental: permitiendo analizar situaciones de deforestación, planificación de cultivos, monitorización de catástrofes medioambientales o cambios en la costa debidos al cambio climático entre muchos otros casos.

3. Arqueología: las nubes de puntos suponen una herramienta ideal para preservar digitalmente el patrimonio (de una manera inocua con respecto al bien original), planificar restauraciones e intervenciones orientadas a la conversación y llevar a cabo reconstrucciones virtuales orientadas a la investigación o a su difusión entre el gran público.

Otros campos a destacar pueden ser la realidad virtual, donde se utilizan nubes de puntos como alternativa a mallas poligonales, o la navegación automática en vehículos, ya sea para que el propio vehículo pueda generar un modelo de sus alrededores como para suministrar información a este sobre carreteras etiquetadas con información de señales de tráfico. (Sugimoto, 2017, Dong, 2017)

3.2 - Tecnologías LiDAR y el estándar LAS

La tecnología LiDAR consiste en el uso de un láser que dispara haces de luz hacía la superficie del modelo a escanear. Estos haces son reflejados por el modelo, y a su vez, capturados por un sensor alojado en el dispositivo de captura. De este modo, es posible ubicar el punto contra el cual rebotó el haz de luz en función del tiempo transcurrido y la posición desde la cual se disparó.

Dentro de los escaneos LiDAR encontramos una categorización entre terrestres o aéreos y entre estáticos o dinámicos. Las nubes de puntos utilizadas en sistemas geográficos habitualmente son adquiridas de forma aérea, ya sea en vehículos tripulados o drones. La aparición de estos últimos ha supuesto una reducción en la barrera de entrada para la adquisición de nubes de puntos desde un punto de vista económico y, en consecuencia, un incremento en el uso y número de nubes de puntos existentes a día de hoy.

Como resultado de un escaneo LiDAR, se obtiene una nube de puntos. Estas nubes de puntos se distribuyen en uno o más ficheros, por lo general en formatos binarios, siendo LAS el estándar más utilizado junto a su versión comprimida, LAZ. El estándar LAS fue propuesto por la *American Society for Photogrammetry and Remote Sensing* (ASPR), con el objetivo de definir un formato abierto aplicable por todos los proveedores. LAZ a su vez fue diseñado por Martin Isenburg con el objetivo de definir una versión optimizada y comprimida del estándar LAS, abordando el problema de espacio producido por unos dispositivos de escaneo que cada vez son capaces de generar ficheros con un mayor volumen de información y que pueden requerir grandes cantidades de almacenamiento en los equipos y dispositivos de memoria externos de los usuarios (Isenburg, 2013). Otros formatos a resaltar son SPD, PCD o HDF5.

Actualmente encontramos 4 versiones del estándar LAS: 1.1, 1.2, 1.3 y 1.4. La estructura general de un fichero LAS en el estándar 1.4 consta de los siguientes componentes:

1. *Public Header Block*: contiene información fundamental de la nube de puntos registrada, como el número de puntos presentes, coordenadas mínimas y máximas, fecha en la que fue generada, la versión del fichero o el formato de los puntos almacenados.

2. *Variable Length Records* (VLRs): son metadatos adicionales, que pueden contener información variada. Por ejemplo, encontramos el aquí el sistema de coordenadas utilizado para identificar los puntos. Este bloque puede tener una capacidad máxima de 65,535 bytes

3. *Point Data Records*: son los puntos en sí. Habrá tantos de ellos como el valor especificado en la cabecera del fichero.

4. *Extended Variable Length Records* (EVLRs): representa una solución al límite de tamaño del bloque VLR. Se añaden al final del fichero y carece de restricciones de tamaño, por lo que los metadatos adicionales que no tienen cabida en el bloque de VLR se añaden aquí.

Del mismo modo, también existen distintos formatos para los puntos, enumerados secuencialmente del 0 al 10. La inclusión de nuevos formatos de puntos ha ido ligada a la aparición de los distintos formatos de LAS. En la siguiente ilustración podemos ver los atributos que definen a un punto en el formato 0. Estos atributos son básicos y estarán presentes en el resto de formatos. Encontramos, por ejemplo, las coordenadas (X, Y, Z), la intensidad, el número de retorno (un único pulso LiDAR puede devolver más de un punto, este valor identifica el orden del punto dentro de la secuencia devuelta por el pulso emitido) o la clasificación (permite clasificar el punto dentro de unas categorías preestablecidas: vegetación, edificio, terreno, agua…).

Item	Format	Size	Required
X	long	4 bytes	*
Y	long	4 bytes	*
Z	long	4 bytes	*
Intensity	unsigned short	2 bytes	
Return Number	3 bits (bits 0 – 2)	3 bits	*
Number of Returns (given pulse)	3 bits (bits 3 – 5)	3 bits	*
Scan Direction Flag	1 bit (bit 6)	1 bit	*
Edge of Flight Line	1 bit (bit 7)	1 bit	*
Classification	unsigned char	1 byte	*
Scan Angle Rank (-90 to +90) – Left side	char	1 byte	*
User Data	unsigned char	1 byte	
Point Source ID	unsigned short	2 bytes	*

Ilustración 2 Atributos del formato de punto 0 en el estándar LAS

En los nuevos formatos se han incluido atributos adicionales como el color (tanto RGB como NIR), hora GPS y otras características fotométricas adicionales. (ASPRS, 2019).

3.3 - Big Data y modelos de nubes de puntos

La gestión de información geográfica se puede categorizar como un problema de Big Data. En concreto, las nubes de puntos cuentan cada vez con un mayor volumen de información, gracias a escaneos con mayores niveles de resolución y el aumento en número de metadatos que aportan información adicional sobre cada punto. Junto a la expansión en popularidad de la tecnología LiDAR, esto implica tener que gestionar un mayor número de nubes de puntos y con un mayor número de puntos cada una, con el matiz adicional de que provienen de distintas fuentes, utilizando distintos estándares de fichero, atributos o resolución. La importancia de la dimensión temporal en la monitorización de la información espacial supone además mantener modelos de una misma zona generados en distintos momentos, lo que aumenta exponencialmente la cantidad de información a gestionar.

Dentro del Big Data se definen 5 dimensiones conocidas como las 5 Vs: *volumen*, *variedad*, *valor*, *velocidad* y *veracidad*. En base a lo expuesto previamente, podemos ver como las dos primeras son una realidad actual en la gestión de la información de este tipo de modelos. El *valor* de estos modelos para representar información útil y relevante en múltiples disciplinas es cada vez mayor. La *velocidad* para procesar la información y poder suministrarla a los usuarios finales también es un problema real de este ámbito, con el objetivo de facilitar una toma de decisiones lo más rápida y efectiva posible. Por último, la *veracidad* de la información suministrada es crucial para que los usuarios finales no basen sus decisiones en datos erróneos o corruptos. (Evans, 2014)

Tradicionalmente los ficheros generados durante un escaneo LiDAR se almacenan en dispositivos de memoria primarios y secundarios, manteniendo, en el mejor de los casos, un sistema de estructura de directorios que permita organizarlos. Sin embargo, a medida que aumentan de tamaño y número estos datasets, el acceso y mantenimiento es más complejo, siendo difícil realizar consultas en base a criterios espaciales y temporales a la zona de interés, encontrándonos ante un problema de escalabilidad.

El uso de tecnologías orientadas a Big Data es necesario para poder dar una solución a estos inconvenientes, tanto a nivel de un modelo de programación asíncrono, reactivo, que facilite el paralelismo y haga un uso responsable de los recursos del sistema, como en el uso de sistemas de almacenamiento que permitan el escalado horizontal y accesos eficientes independientemente del volumen de información almacenado. Aquí encontramos distintos enfoques, pudiendo utilizar sistemas de archivos distribuidos (Kramer, 2015), o bases de datos, principalmente de tipo noSQL, pudiendo diferenciar en estas entre un almacenamiento a nivel de punto o usando un fichero de un formato como el LAZ (Boehm, 2014, Deibe, 2018). Este último enfoque puede ser el más interesante, al permitir una buena optimización del almacenamiento gracias a la compresión propia de LAZ junto a la versatilidad de un sistema de bases de datos.

3.4 - Estructuras de datos

Uno de los aspectos claves planteados en el proyecto es la distribución de los modelos de puntos en distintos niveles de detalle siguiendo un enfoque espacial, permitiendo acceder a zonas del espacio de forma eficiente y escalada.

Existen múltiples estructuras de datos utilizadas por distintos autores para almacenar modelos de nubes de puntos. La más extendida es el octree, utilizando enfoques out-of-core que se apoyen en memoria secundaria para procesar datasets de gran tamaño. Otras alternativas utilizadas son el Kd-tree, R-tree, estructuras basadas en vóxeles dispersos, grids o estructuras anidadas e híbridas que combinen una o más de las ya mencionadas.

En este trabajo se utilizará como estructura un octree sin redundancia donde todos los nodos podrán almacenar puntos, pudiendo de este modo ofrecer distintos niveles de detalle y facilitar la transmisión progresiva de información en una zona al mismo tiempo que se mantienen unos niveles espacio ocupado similares a los del dataset original, al quedar un punto asociado a un solo nodo de la estructura.

4- Requisitos iniciales

Para definir los requisitos iniciales, se ha hecho un desglose en requisitos funcionales y de calidad. Los requisitos funcionales definidos en esta sección tienen un carácter genérico y a lo largo de las primeras iteraciones del proyecto fueron concretándose, dando lugar a una API que define las acciones que un usuario puede llevar a cabo en el sistema, un modelo de datos que soporta dicha API y finalmente, una implementación de dicho modelo en la estructura de la base de datos. Se han identificado los siguientes:

- El sistema debe permitir a los usuarios almacenar modelos de nubes de puntos
- El sistema debe procesar un modelo de nubes de puntos, generando una estructura jerárquica que organice la información de esta.
- El sistema debe permitir a los usuarios realizar consultas sobre la estructura asociada a una nube de puntos y recuperar conjuntos de puntos de esta.
- El sistema debe permitir a los usuarios encontrar modelos de nubes de puntos en base a criterios espaciales.
- El sistema debe permitir a los usuarios encontrar modelos de nubes de puntos en base a criterios temporales.
- El sistema debe permitir a los usuarios encontrar modelos de nubes de puntos en base a identificadores nominales.

En cuanto a los requisitos de calidad, se han definido los siguientes:

- El sistema debe ser fácilmente escalable.
- El sistema debe de estar desacoplado del sistema de gestión de base de datos que se utilice en primera instancia, permitiendo probar nuevas alternativas en el futuro modificando solamente la capa de persistencia.
- El sistema debe dar cobertura a distintos formatos y estándares de nubes de puntos.

- El sistema debe mantener la integridad de los modelos de nubes de puntos transferidos.

5- Alcance

Al tratarse de un proyecto de investigación con una metodología ágil, el alcance del proyecto dependerá de los avances realizados dentro de los plazos de tiempo establecidos. Inicialmente, el objetivo que se plantea es dar soporte a todos los requisitos especificados en el capítulo 4- Requisitos iniciales, si bien puede variar en función del desarrollo y las dificultades encontradas a lo largo de este.

Como entregables del proyecto, se deberá aportar el código fuente del proyecto generado, así como un contenedor Docker que facilite su despliegue. A nivel de documentación, se entregará esta misma memoria, así como la documentación de la API diseñada en formato OpenAPI 2.0/3.0 de cara a que futuros clientes puedan acceder al servidor.

6- Estudio de alternativas y viabilidad

En esta sección se pretenden presentar las diferentes alternativas desde un punto de vista teórico y tecnológico sobre las que desarrollar el sistema planteado.

6.1- Servidor

Para la implementación de la aplicación servidor, buscamos tecnologías que funcionen permitan procesar grandes cantidades de información. Nos centraremos principalmente en tecnologías basados en la máquina virtual de Java (JVM).

La principal alternativa considerada y la que finalmente se ha utilizado en el proyecto es Spring. Se trata de un framework para Java que ofrece desde hace un tiempo el módulo conocido como Webflux, basado en el estándar Reactive Streams,

definiendo un modelo reactivo, asíncrono y no bloqueante. Esto nos posibilita primero de todo poder trabajar con grandes datasets, evitando problemas asociados al heap de la JVM y poder dar soporte a una transmisión de datos con el cliente de forma progresiva. También nos da herramientas para facilitar el paralelismo en la aplicación, pudiendo agilizar los tiempos en algunas tareas complejas como podría ser la inserción del dataset, así como una gran cantidad de módulos que podemos integrar en nuestra aplicación para facilitar tareas como la conectividad con bases de datos mediante Spring Data utilizando también un enfoque reactivo. Además, cuenta con una amplia documentación y existe una experiencia previa de desarrollo con el framework Spring, aunque con su versión no reactiva.

Existen otros frameworks que se ajustan a las especificaciones de Reactive Streams y que pueden ser considerados. Por ejemplo, encontramos Akka, un framework con soporte para Scala y Java con especial énfasis en el apartado de la concurrencia. Otra alternativa es Vert.x, con características similares. También caben destacar otras tecnologías clásicas en Big Data como Hadoop y Spark.

6.2 - Manejo de ficheros LAS/LAZ

Uno de los principales problemas detectados en el desarrollo de la solución del backend a través de Java es la existencia de librerías y herramientas para poder trabajar con ficheros LAS/LAZ. Como librerías de código abierto las opciones disponibles son laszip4j y JLas.

Nombre	Características
laszip4j	• En desarrollo • Solo operaciones de lectura • Compatibilidad total con versiones LAS/LAZ 1.1, 1.2 y 1.3 y parcial con 1.4

JLas	• Sin recibir actualizaciones desde 2016 • Solo operaciones de lectura • Compatibilidad con versiones LAS/LAZ 1.1, 1.2 y 1.3 • No soporta todos los formatos de punto presentes en dichas versiones

Tabla 1. Comparativa entre las librerías para ficheros LAS/LAZ disponibles en Java

En ambas las únicas operaciones disponibles son las de lectura, por lo que la escritura de ficheros LAS/LAZ implicaría la integración de otras herramientas externas al ecosistema Java. Además, la lectura de ficheros no tiene una cobertura total de todos los formatos disponibles en el estándar LAS en ninguna de las dos librerías, por lo que también es necesario utilizar algún tipo de herramienta para la lectura de ficheros en los que el formato no sea compatible. A pesar de todos los inconvenientes mencionados, se ha decidido utilizar laszip4j inicialmente en el sistema para las operaciones de lectura de ficheros que lo permitan. La librería nos ofrece una clase LASPoint que representa cada uno de los puntos leídos del fichero. Sin embargo, existe una carencia evidente en la escritura. Para solucionar este problema, podemos utilizar la librería Laszip, desarrollada en C++, y elaborar una librería externa a la que nuestra aplicación principal pueda llamar para las operaciones de escritura de fichero.

La integración se puede llevar a cabo de distintas maneras. Por ejemplo, se podrían escribir los puntos en algún formato ASCII o almacenarlos en la propia base de datos de forma temporal para que posteriormente, la librería pueda ser llamada con algunos argumentos que permitan identificar la información necesaria para generar el fichero LAZ. Sin embargo, la opción más eficiente pasaría por utilizar la librería *Java Native Interface* (JNI). El uso de JNI puede resultar complejo y difícil de acoplar con el resto del sistema. *Java Native Access* (JNA) es una librería que ofrece una alternativa a JNI siguiendo sus mismos fundamentos. La principal ventaja de JNA es una mayor facilidad de uso, permitiendo un mayor nivel de abstracción y simplificando el desarrollo. Sin embargo, el rendimiento es ligeramente inferior al de

JNI. Se plantea el uso de JNA en la primera versión del enfoque de desarrollo de la versión en memoria del octree, y en caso de que los resultados sean satisfactorios, realizar el desarrollo directamente en JNI para conseguir el mejor rendimiento posible.

Para el desarrollo de una implementación out-of-core, se plantea el uso de LASTools, basado de forma interna en Laszip. Se trata de una de las opciones más potentes para trabajar con ficheros LAS/LAZ. De este modo, el sistema se encargaría de ir lanzando las operaciones apropiadas como procesos del sistema operativo a través de LASTools, mientras que en memoria almacenaremos la información necesaria para poder calcular los parámetros adecuados con los que lanzar dichos procesos. Las funcionalidades a las que nos permite acceder son muy variadas y la gran mayoría quedan fuera del alcance de este proyecto, centrándonos principalmente en las siguientes:

- lasinfo: muestra información clave del fichero LAS/LAZ, como la cabecera y los VLRs y EVLRs.

- las2las: nos permite realizar múltiples transformaciones sobre un fichero LAS/LAZ, como, por ejemplo, realizar un sampling de los puntos presentes en el fichero, crear un nuevo fichero a partir del original quedándonos solo con aquellos puntos encuadrados dentro de una serie de coordenadas o reproyectar los puntos a un nuevo sistema de coordenadas.

- lasmerge: permite realizar la unión de múltiples ficheros en uno.

El uso de LASTools nos permite no depender de Laszip4j. Sin embargo, para algunas operaciones de lectura más sencillas, su uso puede ser excesivo. Por ejemplo, para extraer las coordenadas máximas y mínimas definidas en el header de un fichero LAZ, tendríamos que aplicar lasinfo y procesar el texto generado con alguna expresión regular hasta detectar las líneas en las que aparece esa información. Para dar el soporte total a todos los ficheros LAZ que Laszip4j no nos ofrece y limitar

LASTools a operaciones fundamentalmente de creación de ficheros, se podría escribir algún script sencillo en Python utilizando la librería Pylas, de tal forma que invocando el script correspondiente nos pudiera devolver la caja envolvente de un fichero directamente o algún otro atributo de la cabecera que necesitemos.

6.3- Persistencia

La persistencia de la información generada por el sistema puede ser enfocada de dos maneras distintas.

Por un lado, podemos realizar el almacenamiento a nivel de punto, definiendo un modelo que almacene tanto las coordenadas como los atributos adicionales. Puesto que los atributos variarán en función de cada nube de puntos este enfoque sería más interesante para bases de datos de tipo no estructurado. Aun así, este almacenamiento daría como resultado una cantidad enorme de filas/documentos y difícil de gestionar. Algunas bases de datos relacionales orientadas a información geoespacial como Oracle Spatial y PostGIS utilizan conjuntos de puntos que encapsulan en cada fila a un número determinado de estos, tratándolos como una nube de puntos de tamaño muy reducido y que a nivel conceptual no dista mucho de lo que sería un fichero LAZ.

El otro enfoque pasa por almacenar la información directamente en ficheros LAZ. Cabe aclarar que siempre que almacenemos ficheros en el estándar LAS lo óptimo es utilizar su versión optimizada, LAZ, para optimizar el espacio ocupado en el servidor y la transmisión de ficheros. Los ficheros se pueden almacenar en la base de datos como contenido binario, ya sea con alguna herramienta de la propia base de datos para almacenar los ficheros o como un atributo más dentro de una tabla/colección.

En cuanto al sistema de bases de datos, se podría plantear una solución basada en bases de datos relacionales utilizando *Reactive Relational Database Connectivity* (R2DBC), un proyecto que permite interactuar con bases de datos SQL de forma reactiva y que se ajusta a los frameworks expuestos en la sección 6.1-Servidor. De este modo podríamos interactuar con bases de datos relacionales de forma reactiva, ofreciendo además conectividad nativa con PostgreSQL y a su vez, con PostGIS al ser una extensión de dicha base de datos.

Sin embargo, la solución inicial escogida para llevar a cabo el proyecto se basa en el uso de bases de datos no relacionales. Puesto que estamos ante un problema de Big Data, la flexibilidad, escalabilidad y rendimiento de este tipo de sistemas de almacenamiento se presupone como una gran ventaja frente a las bases de datos relacionales. (Venkatraman, 2016)

Dentro de las bases de datos noSQL encontramos los siguientes tipos:

1. Bases de datos clave-valor: la información se almacena como pares clave-valor, indexando las claves en estructuras eficientes para su posterior recuperación. El valor almacenado puede ser información de cualquier tipo, sin estar restringido a un tipo de estructura. Dentro de esta categoría encontramos a Redis o Oracle NoSQL.

2. Bases de datos basadas en columnas: siguen una estructura tabular, donde cada fila es identificada por una clave y a su vez contiene una o más columnas. Las columnas son similares a pares clave-valor, donde la clave es el identificador de la columna y el valor el contenido a almacenar. Cassandra o Hbase son bases de datos basadas en columnas.

3. Bases de datos basadas en grafos: están basadas en teoría de grafos. Este enfoque facilita y mejora la eficiencia de acceso en modelos compuesto por un gran número de entidades y relaciones entre ellas, siendo los vértices las

entidades almacenadas y las aristas las propiedades que permiten identificar dichas relaciones. Neo4j es una base de datos basada en grafos.

4. Bases de datos basadas en documentos: representan una extensión de las bases de datos clave-valor donde el valor se representa como un documento codificado en un formato semiestructurado como JSON. En algunos casos como MongoDB, existe un elemento adicional de abstracción denominado colección, con el fin de agrupar documentos similares a nivel conceptual. Otras bases de datos de este tipo son DynamoDB o CouchDB.
   (Davoudian, 2018)

Desde Spring Data tenemos soporte para bases de datos NoSQL de cada una de las categorías: Redis, MongoDB, Cassandra y Neo4j. Para el proyecto se ha decidido utilizar MongoDB como la base de datos que gestionará la persistencia del sistema. Las razones principales que dan soporte a esta decisión incluyen el hecho de que es uno los productos más asentados dentro del paradigma noSQL y se posee experiencia previa trabajando con ella. Sin embargo, el trabajo está abierto a plantear en el futuro transiciones a otros sistemas de bases de datos, tanto noSQL como SQL con el objetivo de analizar los rendimientos con diferentes sistemas de bases de datos.

Mongo da soporte al escalado horizontal mediante *sharding*. Puesto que vamos a trabajar con una cantidad de datos muy elevada, es de esperar que en un futuro esta utilidad sea necesaria para distribuir la carga de almacenamiento entre distintas máquinas. No solo podemos distribuir distintas colecciones en distintas máquinas, sino que también podemos repartir una colección entre varios *shards*, entendiendo este término como un subconjunto de los datos a distribuir.

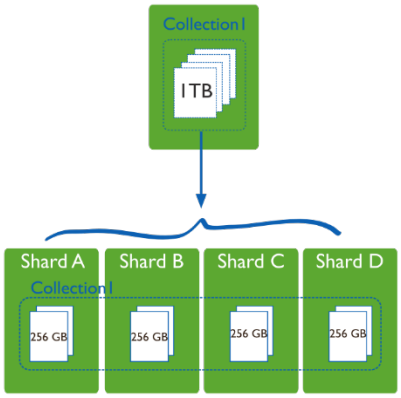

Ilustración 3. Sharding en MongoDB

7- Descripción y diseño de la solución elegida

En esta sección se resume, con el objetivo de facilitar la compresión del sistema, cuales son la arquitectura, modelo de datos y API Restful definidos finalmente en el sistema. El proceso de diseño ha seguido un enfoque iterativo y paralelo a la implementación, por lo que la información detallada a continuación muestra los resultados finales a los que se ha llegado tras las iteraciones presentadas en la sección 11- Desarrollo, en las que se describe de forma más concisa cómo se ha conseguido esta solución y los desafíos encontrados.

7.1- Arquitectura de la aplicación

A continuación, se describen cuáles han sido las tecnologías seleccionadas y la arquitectura general del sistema propuesta. Tendremos una aplicación principal, SPSLidar, implementada con Spring Webflux y segmentada en 3 capas:

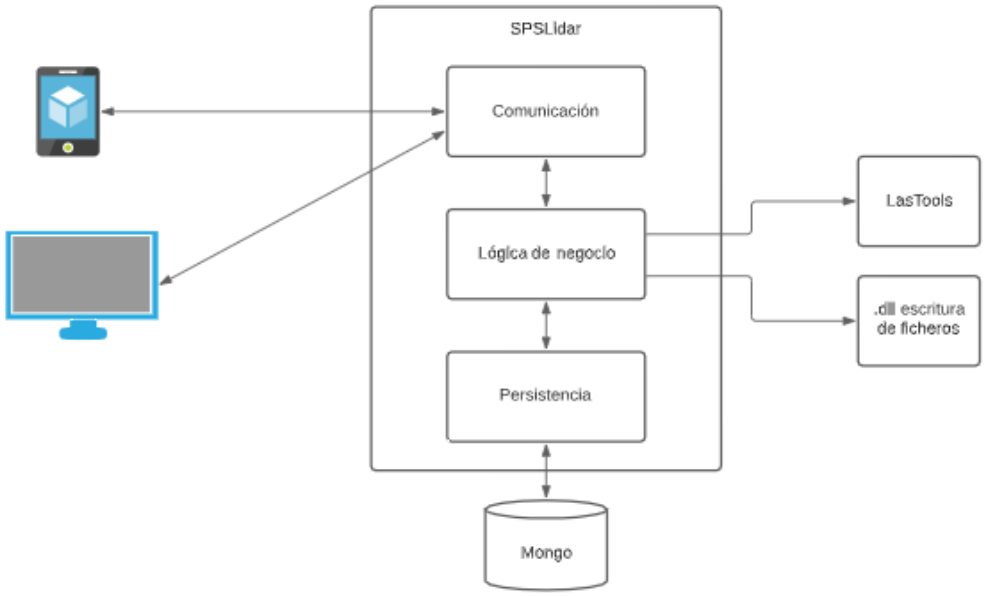

Ilustración 4. Arquitectura básica de la aplicación.

- Comunicación: se encargará de gestionar las peticiones recibidas de los distintos tipos de clientes que quieran interactuar con el sistema. Se utilizará el protocolo HTTP y los datos estarán estructurados en formato JSON.

- Lógica de negocio: implementa todas las funcionalidades y algoritmos necesarios para que se lleven a cabo la funcionalidad esperada. Para la escritura de ficheros se plantean el uso de las LASTools y una librería en C++ propia definida como un .dll, externas ambas a la aplicación. Será desde esta capa donde se realicen las llamadas a estas herramientas cuando se necesite.

- Persistencia: gestiona la comunicación con la base de datos, en este caso se utilizará MongoDB.

Aunque esta estructura muestra una sola instancia de la aplicación y un solo nodo de Mongo, es posible escalar el sistema tanto en la capa de persistencia definiendo un clúster con múltiples servidores de Mongo, así como lanzar instancias adicionales de la aplicación principal.

7.2- Modelo conceptual

Como resultado de los requisitos iniciales, se ha elaborado un modelo lógico base sobre el que se sustentará el sistema. Estará compuesto por tres entidades fundamentales: *workspace*, *dataset* y *datablock.*

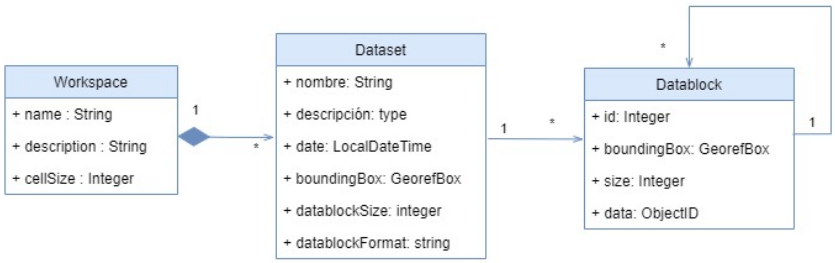

Ilustración 5. Dominio básico del sistema

La entidad workspace actuará como un contenedor de datasets relacionados entre sí, ya sea porque representan información de zonas cercanas, pertenecen a escaneos dentro de un mismo dominio (por ejemplo, escaneos de yacimientos arqueológicos) o cualquier otro motivo. No se imponen restricciones espaciales ni temporales para los datasets que queden registrados bajo el mismo workspace. Un workspace quedará identificado por su atributo *name*, tendrá una descripción que aporte información sobre qué tipo de datasets contendrá y un atributo *cellSize* cuya función se detallará posteriormente.

Un dataset hace referencia a una o varias nubes de puntos obtenidas durante un escaneo en una zona concreta. Quedará identificado por un nombre que lo distingue unívocamente entre el resto de datasets que se almacenen dentro del mismo workspace. Dentro de sus atributos, cabe destacar la fecha de adquisición, que actuará en representación de todas las nubes de puntos que se vayan a asociar a este dataset y las coordenadas suroeste y noreste del área sobre la que se llevó a cabo el escaneo.

En la relación entre workspace y dataset se incluirá un grid disperso. El objetivo fundamental de esta estructura de datos es acelerar las consultas en base a criterios espaciales cuando el número de datasets asignados a un mismo workspace sea elevado. El grid tendrá dos capas: la primera, emplea el sistema de zonas UTM para llevar a cabo un primer filtrado, mientras que en la segunda se definen celdas que tendrán un tamaño igual al atributo cellSize de la entidad workspace. Si el usuario va a utilizar el workspace para incluir datasets que se encuentran en zonas geográficas no muy lejanas entre sí, sería interesante utilizar un tamaño de cellSize menor, mientras que, si se pretende incluir escaneados tomados en zonas distantes, un tamaño mayor sería más recomendable.

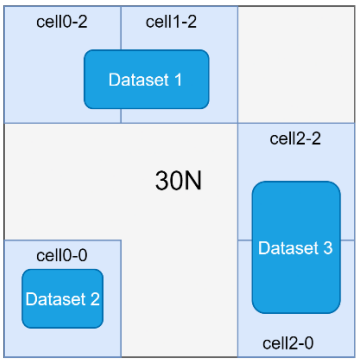

Ilustración 6. Estructura básica del grid.

La entidad dataset no hace referencia directa a los puntos que componen la nube. Podríamos considerar asociar los ficheros de un directamente al dataset, pero esto presenta una doble problemática: primero, los ficheros recibidos pueden ser muy grandes. Si posteriormente un usuario quisiera recuperarlos, el envío a través de la red podría ser lento y con un volumen de información excesivo para el cliente; y segundo, el acceso a zonas concretas del territorio representado en este escaneo quedaría delegado al usuario final, que tendría que recuperar todos los ficheros que se hubiesen almacenado y llevar a cabo un proceso de exploración, muchas veces tedioso y con un alto coste computacional.

Para dar una solución a estos problemas, se introduce el concepto de datablock y el uso de una estructura de datos jerárquica. En este trabajo se ha utilizado un octree como estructura de datos, de tal forma que cada nodo del octree vendrá representado por un datablock. Cada datablock mantendrá una referencia a sus datablock hijos: cuando un datablock vaya a superar el cupo de puntos máximo que puede almacenar, se llevará a cabo un proceso de subdivisión, partiendo el área del datablock padre en 8 regiones a las que se asignará un nuevo datablock. Del mismo modo, un datablock tendría asociado una nube de puntos representada como un fichero, con los puntos que hayan sido asignados a dicho datablock. De esta forma, se plantea una solución en la que nos aseguramos de que el envío de ficheros tenga un tamaño adecuado para su transferencia mediante la red, puesto que los ficheros tendrán una limitación de tamaño impuesta por el número máximo de puntos en un datablock, y se facilita la exploración del territorio mediante un sistema de niveles de detalle, partiendo de los nodos raíces hasta profundizar en las zonas que resulten de interés para el cliente.

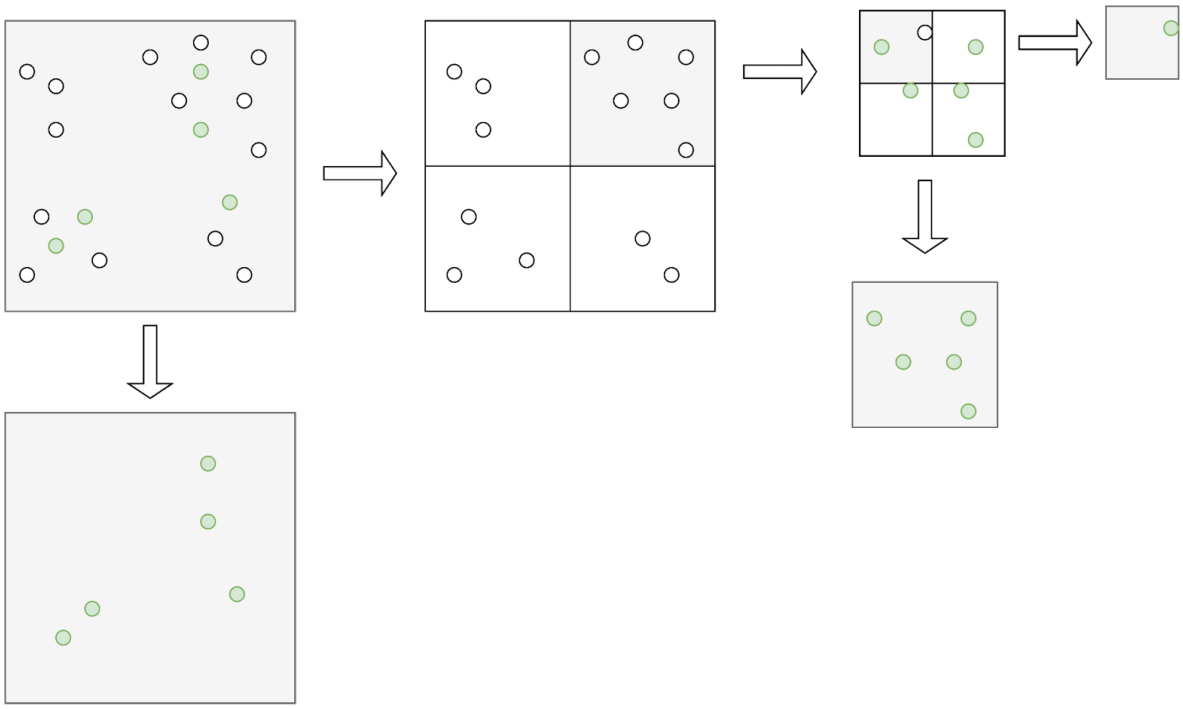

Ilustración 7. Generación del octree de datablocks.

7.3- API Restful

La siguiente tabla muestra todos los endpoints que expone el sistema. Los metadatos transferidos y recibidos por el sistema son esencialmente DTOs que representan cada una de estas entidades.

#	Método	URL	Descripción
1	GET	/**workspaces**	El sistema muestra todos los *workspaces* registrados en el sistema. ["/workspaces/CHJaen"]

2	POST	/**workspaces**	El sistema crea un nuevo workspace.
			{
			"name": "CHJaen",
			"description": "Scanned models of cultural heritage sites in theprovince of Jaén (Spain)",
			"cellSize": 10000
			}

3	GET	/**workspaces**/ {workspace_name}	El sistema devuelve la información del *workspace* al nombre indicado en la petición.
4	GET	/**workspaces**/{workspace_name} /**datasets**? southWest={sw_coord} &northEast={ne_coord} &fromDate={from_date} & toDate={ to_date}	El sistema devuelve los datasets asociados al *workspace* especificado filtrando por la ventana espacial indicada y un intervalo de tiempo de forma opcional. ["/workspaces/CHJaen/datasets/CastilloSCatalina"]

5	POST	/**workspaces**/{workspace_name}
		/**datasets**
			El sistema registra un nuevo *dataset.* Todavía no se le asignará una nube de puntos, esta operación se realiza posteriormente a través del método 9.
			{
			"name": "CastilloSCatalina",
			"description": "Model of Castillo de Santa Catalina (Jaén)",
			"dateOfAcquisition": "2019-05-03T12:10:00Z",
			"boundingBox": {
			"southWestBottom": {
			"easting": 429522, "northing": 4180270, "height": 790,
			"zone": "30S"
			},
			"northEastTop": {
			"easting": 429638, "northing": 4180363, "height": 820,
			"zone": "30S"
			}
			},
			"dataBlockSize": 10000,
			"dataBlockFormat": "LAZ"
			}

6	GET	/**workspaces**/{workspace_name} /**datasets**/{dataset_name}	El sistema devuelve la información del *dataset* especificado.

7	GET	/**workspaces**/{workspace_name}
		/**datasets**/ {dataset_name}
		/**datablocks**/{ datablock_id}
			El sistema devuelve los metadatos asociados a un *datablock*.
			{
			"id": 0,
			"bbox": {
			"southWestBottom": {
			"easting": 429522, "northing": 4180270, "height": 790,
			"zone": "30S"
			},
			"northEastTop": {
			"easting": 429638, "northing": 4180363, "height": 820,
			"zone": "30S"
			}
			},
			"size": 10000,
			"subDataBlocks": [1, 2, 3, 4, 5, 6, 7, 8]
			}

8	GET	/**workspaces**/{workspace_name} /**datasets**/ {dataset_name} /**datablocks**/{datablock_id}/ data	El sistema devuelve el fichero asociado al *datablock* especificado.
9	PUT	/**workspaces**/ {workspace_name} /**datasets**/{dataset_name}/ data	El sistema recibe la nube de puntos asociada a un dataset, generando una estructura de datos jerárquica.

Tabla 2. API Restful expuesta por el sistema

Con los métodos #2, #5 y #9 se realizan las tres operaciones de inserción establecidas en el sistema: crear workspaces, datasets y añadir un conjunto de nubes de puntos. Con los métodos #3 y #6 podemos recuperar workspaces y datasets por sus nombres si los conocemos. Mediante el método #1 podemos obtener un listado de todos los workspaces, mientras que utilizando el #4 podemos llevar a cabo consultas con criterios espaciales y temporales para descubrir nuevos datasets que puedan interesarnos. Una vez hayamos recuperado un dataset podemos explorar la estructura generada durante el método #9 utilizando el método #7 y utilizar el método #8 para descargar el fichero asociado a un datablock específico.

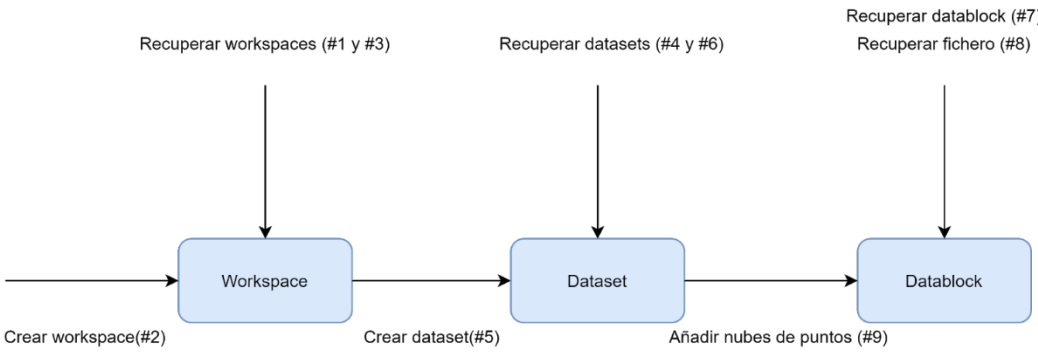

Ilustración 8. Interacción del usuario con el sistema mediante la API.

8- Análisis de riesgos

Para el análisis de riesgos y su priorización se ha seguido un enfoque cualitativo, utilizando la matriz de probabilidad e impacto del INCIBE como referencia.

		IMPACTO		
		Bajo	Medio	Alto
PROBABILIDAD	Baja	Muy bajo	Bajo	Medio
	Media	Bajo	Medio	Alto
	Alta	Medio	Alto	Muy alto

Ilustración 9. Matriz de probabilidad e impacto del INCIBE

Para documentar los riesgos se ha seguido un enfoque basado en plantillas. Los puntos que se definirán para cada riesgo son:

- Riesgo: identificación y breve descripción significativa de este.
- Categoría: para clasificar los riesgos se han utilizado la siguiente topología.
    - Riesgo tecnológico
    - Riesgo personal
    - Riesgo organizacional
    - Riesgo de herramientas
    - Riesgo de requisitos
    - Riesgo de estimación
- Probabilidad de que ocurra el riesgo
- Impacto del riesgo en el proyecto
- Puntuación obtenida a partir de los valores de probabilidad e impacto estimados para el riesgo y la matriz del INCIBE.
- Respuesta al riesgo: plan de actuación en caso de que tenga lugar el riesgo especificado

Identificación del riesgo		Clasificación cualitativa				Respuesta al riesgo
Riesgo	Categoría	Probabilidad	Impacto	Puntuación	Ranking	
Baja por enfermedad	Riesgo personal	Baja	Medio	Baja	3º	Realizar una replanificación del proyecto a corto plazo.
Cambio en los requisitos iniciales	Riesgo de requisitos	Baja	Medio	Baja	3º	Análisis del impacto que tendrían los cambios propuestos y replanificación y priorización del nuevo conjunto de requisitos.
El rendimiento de la base de datos es insuficiente	Riesgo tecnológico	Baja	Alto	Media	2º	Análisis del modelo de datos empleado en la capa de persistencia para detectar posibles fallos. Integración y realización de pruebas con otros sistemas de bases de datos.
LASTools no soporta alguna de las operaciones necesarias en el desarrollo del proyecto.	Riesgo tecnológico	Media	Alto	Alto	1º	Estudio de alternativas, ya sea a nivel tecnológico buscando un kit de herramientas o librería adicional o implementado de forma propia la funcionalidad necesaria o a nivel de diseño, planteando una solución distinta sobre la cual LASTools pueda operar correctamente

Tabla 3. Análisis de riesgos

9- Metodología y planificación temporal

La metodología de desarrollo seguida ha tenido un enfoque ágil. Se ha buscado tener una versión funcional de la aplicación, que de soporte a las distintas capas del sistema y permita llevar a cabo operaciones completas desde la primera iteración. Mediante un enfoque iterativo e incremental, se ha ido progresivamente mejorando el sistema e incorporando distintas funcionalidades.

A nivel de planificación temporal, la implementación del proyecto se ha llevado a cabo en dos etapas. Inicialmente se había planeado terminar el desarrollo en la primera, pero debido a las dificultades técnicas encontradas, el no contar con excesivo tiempo disponible al compaginar este trabajo con el primer curso del Máster en Ingeniería Informática y en última instancia, los efectos ocasionados por la pandemia, se tuvo que ampliar el trabajo al último trimestre de 2020. La primera iteración se sitúa entre los meses de noviembre de 2019 y abril de 2020, mientras que la segunda se llevó a cabo entre octubre de 2020 y diciembre de 2020. Durante la primera etapa se han llevado a cabo las tres primeras iteraciones, que se presentarán a continuación, mientras que en la segunda etapa se han llevado a cabo las dos restantes.

Ilustración 10. Diagrama temporal del proyecto

10- Presupuesto

El desglose presupuestario del proyecto se realizará en tres vertientes, la primera dirigida a los componentes hardware, la segunda al coste del software y la última al coste del personal.

Para los componentes tecnológicos se va a realizar un proceso de amortización. Para ello se van a seguir las bases descritas por la Ley 27/2014 sobre el Impuesto de Sociedades, publicada en el BOE del 28 de noviembre de 2014. Esta ley diferencia en el periodo de amortización correspondiente al software y al hardware, de tal modo que la amortización del software se hará en base a 6 años (2190 días) y la del hardware en base a 8 años (2920 días).

Para la amortización se han realizado dos consideraciones adicionales:

- Puesto que el tiempo de uso se ha documentado en meses, se ha especificado que el número de días por mes será de 30 días.

- El valor de amortización por día se ha redondeado a 4 decimales, ya que pese a ser un valor monetario, los valores obtenidos son pequeños. En el valor definitivo, que vendrá dado en la columna de amortización final se redondeará a dos decimales.

Dentro del hardware se diferenciará entre dos equipos utilizados. El primero se trata de un portátil al que se tuvo acceso en la segunda fase del proyecto, con el objetivo de facilitar el trabajo remoto debido a las consecuencias de la pandemia. El segundo se trata de un sobremesa. Cabe destacar que el equipo utilizado en la primera fase del proyecto y en la segunda no fue el mismo. Sin embargo, puesto que poseen características similares y ya que en el momento de redacción de este documento no se tiene acceso al primero de estos equipos, se contabilizarán en el ámbito presupuestario como el mismo ordenador.

Recurso hardware	Coste (€)	Amortización/día (€)	Tiempo de uso (meses)	Amortización para los meses de uso (€)
Portátil MSI Modern 14 A10RAS	999,99	0,3425	4	41,1
Intel Core i7-10700 CPU @2.90 Ghz	299,99	0,1027	9	27,73
32 GB RAM DDR4 ~3200 Mhz	178	0,061	9	16,47
GeForce GTX 1660 SUPER	299	0,1024	9	27,65
1 TB HDD	34,95	0,012	9	3,24
Fuente alimentación 650 W	49,99	0,0171	9	4,62
Placa base Gigabyte H410M S2H	56,99	0,0195	9	5,27
Disco duro externo 5 TB WD	106,99	0,0366	4	4,39
Monitor Dell	89,99	0,0308	9	8,32
Caja PC iqwo	19,99	0,0068	9	1,84
Teclado	14,99	0,0051	9	1,38
Ratón	9,99	0,0034	9	0,92
Coste amortizado del hardware (€)				142,93

Tabla 4. Presupuesto recursos hardware.

A nivel de software, las herramientas utilizadas han tenido un acceso gratuito, ya sea porque la propia Universidad de Jaén las ofrece directamente a sus estudiantes o porque los propios vendedores facilitan el acceso a estas mediante un correo electrónico de tipo académico. Dentro de este apartado se destaca la utilización de Windows 10 Enterprise como sistema operativo, IntelliJ Pro para el desarrollo de código Java, PyCharm Pro para el desarrollo de código Python, Visual Studio 2019 para el desarrollo de código en C++ y Visual Paradigm para la elaboración de diagramas software.

A nivel de costes de personal, el proyecto fue realizado durante los primeros 6 meses con la financiación de una beca Ícaro. En el siguiente curso, el trabajo realizado quedó encuadrado dentro de un contrato laboral de la Universidad de Jaén. De este

segundo contrato contabilizaremos para este proyecto solo los meses empleados en el desarrollo que aparezca en este TFM.

Puesto	Salario mensual (€)	Duración (meses)	Coste (€)
Beca Ícaro	196	6	1176
Contrato laboral Universidad de Jaén	1166	3	3498
Coste total recursos humanos			4674

Tabla 5. Presupuesto recursos humanos.

El coste total obtenido sería el siguiente:

Recurso	Coste presupuestado (€)
Recursos hardware	142,93
Recursos software	0
Recursos humanos	4674
Coste total recursos humanos	4816,93

Tabla 6. Coste total del proyecto.

Por último, se va a considerar para el importe final del proyecto un porcentaje de gastos indirectos del 10%. Por lo tanto, el importe final sería de 5141,4 €.

Gasto	Coste (€)
Coste del proyecto	4674
Porcentaje gastos indirectos (10%)	467,4
Importe final	5141,4

Tabla 7. Importe final del proyecto.

11- Desarrollo

11.1 - Iteración I

Esta primera iteración se llevó a cabo en los meses de noviembre y diciembre de 2020. Los objetivos que se plantean para esta son el desarrollo de una versión inicial de la aplicación, que integre desde el principio las tres capas presentadas en la arquitectura y de soporte para trabajar con la información asociada a las entidades workspace y dataset.

11.1.1 - Capa de comunicación

Deberemos dar soporte de manera inicial a los métodos expuestos en la API (7.3- API Restful) con identificadores 1, 2, 3, 4, 5 y 6, permitiendo operaciones de creación y lectura sobre workspace y dataset.

En los siguientes diagramas se muestran cuáles son estos métodos, los parámetros que deberán especificar los clientes que quieran acceder a ellos y que información y códigos de respuesta se deberán enviar desde el servidor. Los endpoints de la API para interactuar con cada una de las entidades estarán distribuidos en dos clases que actuarán como controladores, *WorkspaceController* y *DatasetController*. El intercambio de información se realizará mediante instancias de la clase *ResponseEntity* de Spring, parametrizadas para *Mono* o *Flux,* los dos tipos de publicadores ofrecidos en Spring Webflux.

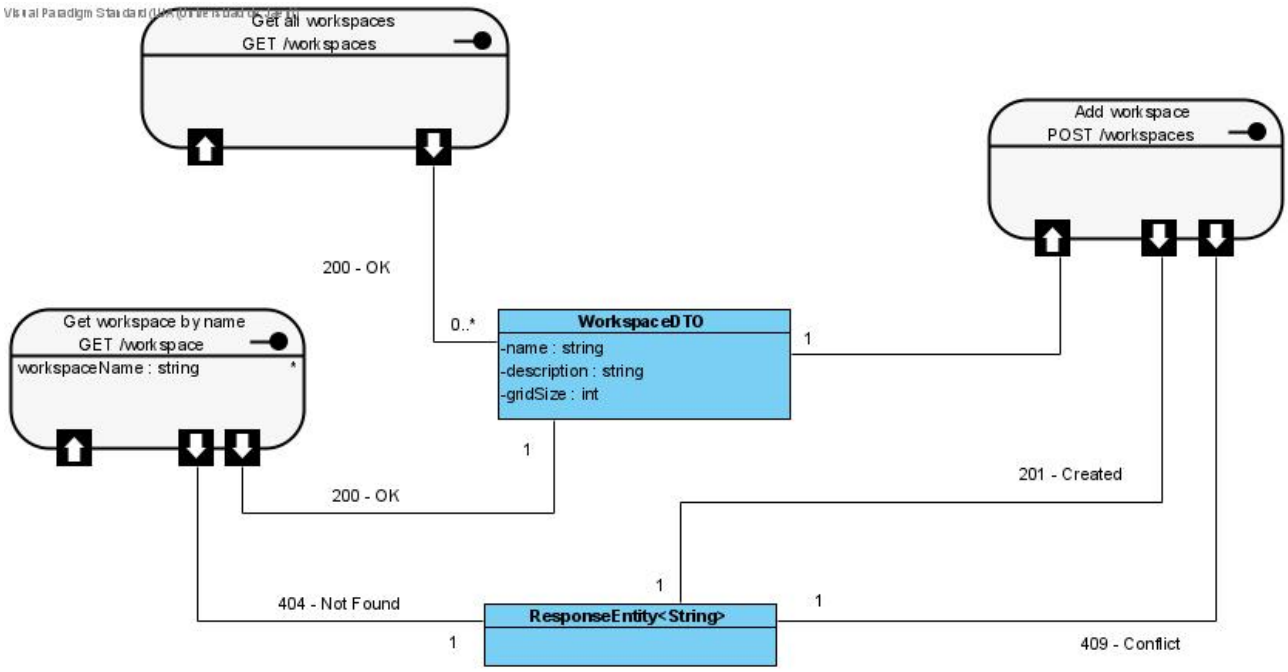

Ilustración 11. Diagrama capa comunicación para la entidad workspace.

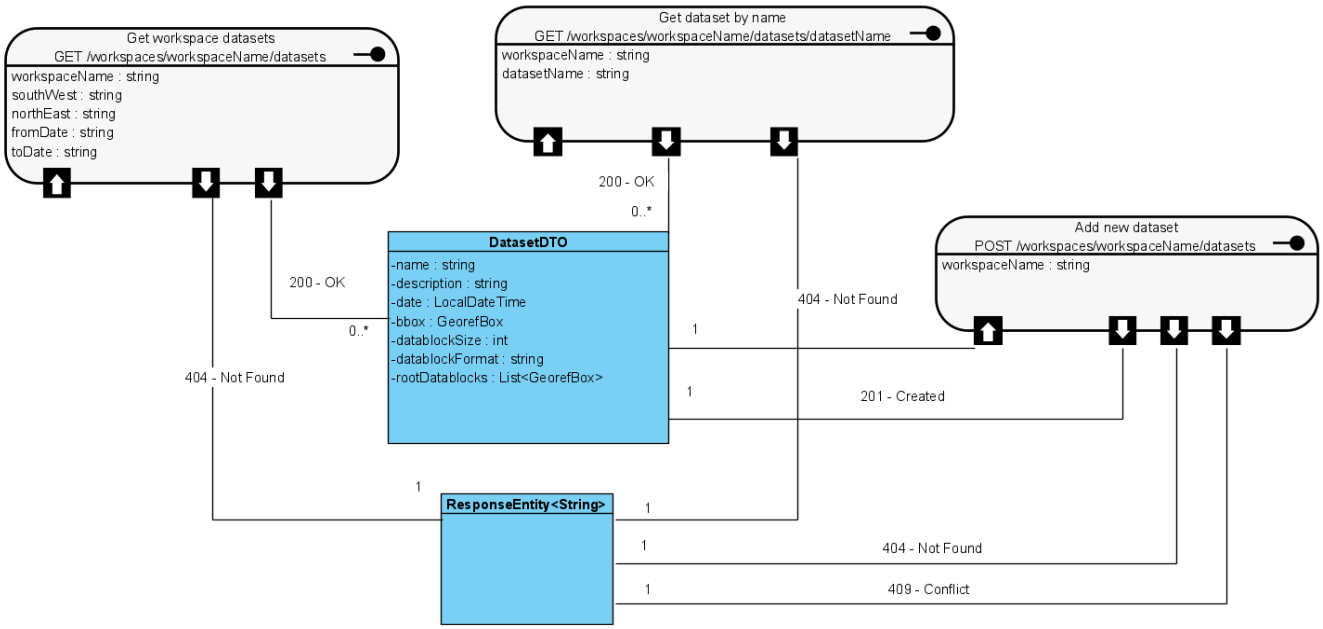

Ilustración 12. Diagrama capa comunicación para la entidad dataset.

11.1.2 - Servicios

Los servicios implementados se ajustan de momento a actuar como enlace entre la capa de comunicación y persistencia. Su función de momento será principalmente realizar comprobaciones básicas para los diferentes escenarios y códigos de respuesta definidos en la API. Pese a dar soporte a las entidades workspace y dataset, todavía no se implementará el grid disperso presentado en la sección 7.2- Modelo conceptual que actúa como nexo entre ellas. Las operaciones por ventana espacial se llevarán a cabo por lo tanto comprobando directamente las cajas envolventes definidas en la entidad dataset.

11.1.3 - Persistencia

A nivel de capa de persistencia, se han definido dos clases que actuarán como repositorios, encargadas respectivamente de las entidades workspace y dataset, así como una clase con la configuración básica de la aplicación para poder comunicarse con Mongo.

La interacción con MongoDB se puede realizar de varias maneras, siendo la más sencilla para el desarrollador el uso de interfaces que a su vez extiendan la interfaz *ReactiveCrudRepository*. Esta se ha de parametrizar, especificando que entidad va a gestionar y el tipo del ID que tendrá el documento cuando se almacene en Mongo. Spring se encarga de manera interna de generar una implementación de esta interfaz, proveyendo por defecto una serie de métodos básicos de tipo CRUD. También podemos añadir nuevas operaciones, definiendo un método que anotamos con *@Query* y al que especificamos la sentencia en formato JSON que debería ejecutar Mongo cuando este se invoque. Si bien este enfoque es bastante sólido y nos da una base sobre la que trabajar de manera casi inmediata, también posee ciertas limitaciones de flexibilidad. Como se ha mencionado previamente, un repositorio debe ser parametrizado con la clase que va a manejar. La clase debe contar con la anotación *@Document* y el nombre de la colección en la que se almacenarán los documentos generados a partir de cada instancia que vaya a ser

persistida en el sistema. Todas las operaciones asociadas a esa entidad se llevarán a cabo siempre en esa misma colección. Por lo tanto, si quisiéramos distribuir nuestra entidad dataset en distintas colecciones, generadas dinámicamente, este método no funcionaría.

La alternativa consiste en utilizar *ReactiveMongoTemplate*, una clase proporcionada por Spring que expone distintos métodos CRUD de uso sencillo, métodos para realizar consultas personalizadas introduciendo sentencias JSON y la posibilidad de crear nuevas colecciones, así como de especificar en cada consulta el nombre de la colección en la que queremos que se lleve a cabo. Para su uso, crearemos dos clases dentro de la capa de persistencia, una para workspace y otra para dataset, que definen los métodos que necesitamos para cada una de estas entidades en el sistema. Dentro de estos métodos se utilizará una instancia de ReactiveMongoTemplate para realizar la operación que corresponda en cada caso. Para desacoplar estas clases de la capa de lógica de negocio, se ha definido una interfaz que deben implementar. Esto nos facilitará en el futuro poder utilizar otros sistemas de bases de datos.

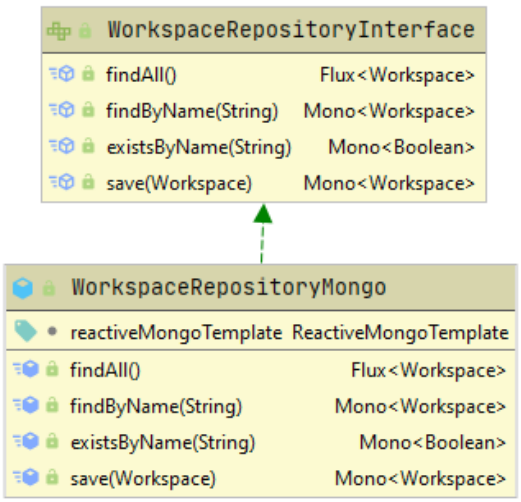

Ilustración 13. Diagrama de clases para la persistencia de la entidad Workspace.

A nivel de entidades, se han definido workspace y dataset como elementos relacionados únicamente por un atributo de tipo String en dataset, que referencia el nombre del workspace al que pertenece. Otra opción posible podría haber sido embeber los modelos asociados a cada workspace en una lista dentro del mismo documento o guardar en cada workspace un array de referencias a los nombres de los modelos asociados a él.

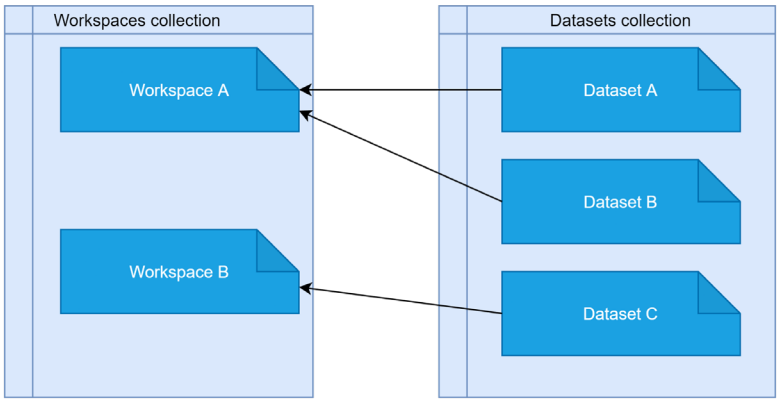

Ilustración 14. Estructura de la base de datos en la iteración 1.

El almacenamiento de estas entidades se realiza a través de las clases *WorkspaceDBDTO* y *DatasetDBDTO*. De este modo, tendremos las clases *WorkspaceDTO* y *DatasetDTO* para la comunicación entre cliente y servidor, *Workspace* y *Dataset* para trabajar en la capa de lógica de negocio y *WorkspaceDBDTO* y *DatasetDBDTO* en la capa de persistencia. Esta estructura se aplicará al resto de entidades que vayan surgiendo a lo largo del proyecto y que requieran algún tipo de tránsito ya sea a través de la red o con Mongo.

11.1.4 - Tests unitarios

Para garantizar el correcto funcionamiento del sistema, se ha decidido desde esta iteración llevar a cabo un proceso de cobertura de cada método de la API con tests unitarios. En esta subsección se explica y define cual es la estructura de clases

utilizada dentro del paquete de tests, mientras que las pruebas llevadas a cabo se comentarán en una subsección adicional al final de cada iteración.

Se ha utilizado el componente de Spring *WebTestClient*, que permite realizar peticiones a la API siguiendo los principios reactivos de Spring Webflux. Se han separado los tests en función de la entidad que pretenden testear: para cada una se han definido dos clases, una bajo el nombre de *[entity]EndpointsCaller* (p. ej, *WorkspaceEndpointsCaller*), que implementa los métodos para hacer las llamadas a la API y otra bajo el nombre de *[entity]Test,* que implementa los tests en sí, especificando los parámetros y situaciones con los que invocar los métodos ofrecidos por el correspondiente *EndpointsCaller*. Un aspecto importante a destacar es que estos tests son de carácter unitario, por lo que la ejecución de un test debe incorporar todos los pasos necesarios para complementarlo. En este sentido, si queremos realizar operaciones sobre la entidad dataset, primero debemos haber insertado en el mismo test un workspace al que vaya asociado. Por lo tanto, DatasetEndpointsCaller tendrá una dependencia hacia WorkspaceEndpointsCaller.

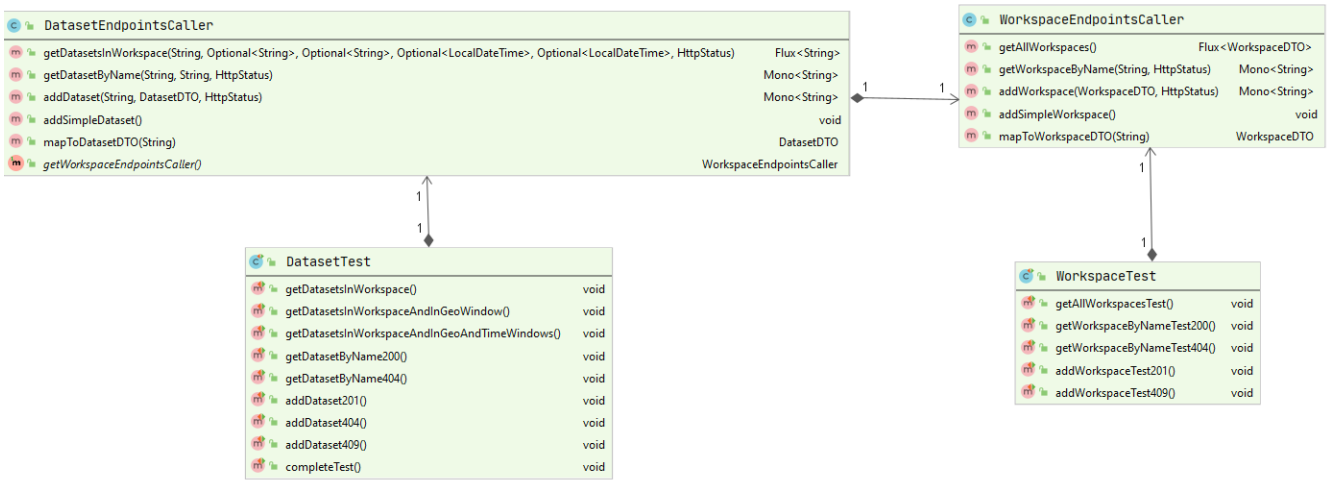

Ilustración 15. Diagrama de clases del paquete Tests en la iteración I

11.1.5 – Pruebas realizadas

La validación de las funcionalidades implementadas se ha llevado a cabo mediante la estructura de tests unitarios introducida en la subsección anterior. Estas pruebas abarcan tanto escenarios donde la operación ha tenido un resultado satisfactorio como erróneo, asegurando en todo momento que los códigos de respuesta devueltos al cliente son los apropiados y que la información almacenada en el sistema es la esperada.

En la entidad workspace, las pruebas realizadas pasan por la inserción de un nuevo workspace, inserción de un workspace duplicado, recuperación de todos los workspaces almacenados en el sistema, recuperación de un workspace existente en el sistema por nombre y recuperación de un workspace no almacenado por nombre. Para la entidad dataset, la inserción y recuperación se han testeado utilizando pruebas con el mismo enfoque. Adicionalmente, se han incluido distintos tests que corroboren el correcto funcionamiento de las consultas espaciotemporales sobre esta entidad.

11.2 - Iteración II

Esta segunda iteración se realizó en los meses de enero y febrero. En esta iteración se comienza con el desarrollo de las funcionalidades asociadas a la entidad *datablock*. Un datablock hará referencia cada nodo de la estructura de datos utilizada. En base al modelo de datos definido, esta entidad vendrá definida principalmente por una caja envolvente tridimensional, referencias unívocas a aquellos datablocks que sean sus hijos e identificadores del workspace y dataset al que pertenece, aunque en esta iteración tendrá una estructura algo más sencilla sobre la que posteriormente se profundizará.

11.2.1 - Soporte para la entidad datablock

En esta iteración todavía no se procederá a implementar la estructura de datos que da sentido plenamente a la clase datablock. De momento, en vez de generar una estructura jerárquica con múltiples datablocks asociados a un dataset, se modela como una entidad sin una relación hacia sí misma, teniendo asociado un fichero LAZ con la nube de puntos original. El modelo de datos presente en esta implementación se muestra en la Ilustración 16.

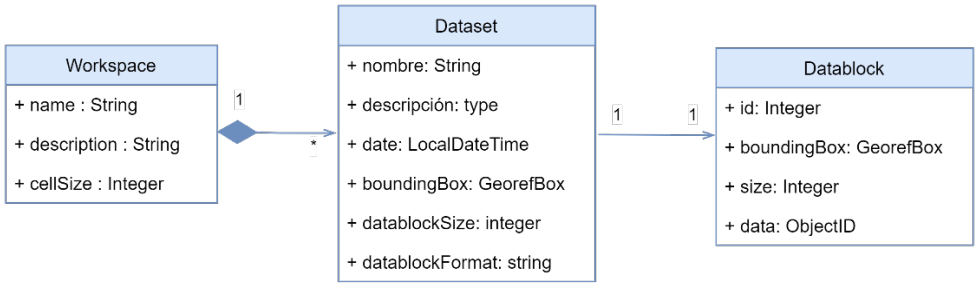

Ilustración 16. Modelo de datos definido en la iteración II.

De este modo nos permite tener una estructura básica con la que poder implementar las funcionalidades en las que se centrará esta iteración: transmisión de ficheros entre cliente y servidor, almacenamiento de ficheros en la base de datos e implementación básica en la capa de comunicación, servicios y persistencia para los métodos de la API 7, 8 y 9. Estos métodos se modificarán y ampliarán en iteraciones posteriores para satisfacer las modificaciones sobre el modelo resultantes del procesamiento de la nube de puntos y su indexación a través del octree. De momento, el objetivo que se plantea es la inserción de nubes de puntos, guardadas directamente en la base de datos, y asociadas a un datablock que vendrá identificado por el nodo 0 y carecerá de nodos hijos. Una vez completada la subida de la nube, podremos recuperar los metadatos del datablock generado durante el proceso, así como la nube enviada originalmente. En la Ilustración 17 se muestran los tres métodos implementados en la API en esta iteración, con los parámetros que aceptan y los códigos de respuesta que podrán devolver al cliente.

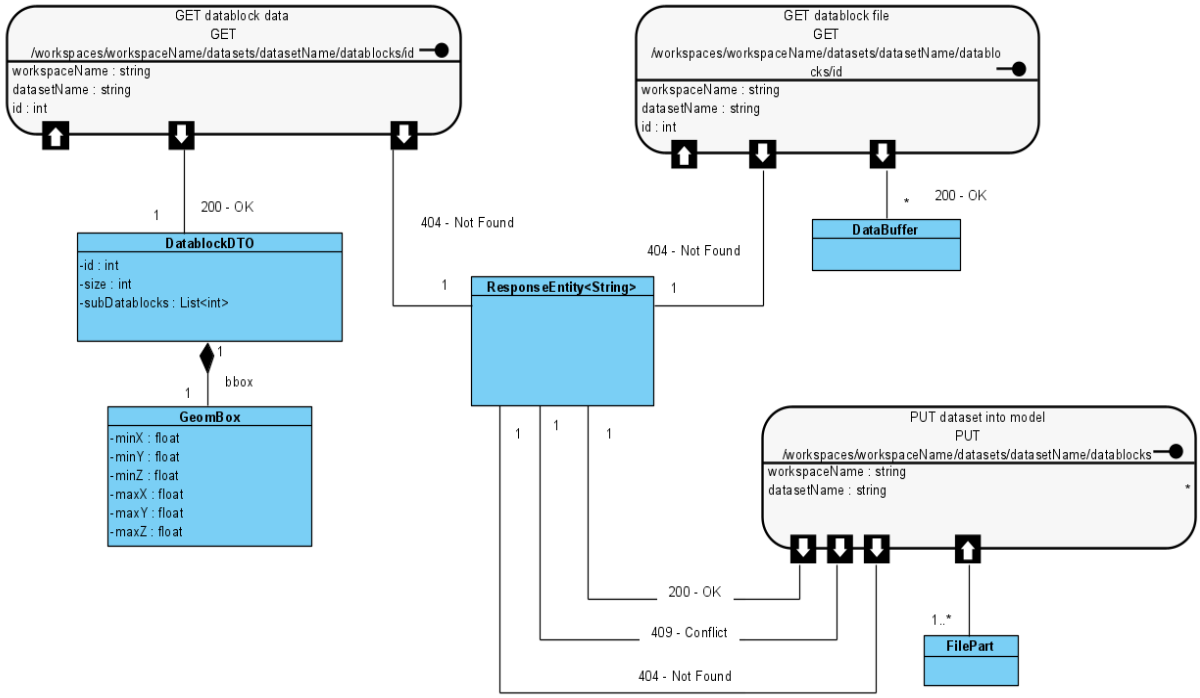

Ilustración 17. Diagrama capa comunicación para la entidad datablock.

11.2.2 - Intercambio de ficheros entre cliente y servidor

La subida de ficheros al servidor es uno de los principales aspectos técnicos a implementar en esta iteración. El sistema debe ser capaz de recibir un dataset completo de ficheros, en formato LAZ y, en el futuro, construir la estructura espacial. El número y tamaño de estos ficheros podrá variar, por lo que debemos de utilizar mecanismos reactivos que eviten un consumo excesivo de memoria en la JVM. Del mismo modo, el servidor deberá enviar los ficheros que genere a un cliente cuando se realice la petición correspondiente.

Algunos de los enfoques seguidos pasaron por enviar el fichero como un array de bytes o un string binario, pero en general no funcionaban correctamente. Finalmente, la elección elegida fue utilizar *FilePart*, una interfaz que define Spring y

que es implementada por el servidor web que de soporte a la aplicación. Un Filepart representa cada fichero enviado en una petición *multipart/form-data* del estándar HTTP. Para poder dar soporte a peticiones que incluyan más de un fichero, definimos un Flux parametrizado con FilePart.

```
@PutMapping(
value = "{workspaceName}/datasets/{datasetName}/datablocks",
       consumes = MediaType.MULTIPART_FORM_DATA_VALUE)
public Mono<ResponseEntity> addPointCloud(
                      @PathVariable String workspaceName,
                      @PathVariable String datasetName,
                      @RequestPart("files") Flux<FilePart> files)
```

En esta cabecera del método definido en el controlador para añadir una nube a un dataset se le indica que recibirá información de un *multipart/form-data* y que esta información irá asociada a un @RequestPart que se ha denominado como *"*files*"*. FilePart define dos métodos útiles para para almacenar la información asociada al fichero: podemos obtener un *Flux<DataBuffer>* con el contenido del fichero vía el método *content* o bien directamente escribir el contenido del fichero a un archivo local usando *transferTo*. Como lo único que queremos hacer es almacenar los ficheros, se ha usado esta última opción. DataBuffer es una interfaz definida en Spring que representa un nivel de abstracción superior a un ByteBuffer de Java.

Para el envío de ficheros desde el servidor al cliente se ha utilizado un Flux<DataBuffer>. En este caso, como solo se devuelve un fichero por petición, podemos utilizar la interfaz DataBuffer. Por un lado, no tendremos problemas de mezclar distintos DataBuffer provenientes de ficheros diferentes en el mismo Flux, ya que solo se solicita uno por petición de usuario. La otra principal ventaja pasa por el hecho de que la herramienta *ReactiveGridFSTemplate* (en la cual se profundiza más en la siguiente subsección) nos devuelve un *ReactiveGridFSResource*, que es fácil de transformar a su vez en un Flux<DataBuffer>.

11.2.3 - GridFS

Otro aspecto clave respecto al manejo de ficheros es su almacenamiento en el sistema de bases de datos. El contenido de estos ficheros lo podríamos incluir como un campo más, embebido en cada documento que represente un datablock. Una característica a considerar de este enfoque es el límite de 16 MBs impuesto por Mongo en el tamaño de documento. Por lo general, se espera almacenar ficheros de un tamaño menor a dicho valor y, a priori, no sería necesario utilizar GridFS. Sin embargo, con el objetivo de dar soporte a tamaños máximos de datablock elevados, que en ciertos casos podrían superar esos 16 MBs, y planteando la posibilidad de que en el futuro quisiéramos almacenar los ficheros originales enviados por el cliente, que sí excederían dicho límite claramente, se ha integrado GridFS en el sistema. Llegado el momento, se podrían realizar pruebas adicionales para estudiar si existe una diferencia de rendimiento significativa entre ambos enfoques e implementar algún mecanismo que, en función del tamaño del fichero, decida de forma dinámica si embeber el contenido del fichero en el datablock o bien use GridFS.

El acceso al servicio de GridFS de Mongo se realiza a través de la clase *GridFileStorageService*. En esta clase se utiliza un objeto reactiveGridFsTemplate, proporcionado por Spring. Los dos casos principales a abordar son el almacenamiento de ficheros y la recuperación de estos.

El almacenamiento de ficheros se realiza pasando una instancia de un datablock. La clase datablock tiene un atributo con la ruta al fichero con la nube de puntos que le corresponde. El objetivo es cargar el fichero de forma reactiva al sistema, y para ello nos apoyaremos en las clases *ResourceLoader* y *DataBufferUtils*. Mediante la primera estableceremos como un objeto de tipo Resource el fichero que queremos almacenar, y que pasaremos como argumento al método *read* de DataBufferUtils. Este método es capaz de leer los contenidos del fichero de forma asíncrona, generando un Flux<DataBuffer>. Debemos definir adicionalmente el

tamaño máximo utilizado para cargar en memoria cada uno de los trozos del fichero, el cual se ha establecido en 1 MB.

Una vez establecido el mecanismo de lectura del fichero almacenado en el sistema de archivos, el paso siguiente es almacenarlo en GridFS. Para ello utilizaremos la clase reactiveGridFsTemplate y su método *store*, al que pasamos como argumento el Flux devuelto por DataBufferUtils::read. El resultado de esta operación es un Mono con el identificador de Mongo del documento en la colección *fs.files* correspondiente al archivo almacenado. Este identificador se lo debemos asociar al datablock para que en el futuro podamos acceder a su archivo LAZ. Un último aspecto a destacar es que podemos definir un tamaño de *chunk* para el almacenamiento del fichero, que determinará en cuantos trozos se dividirá el contenido del archivo en Mongo. Por defecto, se utiliza 255 KBs, de tal modo que el máximo contenido que poseerá cada uno de las subdivisiones del fichero no superará dicho valor. La parametrización de este valor se puede hacer utilizando una clase auxiliar llamada *reactiveGridFsUpload* y más adelante en el proyecto se experimentará con distintos tamaños para analizar posibles variaciones en el rendimiento.

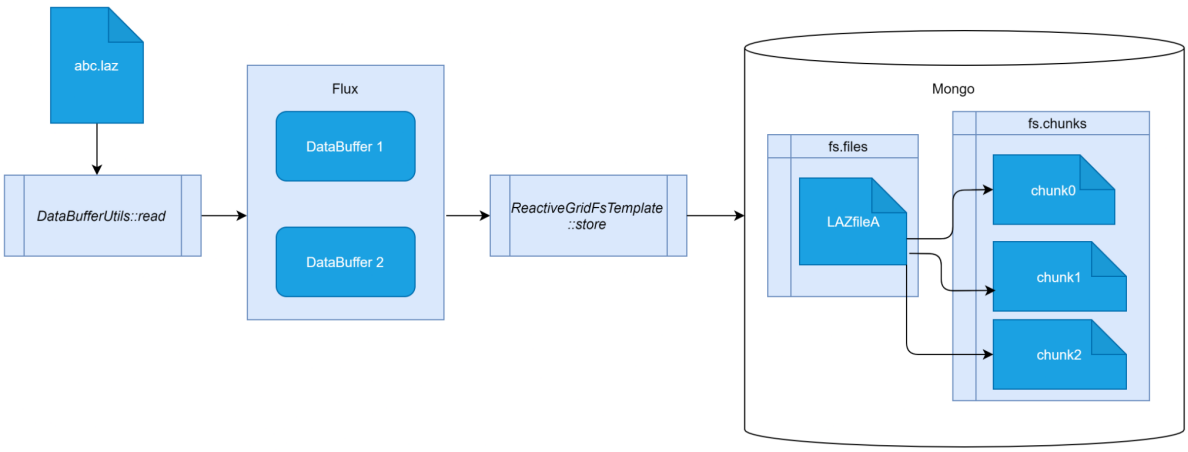

Ilustración 18. Proceso de almacenamiento de ficheros LAZ en Mongo

Por último, el método de búsqueda será abordado de la siguiente manera: primero, realizaremos una consulta sobre la colección donde se almacenan los datablocks, recuperando aquel que se adhiera a los parámetros especificados por el usuario en su llamada al servidor. Todos los datablocks tienen un atributo de tipo *ObjectID*, que representa el identificador de su fichero LAZ asociado dentro de GridFS. Utilizando este atributo, se define una query que busque en la colección fs.files el documento cuyo *_id* coincida con él. Esta consulta se ejecuta a través del método *reactiveGridFsTemplate::findFirst*. En caso de que el resultado sea satisfactorio, nos devuelve una instancia de la clase *GridFsFile*, nuestro punto de entrada al fichero LAZ. Esta clase guarda la referencia al documento dentro de fs.files que queremos recuperar, pero debemos tener en cuenta que el contenido binario del fichero LAZ está distribuido entre múltiples documentos dentro de la colección fs.chunks. GridFsFile nos ofrece una forma sencilla de recuperar todos estos chunks a través de su método *getResource*. Este nos devuelve un objeto de tipo *ReactiveGridFsResource*, el cual podemos a su vez convertir en un Flux<DataBuffer>, que procedemos finalmente a enviar al cliente.

11.2.4 – Pruebas realizadas

Para confirmar que la transmisión de ficheros y su almacenamiento es correcto, se han definido distintos tests que lleven a cabo la subida de ficheros, considerando distintos tamaños de archivo (desde 10 MBs hasta 30 GBs) así como el envío dentro de una misma petición PUT de un solo fichero a un directorio completo compuesto por decenas de archivos LAZ. Puesto que el envío de ficheros es bidireccional, se han desarrollado tests que soliciten al servidor ficheros LAZ almacenados y comprueben la integridad de estos. En estas pruebas, además de los escenarios con códigos de respuestas correctos, se han realizado distintos tests que comprueben que, en caso de pedir un datablock con un ID no existente o asociado a un dataset o workspace que no está almacenado en el sistema, se devuelva el código de respuesta de la familia 400 apropiado.

11.3 - Iteración III

En esta tercera iteración se desarrolla la implementación de la estructura espacial, un octree que almacena los puntos asociados a cada uno de sus nodos en memoria. Se complementarán los métodos implementados en la iteración anterior, en la que ya trabajamos con la entidad datablock, entendida entonces como un elemento sencillo que representaba la nube de puntos al completo. Ahora, basándonos en el octree, se generará una jerarquía de datablocks, cada uno ofreciendo niveles de detalle distintos de una zona concreta del espacio total.

11.3.1 - Generación octree y almacenamiento

La implementación del octree supone un cambio en nuestro modelo de datos. Ahora, aparece una relación entre distintos datablocks, resultado de la conexión padre-hijos propia de la estructura de datos que vamos a implementar.

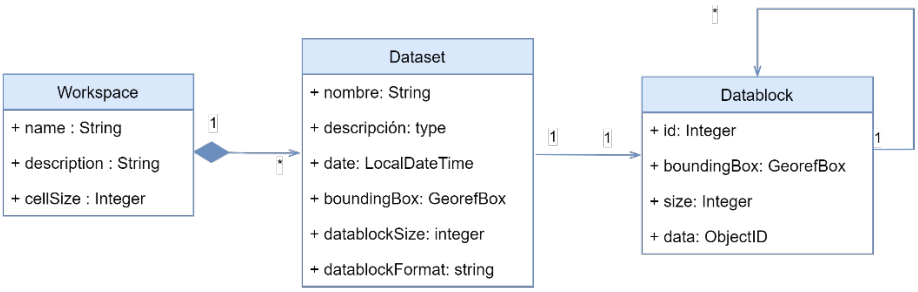

Ilustración 19. Modelo de datos definido en la iteración III.

La generación y almacenamiento del octree incluye la funcionalidad necesaria para llevar a cabo el método #9 de la API definida en la sección 7.3- API Restful. Involucra múltiples clases y métodos, por lo que se ha elaborado un diagrama de secuencia presente en la siguiente ilustración, que, sin ser demasiado exhaustivo, pretende mostrar el funcionamiento general de este método y las clases involucradas.

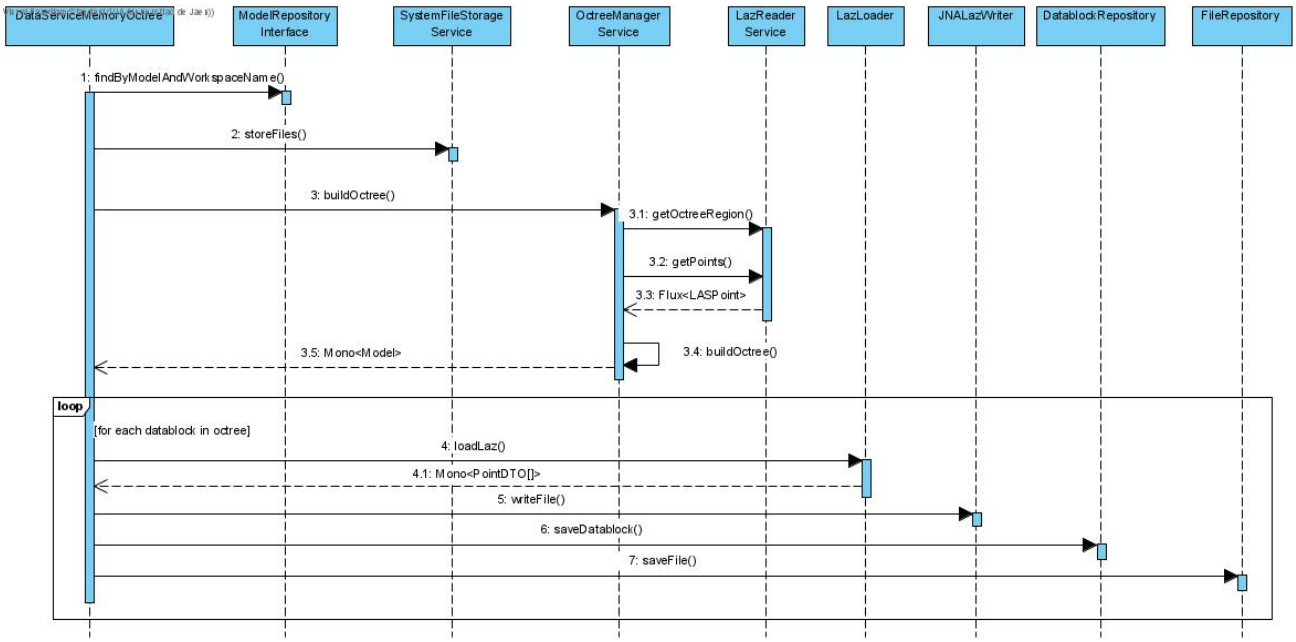

Ilustración 20. Diagrama de secuencia simplificado para la inserción de un dataset en el sistema.

Los pasos necesarios pasan por buscar el modelo al que se asociará la nube de puntos, para comprobar que realmente existe y la operación es válida. Tras ello, se almacenan los ficheros y se procede a construir el octree, utilizando un *Flux<LASPoint>* en el que se irán publicando los puntos de la nube de puntos a medida que se vayan leyendo de los ficheros. Este proceso se lleva a cabo en la clase LazReaderService, donde se encapsulan los métodos basados en Laszip4j.

Cuando se genera un datablock, este tomará como máximo tantos puntos como se hayan establecido en el atributo maxDatablockSize, definido en la entidad dataset. En el momento en el que se alcanza dicho límite, se procede a crear ocho subregiones de igual tamaño entre ellas, utilizando como referencia la caja envolvente del datablock padre. A partir de estas subregiones, se instancian ocho datablocks, que pasarán a ser los hijos del datablock del que partíamos, el cual guardará una referencia a ellas para poder explorar la estructura en el futuro. Existe otro criterio de parada, basado en el concepto de profundidad máxima, implementado con el objetivo de evitar octrees demasiado complejos. Cuando un datablock se encuentre a un nivel

de profundidad igual al definido en esta restricción, se ignorará el límite de puntos máximo y no tendrá hijos.

Un aspecto importante en esta estructura de datos espacial que estamos implementando es en qué sistemas de coordenadas trabajamos. Aunque posteriormente se utilizarían coordenadas UTM, en esta primera implementación, el octree utilizará valores normalizados, por lo que tanto los puntos como las cajas envolventes de cada nodo estarán definidos en el intervalo [0,1]. Para ello, utilizando dos puntos que representan las coordenadas máximas y mínimas del modelo y un valor que representa la diagonal que conecta ambas coordenadas, se calcula las coordenadas normalizadas de cada nuevo punto.

```
diagonal = sqrt((maxX-minX)^2+(maxY-minY)^2+(maxZ-minZ)^2)

normalizedX = (originalX - minX)/(diagonal)
normalizedY = (originalY - minY)/(diagonal)
normalizedZ = (originalZ - minZ)/(diagonal)
```

Un aspecto a destacar también es que la información relativa a las coordenadas en un LASPoint no representa los valores reales de las coordenadas directamente. En la cabecera de cada fichero LAZ encontramos unos valores de *scale* y *offset* para cada dimensión. A cada punto deberemos aplicar la siguiente operación antes de aplicar las operaciones de normalización introducidas anteriormente:

```
originalX = lasPoint.getX()*xScale + xOffset
originalY = lasPoint.getY()*yScale + yOffset
originalZ = lasPoint.getZ()*zScale + zOffset
```

La clase OctreeManagerService se encargará de iniciar el proceso de creación, gestionando las llamadas a LazReaderService y creando una instancia inicial que represente el nodo raíz con la clase datablock. A medida que se le pasen puntos, este irá creando recursivamente nodos hijos, repitiéndose el proceso hasta completar la asignación del total de puntos.

Cuando se complete la creación del octree, el nodo raíz se asigna al modelo que le corresponde, estableciendo la relación entre ambos en memoria y posteriormente, en la base de datos. Tras ello, se procederá a generar un Flux con los nodos del octree. Esto se realiza gracias al operador *expand* presente en Mono. Gracias a este, podemos generar de forma recursiva un Flux que contendrá todos los nodos del sistema. A partir de aquí, podemos proceder a ir invocando mediante JNA la librería en C++ que se encargará de generar el fichero correspondiente, y finalmente, almacenando los ficheros en GridFS.

La principal limitación de este planteamiento pasa por el hecho de que, si bien los puntos son leídos de forma reactiva, no importará demasiado ya que el proceso de creación del octree implica que solo se finaliza cuando se hayan asignado todos los puntos, y, por lo tanto, acabaremos indistintamente con toda la información de la nube de puntos en memoria. Además, en la creación de los ficheros vía JNA, se utiliza un array como argumento para la librería en C++, que guarda instancias de una clase llamada *PointDTO*, basada en la clase LASPoint de la librería Laszip4j pero adaptada para adherirse a los requisitos de JNA. Puesto que LASPoint no nos da acceso a todos los atributos en los formatos de punto más modernos del estándar LAS, PointDTO carecerá de algunos de estos atributos y, por lo tanto, el fichero LAZ generado también. Los ficheros LAZ creados mediante esta estrategia pueden carecer de algunos de los metadatos presentes en la nube de puntos original. Aun así, la mayoría de atributos más relevantes para tareas de visualización (posición, color, etc.) sí quedan registrados, por lo que esta implementación podría ser funcional si trabajamos con una nube de puntos relativamente pequeña y con formatos de puntos sencillos de la especificación del estándar LAS.

11.3.2 - JNA

Aunque previamente se ha indicado el rol que tiene JNA y la librería implementada en C++ a la hora de escribir ficheros, en esta subsección se profundizará en cómo se ha llevado la integración de estas tecnologías en el sistema.

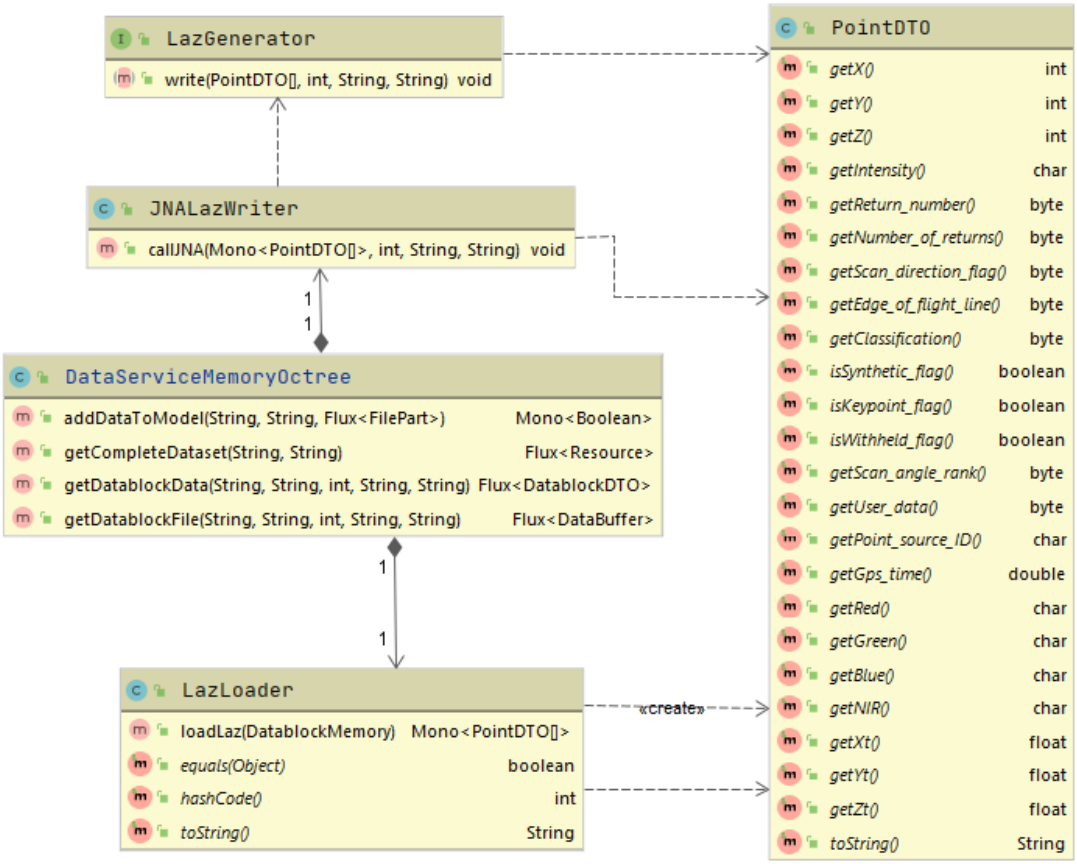

Ilustración 21. Diagrama de clases del paquete spslidar.jna

La clase PointDTO encapsula los atributos accesibles de un punto en el estándar LAS. La clase no hace distinción entre formatos, simplemente recopila todos aquellos disponibles en una instancia de LASPoint y, en caso de que alguno no esté presente, lo asigna a null. Los tipos especificados para cada atributo siguen las convenciones definidas en la documentación de JNA y se toman como referencia los tipos de la clase LasPoint de la librería LASLib de C++, puesto que es la clase a la que queremos convertir nuestros puntos en última instancia. Por ejemplo, el atributo que representa la clasificación de un punto en LasPoint se representa como un char, por lo que en nuestra clase PointDTO en Java el atributo debe ser de tipo byte.

Native Type	Size	Java Type	Common Windows Types
char	8-bit integer	byte	BYTE, TCHAR
short	16-bit integer	short	WORD
wchar_t	16/32-bit character	char	TCHAR
int	32-bit integer	int	DWORD
int	boolean value	boolean	BOOL
long	32/64-bit integer	NativeLong	LONG
long long	64-bit integer	long	__int64
float	32-bit FP	float	
double	64-bit FP	double	
char*	C string	String	LPCSTR
void*	pointer	Pointer	LPVOID, HANDLE, LP*XXX*

Ilustración 22. Mapeo de tipos Java - C++ definido en JNA.

La interfaz LazGenerator define los métodos que se han implementado en la librería en C++ y que queremos llamar desde Java. En nuestro caso consta solo del método *write*, al que se le pasará un array de PointDTO, el número total de puntos, la ruta al fichero original del dataset y la ruta donde queremos escribir el archivo resultante. El motivo por el que pasamos una referencia al fichero original es para extraer la información de la cabecera: ya que sus datos no se modifican durante la fase de procesamiento y creación del octree, podemos llamar directamente al archivo original para extraer la información de él, en vez de cargarla en Java y transmitirla mediante JNA. Solo hay una excepción, las coordenadas máximas y mínimas del fichero, que solo serían válidas para el nodo raíz (el cual si abarca la misma área que el fichero original). Estas las actualizamos manualmente en cada fichero, ya que a medida que escribimos los puntos proporcionados en el array, podemos definir dos

variables que vayan actualizándose con las coordenadas máximas y mínimas, para finalmente actualizar los valores de la cabecera correspondientes a estas.

La clase *JNALazWriter* es la clase que se encarga de lanzar la operación de escritura. Dentro de la clase se ha definido un método denominado *callJNA*, que genera una instancia de LazGenerator. En base al diagrama de clases expuesto anteriormente, LazGenerator es una interfaz. La implementación de esta interfaz será nuestra librería implementada en C++, la cual debemos exportar como un .dll (en nuestro caso hemos llamado al archivo *DllLaz.dll*) y almacenar dentro del proyecto. Con el siguiente fragmento de código JNA realiza la asociación entre la interfaz de Java y el .dll que la implementa:

```
LazGenerator = (LazGenerator) Native.load("DllLaz", LazGenerator.class);
```

El siguiente paso sería llamar desde nuestro código en Java la operación *write* de LazGenerator. Sin embargo, todavía es necesario preparar el array de puntos que queremos enviar al .dll. Este proceso de conversión se realiza en la clase *LazLoader*. Transforma los puntos de un datablock, encapsulados en la clase LASPoint provista por Laszip4j, en un array de tipo PointDTO, nuestra clase propia. Este array además debe usar memoria contigua para ser correctamente transferido. El proceso general lo orquesta la clase *DataServiceMemoryOctree*, llamando a LazLoader para obtener el array y pasándolo a JNALazWriter para que finalmente, LazGenerator lo incluya como argumento en su llamada al .dll.

11.3.4 – Pruebas realizadas

Con el fin de corroborar la correcta creación del octree y la escritura de ficheros LAZ, se han definido distintas pruebas utilizando distintos conjuntos de datasets LiDAR. Puesto que durante estos tests hemos detectado los problemas de rendimiento en nubes de puntos grandes inherentes a la manera en que se genera el octree en memorria, los experimentos realizados han ido orientados principalmente a

nubes de puntos de menor tamaño. Los ficheros creados tras el procesamiento se han aunado en uno solo para corroborar que el número de puntos distribuido es el correcto y se mantiene la integridad de la nube de puntos con respecto a la original. Del mismo modo, se ha llevado a cabo la visualización de multiples de las nubes de puntos generadas mediante la librería implementada en C++ para comprobar que los puntos y las regiones definidas siguen la estructura espacial esperada. Finalmente, utilizando los tests planteados en la iteración anterior, se han llevado a cabo pruebas para simular la exploración del octree, recuperando datablocks, profundizando a través de los hijos y pidiendo sus ficheros asociados.

11.4 - Iteración IV

11.4.1 - Generación del octree out-of-core

En esta cuarta iteración, se diseña e implementa una alternativa basada en un enfoque out-of-core para la distribución de los puntos de un modelo de nube de puntos. El trabajo desarrollado en la iteración III nos demostró que en la implementación basada en memoria no satisfacía plenamente los requisitos planteados para el sistema. Mediante el kit de herramientas de LasTools y su integración en el sistema, se implementará un proceso donde la creación y manipulación de ficheros se llevará a cabo integramente a través de ficheros en disco. El octree de datablocks seguirá presente, pero teniendo en cuenta que estos ya no contendrán en memoria sus puntos asociados, sino una referencia a la ubicación del archivo LAZ que registra su nube de puntos asociada en el sistema de archivos. Esto nos permitirá manipular nubes de puntos de tamaños mucho mayores, siendo la única limitación la cantidad de espacio disponible en la memoria secundaria del equipo donde se ejecute la aplicación.

Con el objetivo de poder elegir entre los dos enfoques que convivirán en el sistema se utiliza una interfaz denominada *DataService,* que nos permitiría cambiar en tiempo de ejecución entre ambos enfoques.

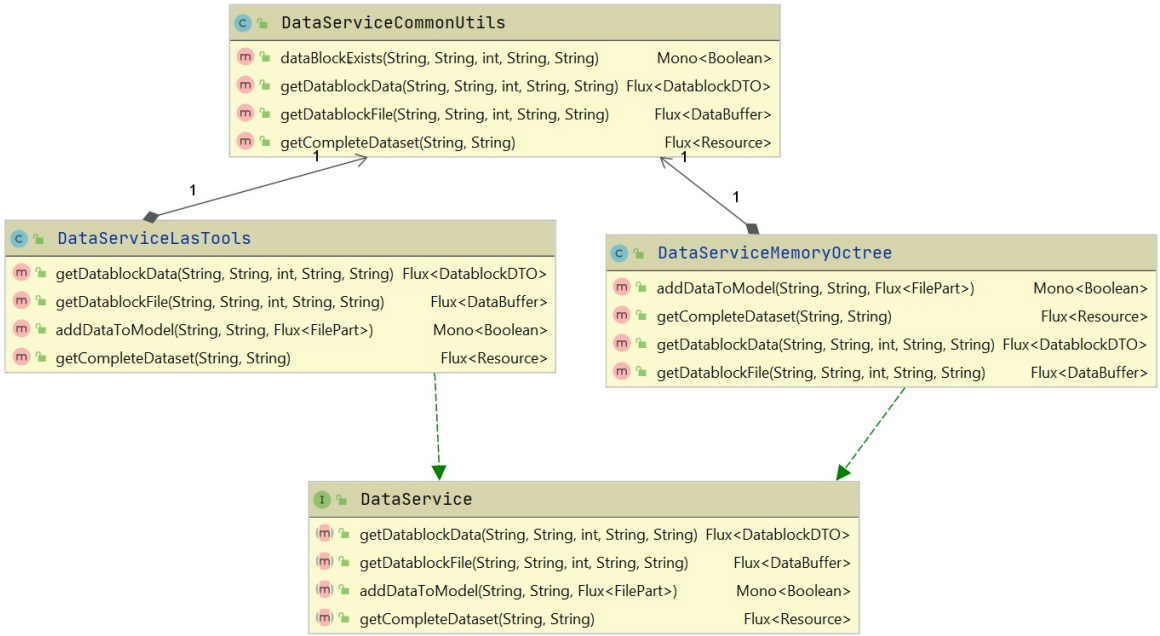

Ilustración 23. Diagrama de clases con los servicios que trabajan sobre datablock.

La interfaz define 3 métodos:

- *getDatablockData* define el método a implementar para devolver un Datablock específico.
- *getDatablockFile* define el método para recuperar un fichero correspondiente a un Datablock.
- *addDataToModel* define el método para realizar el proceso de inserción de los datos.

Si bien getDatablockData y getDatablockFile se espera que sean métodos idénticos en todas las implementaciones de esta interfaz y de hecho, se define una

clase auxiliar que los implementa llamada *DataServiceCommonUtils*, en el método *addDataToModel* habrá cambios sustanciales entre ambas implementaciones.

Del mismo modo, para interactuar con el kit de LASTools, se implementa la clase *LasToolsService*. Esta clase contendrá una gran variedad de métodos, que encapsulan llamadas específicas a diferentes utilidades de LASTools. A continuación, se resumen brevemente algunos de los más relevantes:

- *changeUserData*: cambia el atributo *user_data* de todos los puntos de un fichero a un valor a especificar.
- *sampleDataFromFile*: realiza un sampling del fichero utilizando *las2las* y *keep_random_fraction.*
- *sampleDataFromFileWithKeepNth*: realiza un sampling del fichero utilizando una estrategia alternativa basada en *las2las* y *keep_every_nth*.
- *mergeWithDuplicates:* une dos ficheros, eliminando cualquier punto repetido en base a sus coordenadas (X, Y, Z).
- *createChildNode*: crea un nuevo fichero LAZ con los puntos del fichero input que estén contenidos en una bounding box especificada como argumento mediante *-keep_xyz*,
- *createRootFile:* crea un fichero LAZ para cada grid. La principal diferencia con el método *createChildNode* consiste en que la bounding box especificada es bidimensional y solo usaremos las coordenadas (X, Y).
- *mergeFiles:* une múltiples ficheros en uno solo.
- *getUTMZone:* obtiene la zona UTM a la que pertenece el fichero. Esta acción se lleva a cabo capturando la salida de pantalla del comando lasinfo y aplicando un filtrado mediante expresiones regulares.

Todos estos métodos utilizan la clase *ProcessBuilder* de Java, que permite generar procesos a nivel del sistema operativo. Es importante tener instalado

LASTools y declarado como variable de entorno, puesto que todos estos métodos encapsulan llamadas a varias de sus utilidades. Para ejecutar estas llamadas al sistema, utilizamos el método *ProcessBuilder::start*. Puesto que estas operaciones son bloqueantes, para poder embeberlas dentro del flujo reactivo de la aplicación, las lanzamos dentro de un *Mono::fromCallable*, un mecanismo que nos ofrece Webflux para tareas externas que devolverán un solo elemento (en nuestro caso será la ruta al fichero que se haya creado en la operación correspondiente).

A partir de estos métodos se han desarrollado dos estrategias de generación de ficheros. La primera es la más compleja, requiere de más pasos y un tiempo de ejecución superior. Utiliza como elemento clave el atributo *user_data*, un metadato asociado a cada punto en el estándar LAS que carece de demasiada relevancia. La idea es partir de un fichero inicial (al que nos referiremos como fichero #1) en el que todos los puntos posean el mismo valor de user_data y aplicar el comando *keep_random_fraction*, de tal forma que se obtenga un fichero resultante (fichero #2) con un número de puntos muy cercano al valor máximo establecido y ya preparado para ser almacenado en la base de datos.

Sin embargo, para evitar redundancia son necesarios una serie de pasos adicionales, puesto que los puntos sampleados del fichero #1 no son extraídos de este, sino solamente copiados en el fichero #2. Para eliminar estos puntos del fichero #1, aplicamos *set_user_data* sobre el fichero #2, especificando un valor distinto al que tienen los puntos en el fichero original. Para estandarizar, se ha utilizado el valor 0 para marcar los puntos del fichero #1 y 1 para los puntos del fichero #2. Esta operación nos dará un nuevo fichero (fichero #3), que contiene los mismos puntos que en el fichero #2 excepto por el hecho de que su atributo *user_data* estará asignado a 1. A posteriori, haremos un *merge* de los ficheros #1 y #3 y aplicaremos la operación *lasduplicate -unique_xyz*. El resultado será un fichero nuevo que contendrá todos los puntos, pero con el matiz de que los puntos que habían sido seleccionados para el fichero #2 ahora tendrán un user_data igual a 1, lo cual nos permite distinguirlos del resto. Con la operación *las2las -keep_user_data 0*, obtenemos finalmente un fichero que representa el conjunto de puntos base menos

los puntos que han sido sampleados, garantizando que no exista redundancia. Este nuevo fichero procede a subdividirse en las regiones de los nodos hijos, con 8 llamadas a *las2las -keep_xyz* acompañadas en cada una de un argumento que represente la caja envolvente de cada subregión. Si en alguna de las regiones se genera un fichero vacío, el sistema detecta que no hay puntos para esta, y, por lo tanto, se descarta ese nodo.

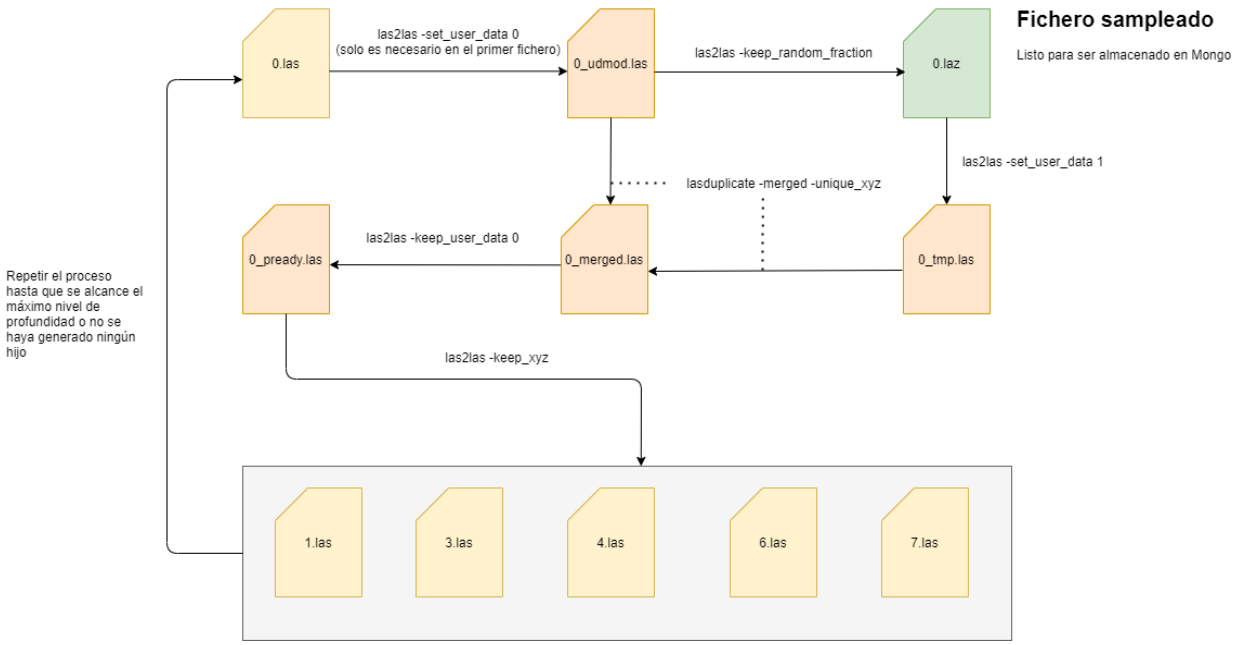

Ilustración 24. Método I para la construcción de la estructura de ficheros basada en un octree.

En la segunda estrategia se aplica un enfoque más sencillo. Mediante el comando *las2las -keep_every_nth* podemos construir un fichero sampleado quedándonos con un punto de cada X. El valor de X lo podemos calcular realizando la división entre el número de puntos contenidos en el fichero y el máximo número de puntos permitidos por datablock y quedándonos con el valor entero inmediatamente inferior al valor obtenido. El resultado será una nube, subconjunto de la original, con un número de puntos que será menor al máximo establecido, aunque rara vez exactamente igual, pues solo se dará esa situación si el resto de la división es 0. La

principal ventaja de este método es que para eliminar la redundancia podemos aplicar directamente el comando *las2las -drop_every_nth* con el mismo valor X y obtendremos el fichero listo para ser particionado.

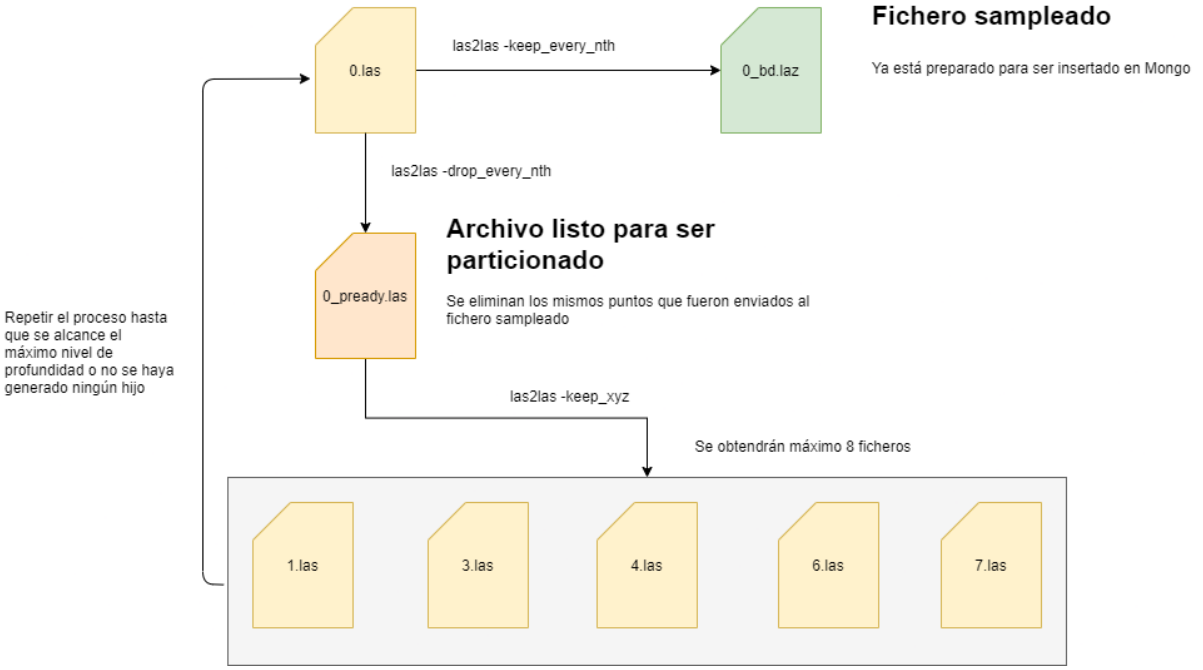

Ilustración 25. Método II para la construcción de la estructura de ficheros basada en un octree.

Independientemente de la estrategia utilizada, un aspecto relevante a considerar es la elección de usar LAS o LAZ como archivo para los ficheros temporales generados durante los dos algoritmos presentados. Existe una relación inversa entre el espacio ocupado en disco por los ficheros y el tiempo necesario para llevar a cabo la creación de los ficheros, ya que el formato LAS si bien tendrá ficheros de mayor tamaño, reduce considerablemente el tiempo de ejecución. Puesto que LAZ es la versión comprimida de LAS, el espacio ocupado es bastante inferior (en torno unas 10 veces menos). Sin embargo, el tiempo de ejecución será mayor, puesto que la compresión y descompresión del formato LAZ requiere un tiempo de computación adicional.

El sistema permite alternar entre el formato LAS y LAZ, definiendo una propiedad que es posible modificar en tiempo de ejecución para decidir que formato utilizar. Por ejemplo, podríamos utilizar el formato LAS por defecto y en caso de que el espacio ocupado alcance límites críticos, utilizar LAZ. Es importante destacar que esta elección solo afectará a los ficheros temporales creados, es decir, aquellos que se necesitan para poder generar los ficheros finales asociados a cada nodo. Los ficheros que se almacenen en base de datos siempre utilizan el formato LAZ, puesto que, al ser permanentes, debemos de optimizar todo lo posible el espacio ocupado. Aunque no ha quedado plasmado en los diagramas presentados con el fin de simplificar la visualización, se produce un borrado de los ficheros que tengan un carácter no definitivo (aquellos que no se almacenan en la base de datos) y no vayan a ser utilizados a posteriori. Adicionalmente, al concluir el proceso de almacenamiento de ficheros LAZ, se procede a eliminar el directorio al completo con todos los ficheros que hubiese.

11.4.2 - Grid

La introducción del grid supone el último cambio llevado a cabo en el modelo de datos y que da como resultado aquel presentado en la sección 7.2- Modelo conceptual. Esta estructura de datos aparece como un mecanismo para gestionar los datasets existentes en un workspace, con el objetivo de mejorar el rendimiento de las consultas espaciales en situaciones donde el número de datasets asociado a un workspace sea elevado.

El grid consta de dos capas. La primera, utiliza el propio sistema de zonas UTM: cada dataset define dos coordenadas y en ambas se especifica una zona UTM, siendo habitualmente la misma, aunque también se considera la posibilidad de que se encuentren en zonas UTM adyacentes. El resultado será la indexación del dataset a una o dos zonas UTM. Una vez realizado este primer filtrado, procedemos a dividir la zona UTM en un sistema de celdas disperso. El tamaño de las celdas vendrá determinado por el atributo cellSize de la clase workspace. En este caso, un dataset puede quedar asignado a múltiples celdas, las cuales guardarán una lista con todos

los identificadores de los datasets que solapen con ellas. Las consultas espaciales a partir de este momento harán la comprobación en base a esta nueva estructura de celdas y los valores de sus coordenadas, frente al enfoque previo que requería consultar las cajas envolventes de todos los datasets.

A nivel de implementación, debemos hacer unas consideraciones adicionales. Debemos tener en cuenta que la caja envolvente de un dataset se pasa cuando se registran sus metadatos (operación #5 de la API), momento en el que todavía no tenemos los ficheros que alimentará al modelo. El grid se genera en esta misma etapa y sigue un enfoque optimista, generando celdas para dar soporte a toda la caja envolvente definida, asumiendo que los ficheros que posteriormente lleguen se adecúan completamente a ella.

Sin embargo, esto no tiene por qué ser así, ya que la nube de puntos que posteriormente se asigne al dataset no tiene por qué representar un área regular o puede que el incluya múltiples ficheros que no representen zonas plenamente conexas entre ellas, con áreas vacías intermedias.

A continuación, se muestra un ejemplo con el objetivo de ilustrar un caso concreto de esta discrepancia existente entre la caja envolvente del dataset inicial y la superficie real de las nubes de puntos que finalmente se le asignen, así como la solución planteada. Supongamos que tenemos un dataset que, en base a su caja envolvente y al tamaño de rejilla definido para el workspace al que pertenece, solapa con 4 celdas: A, B, C y D. Ya que la caja envolvente del modelo solo cuenta con dos puntos, asumiremos siempre una forma rectangular.

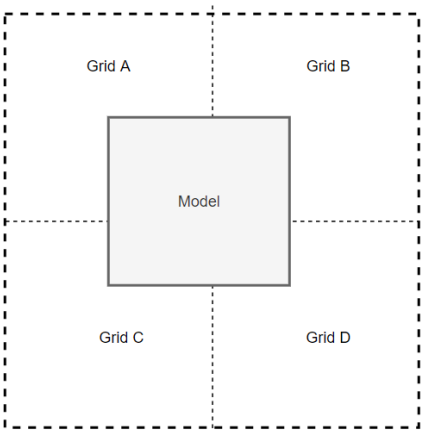

Ilustración 26. Modelo que abarca múltiples celdas de un grid.

En la siguiente imagen se muestran los ficheros que un usuario sube para alimentar este dataset. Consta de tres ficheros, los cuales efectivamente respetan las coordenadas mínimas y máximas que se especificaron para el dataset en el momento de su creación: la coordenada suroeste del fichero 2.laz corresponde a la coordenada mínima mientras que la coordenada noreste del fichero 3.laz corresponde a la coordenada máxima.

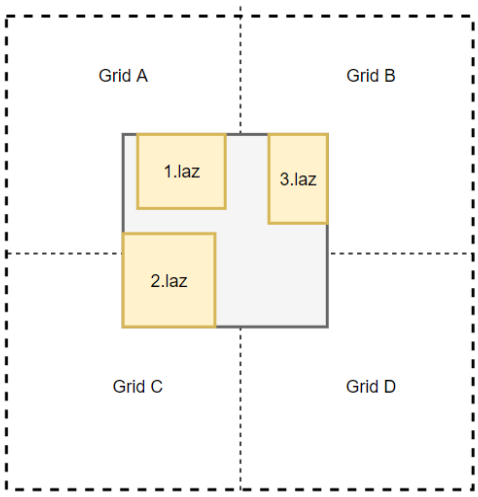

Ilustración 27. Ejemplo de un modelo que abarca múltiples celdas de un grid

Sin embargo, vemos que ningún fichero llega a solapar con la celda D. Por lo tanto, estamos asociando a una celda del grid un dataset que realmente no posee

puntos en ella. Para solucionar este problema, antes de construir el octree se sigue un procedimiento de manipulación y partición de los ficheros LAZ recibidos para separarlos tantos por zona UTM como por celdas del grid. La idea fundamental es utilizar las coordenadas de las celdas del grid para generar nuevos ficheros específicos para cada una de ellas, conteniendo solo los puntos que estén presentes en dicha celda. Continuando con el ejemplo previo, se generarían tres ficheros: la celda A tendrá asociado un fichero que contendrá todos los puntos de *1.laz* y una porción del fichero *2.laz*, la celda B tendrá todos los puntos del fichero *3.laz* y la celda C tendrá los puntos del fichero *2.laz* que caigan dentro de él, que serán la mayoría. El algoritmo también probaría a generar un fichero para la celda D, pero al no haber ningún punto, la operación no devolvería nada. En este momento es cuando detectamos que dicha celda no debe estar asociada a nuestro dataset, y por lo tanto, se lleva a cabo una actualización en el sistema para eliminar dicha relación.

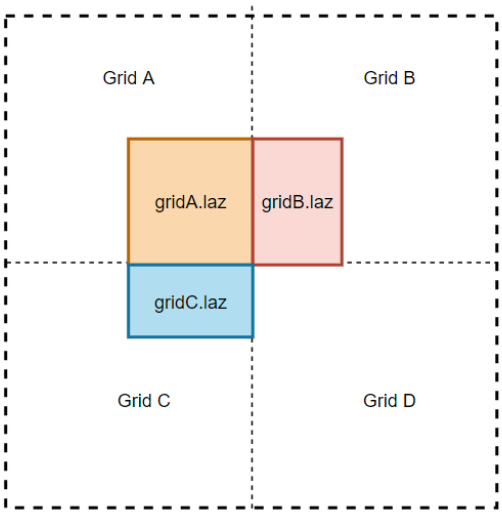

Ilustración 28. Ficheros bases para cada celda del grid.

Otro caso particular a mencionar ocurre cuando tenemos modelos cuyas cajas envolventes abarcan dos zonas UTMs diferentes. En este caso a la hora de generar las celdas del grid, se hace un proceso de reproyección de las coordenadas, de tal forma que generamos dos cajas envolventes, una con las coordenadas *southWest* y *northEast* en la zona UTM de *southWest* y otra con las coordenadas proyectadas en

la zona UTM de *northEast*. Este proceso incluye un paso intermedio, en el que se pasa de coordenadas UTM a un sistema global de longitud-latitud para posteriormente volver a pasar al sistema de coordenadas UTM, pero especificando una zona UTM distinta. En este caso particular se van a generar más celdas de lo habitual y muchas de ellas no tendrán asociadas puntos. Sin embargo, cuando se lleve a cabo la subida de la nube de puntos, en todas esas celdas que no tengan puntos se procederá a desreferenciar el dataset, utilizando el planteamiento visto anteriormente. La principal ventaja de este enfoque es que el dataset y el grid se adaptan a los ficheros finalmente recibidos, evitando tener que llevar a cabo operaciones pesadas que supongan tener que reproyectar todos los ficheros recibidos a una misma zona UTM mediante LASTools.

La siguiente ilustración demuestra cómo sería el proceso para pasar de los ficheros originales enviados por un usuario a los archivos LAZ que finalmente se utilizan para construir los octrees y que se ajustan a la estructura del grid.

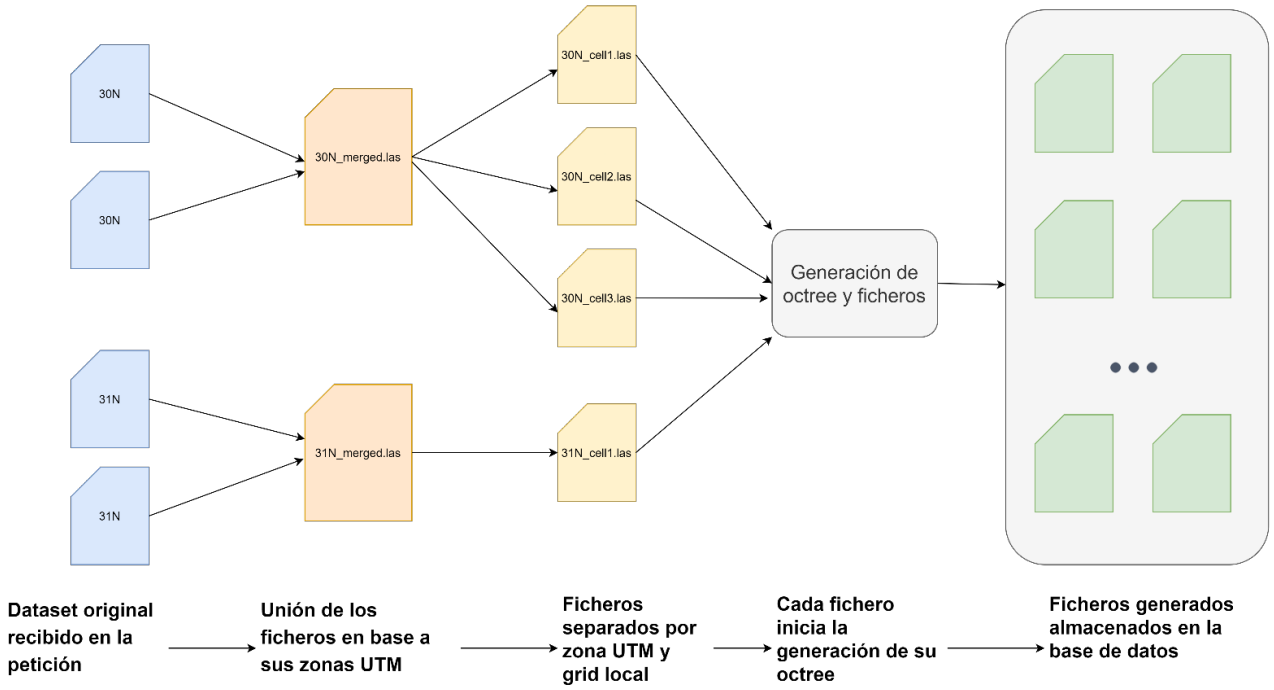

Ilustración 29. Procesamiento de los ficheros originales en base al grid.

11.4.3 – Pruebas realizadas

La principal funcionalidad añadida en esta iteración es la creación del octree mediante LASTools. Para determinar que los ficheros generados se ajustan al octree y respetan la integridad de la nube de puntos original, se ha llevado a cabo un proceso de verificación sobre la estructura de directorios creada en el sistema durante esta fase, combinando los ficheros que se almacenan posteriormente en la base de datos para visualizar la nube de puntos y comprobar que el número de puntos y la visualización global es la esperada. De forma individual, se han revisado distintas secuencias de ficheros que siguen una relación padre-hijo para comprobar que el particionamiento se hace de forma apropiada. Mediante los métodos de la API se han llevado a cabo pruebas que verifiquen la correcta recuperación de datablocks y ficheros, así como la inserción de modelos de nubes de puntos de distintos tamaños, utilizando distintos datasets con números de puntos que varían entre 250.000 y 1.000.000.000.

Adicionalmente, se han realizado consultas espaciales sobre la entidad dataset para confirmar el correcto funcionamiento del grid disperso implementado en esta iteración. Se han definido distintos escenarios, variando el tamaño de rejilla de la celda, creando datasets que abarcan desde una sola celda hasta múltiples de ellas, así como casos especiales en los que las coordenadas están expresadas en zonas UTM distintas. Puesto que el grid también tiene una influencia directa sobre la fase de creación del octree, al definirse un nodo raíz por celda, se han realizado comprobaciones que confirmen la correcta segmentación de la nube de puntos en base a estas y el sistema de zonas UTM cuando se lleva a cabo la fase de procesamiento de una nube de puntos.

11.5 - Iteración V

En esta quinta y última iteración, el trabajo realizado ha ido destinado a ultimar algunos detalles del servidor, llevar a cabo la implementación de un cliente que nos permita interactuar con el servidor y realizar pruebas de experimentación con el objetivo de medir el rendimiento del sistema, detectar fallos y probar diferentes configuraciones.

11.5.1 - Reestructuración de las colecciones de Mongo

A nivel de persistencia, el cambio más significativo ha venido en la estructura de las colecciones de Mongo. Hasta el momento, existían únicamente cinco colecciones: workspaces, datasets, datablocks, fs.files y fs.chunks. Se ha decidido modificar este enfoque, de tal forma que ahora habrá tantas colecciones para gestionar las entidades de dataset y datablock como workspaces haya. De este modo, se generarán de forma dinámica nuevas colecciones que seguirán la estructura de nombres *ws_datasets* y *ws_datablocks*, siendo *ws* el nombre del workspace al que estarán asociadas. Además, se ha creado una colección nueva para gestionar la entidad grid. Las celdas de un grid asociadas a un determinado dataset estaban embebidas como un atributo más en cada documento de la colección dataset. Ahora, sin embargo, representan una nueva colección, en la que cada celda tiene las coordenadas y zona UTM que la identifican unívocamente junto una lista con las referencias a los modelos que solapan con él. Al igual que en ws_datasets y ws_datablocks*, ws_grid* seguirá la misma estructura de nombres. En la imagen siguiente se muestra la implementación final de la estructura de la base de datos en Mongo.

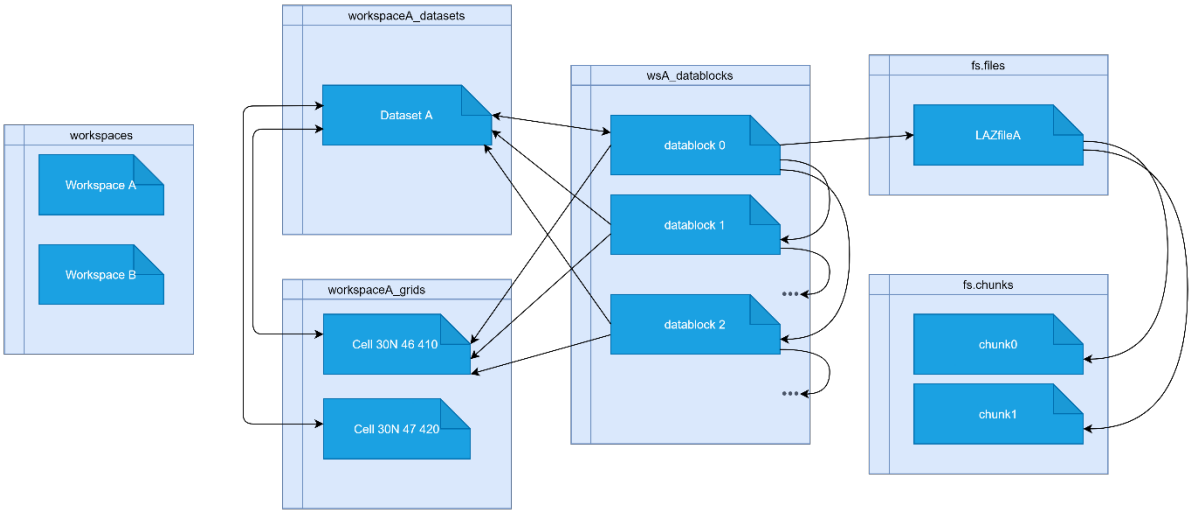

Ilustración 30. Estructura final de las colecciones de Mongo

Como se puede apreciar en el diagrama, los documentos de la colección workspace no mantienen una relación directa con los documentos de otras colecciones. La forma en que se vinculan es a través de los nombres de las propias colecciones: el documento Workspace A daría lugar a las colecciones *workspaceA_datasets, workspaceA_grids* y *workspaceA_datasets*. Todas las consultas que se lleven sobre las colecciones que utilizan este sistema de distribución constarán de dos partes: la primera, especifica los criterios de consulta dentro de una colección (p.ej., una búsqueda por nombre de un dataset), mientras que la segunda define cuál es la colección sobre la que aplicar dicha consulta.

La colección grids tendrá una relación bidireccional con datasets. Cada documento de la colección guarda una lista con los modelos con los que solapa. Esto agiliza las búsquedas por ventana espacial en el método #4 de la API. La relación de dataset a grid tienen un carácter auxiliar, guardando cada dataset una lista de las celdas del grid con las que está relacionado. La razón de esta dependencia viene dada por la implementación del método #9 de la API. Puesto que tenemos que generar un octree para cada celda del grid, y por lo tanto, tenemos que conocer cuáles son todas las celdas del grid para segmentar la nube de puntos original en diferentes regiones que posteriormente sirvan de base para construir los octrees, el mantener

una referencia directa desde el modelo a las celdas nos ahorra tener que recorrer toda la colección de grids al completo para encontrar cada una de las celdas que tienen una referencia al modelo.

La relación existente entre datasets y datablocks se dará tras la subida de una nube de puntos, de tal forma que un dataset guarde las referencias a uno o varios datablocks, los cuales corresponderán a los nodos raíces de los distintos octrees generados. De este modo, el cliente una vez recupere un dataset, conocerá los datablocks iniciales sobre los que comenzar a explorar el modelo. A su vez, cada datablock tendrá un campo en el que se referencie el identificador del dataset que le corresponda y la celda del grid en la que está ubicado. De este modo, con un índice secundario sobre el nodo del datablock y estos dos campos, se pueden resolver las llamadas de búsqueda de datablock necesarias en los métodos de la API #7 y #8. En este último, como pretendemos recuperar el fichero asociado al datablock, necesitaremos una operación adicional, recuperando la referencia del datablock corresponde a la colección fs.files. A partir de aquí, mediante GridFS se recuperará el contenido del fichero, utilizando las referencias del documento almacenado en fs.files a los contenidos almacenados en fs.chunks.

11.5.2 - Desarrollo de un cliente en Python

Uno de los objetivos que se definieron en el proyecto fue el desarrollo de un cliente. Por un lado, queremos tener una herramienta que nos permita interactuar con el sistema. Hasta el momento, se estaba utilizando Postman y los propios test unitarios de Java para llevar a cabo todas las pruebas necesarias. Sin embargo, el cliente pretende ser una herramienta más accesible y fácil de utilizar. Del mismo modo, también se pretende realizar una experimentación y obtención de resultados para validar la implementación. Los mismos métodos utilizados para llevar a cabo peticiones sobre el servidor se podrán reutilizar en distintos scripts que nos permitan automatizar la fase de experimentación.

El proyecto se ha desarrollado utilizando Python y la librería *Requests*. La implementación ha sido relativamente sencilla, generando un método específico que encapsula la llamada al servidor para cada uno de los métodos expuestos en la API. El principal problema detectado surgió al asociar un dataset de gran tamaño a la petición de subida de una nube de puntos. Por ejemplo, ante un dataset de unos 30 GBs de tamaño, el cliente lanzaba excepciones consecuencia de un consumo de memoria excesivo. La causa de esto viene dada por la incapacidad de esta librería de realizar un streaming de ficheros de forma nativa. La solución ha venido dada gracias a una librería complementaria a Requests, denominada *Requests Toolbelt*. Utilizando su clase *MultipartEncoder* y asignándole los ficheros que se quieran incluir en la petición, se procederá a realizar el envío de estos de forma progresiva.

Hay que tener en cuenta que este cliente no tiene una finalidad de visualizar nubes de puntos, sino simplemente interactuar con el sistema añadiendo metadatos, datasets y recuperando información, ya sea aquella referida a las entidades definidas en el sistema o los propios ficheros LAZ. Los ficheros LAZ que se descarguen pueden ser visualizados posteriormente con alguna herramienta adicional.

A continuación, se quiere mostrar cómo sería el resultado de interactuar con un dataset pequeño de una zona selvática (en torno a 250.000 puntos), asociándole un valor de máximo de puntos en datablock de 30.000. Mediante el cliente, se ha llevado a cabo el proceso de subida de información para posteriormente descargar una serie de ficheros LAZ, de tal forma que se explore hasta un nivel de profundidad 2 en una zona concreta, como se muestra en la siguiente ilustración.

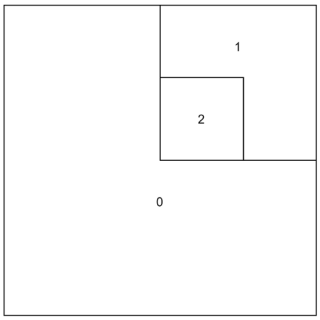

Ilustración 31. Niveles de profundidad solicitados al servidor.

Si bien nuestra estructura es un octree, como el dataset representa una zona del terreno que es más fácil de visualizar desde una perspectiva aérea, para el nivel de profundidad 1 y 2 se han descargado los dos nodos hijos que comparten la mismas coordenadas suroeste y noreste, diferenciándose solo por la altura. A continuación, se muestra una imagen que expone en la parte izquierda los puntos asociados a los ficheros descargados y en la derecha, la misma visualización pero acompañada de una guía que define las distintas zonas donde se ha profundizado.

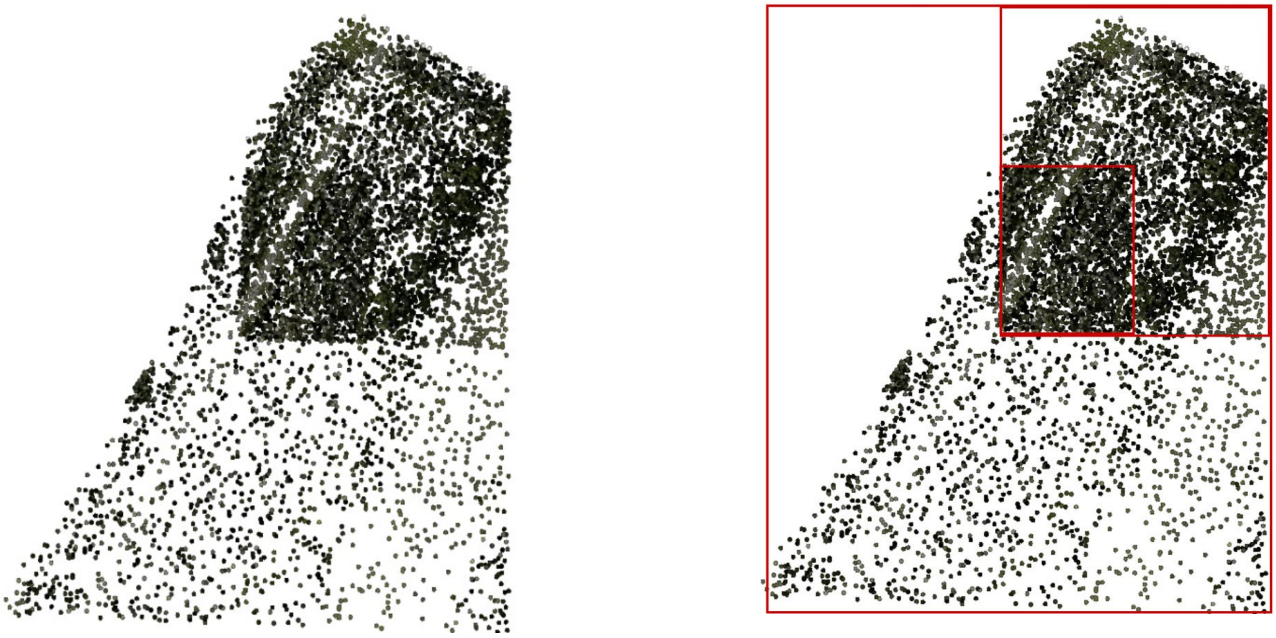

Ilustración 32. Visualización de la nube de puntos correspondiente a los ficheros solicitados.

11.5.3 - Métricas en servidor vía Actuator e índices en Mongo

En la primera tanda de experimentos que se lanzaron, se detectaron algunas anomalías en los tiempos de respuesta, con una gran discrepancia en los resultados en función del tamaño del dataset. Con el objetivo de obtener una visión más cercana de lo que estaba ocurriendo se implementó la medición de métricas también a nivel de servidor. Para ello, se ha utilizado el módulo *Actuator* de Spring. En Webflux,

podemos medir los datos de procesamiento asociados a diferentes métodos utilizando los métodos *name()* y *metrics()*. Estos métodos se invocan sobre una secuencia de operaciones realizadas sobre un publicador. Con *name()* especificamos el identificador que tendrá la métrica y con *metrics()* le indicamos a Spring que active su evaluación. Por ejemplo, para medir el tiempo empleado en el acceso a Mongo desde el repositorio en la búsqueda de un *datablock*, lo podríamos hacer de la siguiente manera:

```
return reactiveMongoTemplate.find(query, DatablockDBDTO.class, collection)
        .name("db.datablock.get")
        .metrics();
```

Los resultados son publicados bajo el endpoint de *actuator/metrics*. Quedarán registradas una serie de métricas asociadas a *db.datablock.get*, siendo la más relevante *db.datablock.get.flow.duration.* Esta nos indicará el tiempo total transcurrido en la realización de dichas operaciones, el número de veces que han sido invocadas y el tiempo máximo.


```
{
  "name": "db.datablock.get.flow.duration",
  "description": "Times the duration elapsed between a subscription and the onComplete termination of a sequence that did emit
some elements",
  "baseUnit": "seconds",
  "measurements": [
    {
      "statistic": "COUNT",
      "value": 1214.0
    },
    {
      "statistic": "TOTAL_TIME",
      "value": 0.9912665
    },
    {
      "statistic": "MAX",
      "value": 0.0
    }
  ],
```


Ilustración 33. Ejemplo de las métricas expuestas por el servidor a través de Actuator.

Mediante estas métricas se pudo comprobar que el sistema estaba pasando un tiempo excesivo en la consulta para la recuperación de un *datablock* en Mongo, y que el tiempo dependía del número de documentos almacenados en el sistema. La

causa de esto era el índice definido sobre la colección *ws_datablocks*, puesto que realmente no se estaba creando. Se procedió a realizar una implementación mediante enfoque programático de este índice para solucionar el problema en vez de utilizar el método basado en el uso de anotaciones que previamente se estaba utilizando. Con este índice correctamente implementado, la búsqueda, especialmente en los casos en los que el tamaño de *datablock* es reducido y, en consecuencia, se almacenan más *datablocks* en el sistema, los tiempos de recuperación de *datablock* mejoraron drásticamente.

11.5.4 - Inclusión de lasoptimize en el flujo de procesamiento de los ficheros

Uno de los principales defectos encontrados durante la fase de experimentación realizada durante esta iteración fue el excesivo tamaño ocupado en disco por los datos generados tras el proceso de procesamiento y almacenamiento de una nube de puntos. Si comparamos el tamaño de los ficheros originales con el de los datos resultantes en la base de datos a través de la siguiente gráfica, podemos determinar como el espacio necesario en disco casi se duplica.

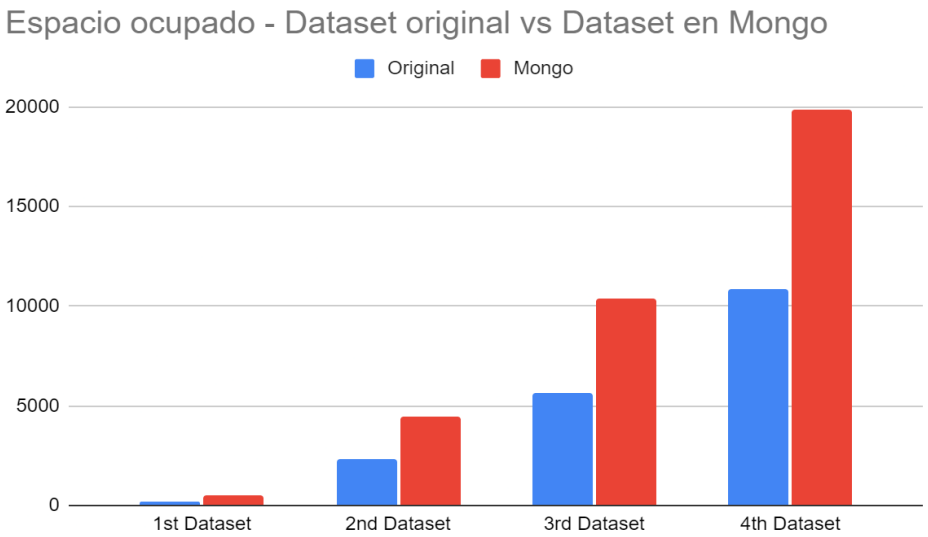

Ilustración 34. Comparativa del espacio ocupado por el dataset en su versión original y su versión sin optimizar en Mongo.

A raíz de esto, se realizaron múltiples pruebas en Mongo modificando el tamaño de chunk, pero el espacio ocupado seguía siendo esencialmente el mismo. En este momento se detecta que el tamaño de los ficheros generados, tras combinar mediante LASTools aquellos archivos cuyo fin es ser almacenados en el sistema de bases de datos, es considerablemente mayor que el dataset original, pese a que la información contenida es la misma, representando el mismo número de puntos. Sospechamos que los ficheros generados en la fase de inserción del dataset no estaban siendo creados del modo más eficiente, por lo que se decidió aplicar la herramienta lasoptimize, también incluida en el kit de LASTools, sobre los ficheros que van a ser almacenados en Mongo, incluyendo un paso adicional dentro de la creación del octree que haga este proceso. El resultado es una mejora muy considerable a nivel de espacio ocupado, alcanzando unos niveles prácticamente idénticos a los del dataset original y siendo la ligera diferencia existente todavía entre ambos valores consecuencia de que, al haber un mayor número de ficheros, cada uno tendrá su propia cabecera y metadatos adicionales intrínsecos al estándar LAS, además de que el número de datablocks almacenados será mayor puesto que el octree generado será más complejo.

12- Experimentación y resultados

Con el objetivo de validar la implementación realizada y obtener una visión general del rendimiento del sistema, se ha llevado un proceso de experimentación. Si bien el sistema desarrollado se trata de un prototipo y se podría someter a mejoras tanto en la aplicación como en la configuración de la base de datos, se pretende tener unos resultados que sirvan para demostrar que este funciona y podría operar en un entorno controlado, avalado por una serie de datos que muestren cuál sería su capacidad a la hora de añadir nubes de puntos y satisfacer las consultas sobre estas de los usuarios.

Los tests realizados se han dividido en dos tipos: el primero pretende medir el rendimiento del sistema en su interacción con un único usuario, que realizará un proceso de inserción de un dataset y posteriormente, procederá a solicitar un presupuesto de puntos que se traducirá en un número de peticiones de ficheros LAZ; en el segundo tipo de test, se pretende evaluar el rendimiento del sistema al interactuar con múltiples usuarios de forma concurrente, que realizarán consultas de recuperación de ficheros.

12.1 - Datasets utilizados

En estos tests se experimentará con 4 datasets. Estos tienen las siguientes características:

Datasets	Dataset I	Dataset II	Dataset III	Dataset IV
Número de puntos	30.401.539	244.894.563	534.073.172	1.068.146.345
Espacio ocupado sin optimizar (MBs)	175	2326	5626	10.851
Espacio ocupado optimizado (MBs)	138	1850	4085	8055
Celdas del grid generadas	3	4	4	4
Formato de punto LAS	3	8	8	8

Tabla 8. Características de los datasets utilizados.

Todos los tests representan información de la ciudad de Pamplona. El primer dataset posee información del año 2011, mientras que el resto son de 2017. A todos los datasets se les ha asignado a un workspace con un tamaño de celda de 10.000 metros. El dataset I y el dataset II representan las mismas regiones de la ciudad, motivo por el que solo se generan 3 celdas, ya que en la celda del grid que compondría el cuadrante inferior izquierdo no hay puntos. El dataset III y el dataset IV por su parte sí que tendrán puntos distribuidos por toda el área.

A continuación, se muestran unas imágenes para cada dataset. Si bien visualmente el dataset I y el II se pueden diferenciar por los colores, el III y el IV parecen idénticos, ya que la única diferencia es la densidad de puntos, difícil de apreciar en una vista aérea y desde una perspectiva lejana. Las visualizaciones se han obtenido mediante la herramienta lasview de LASTools.

Ilustración 35. Vista cenital del dataset I.

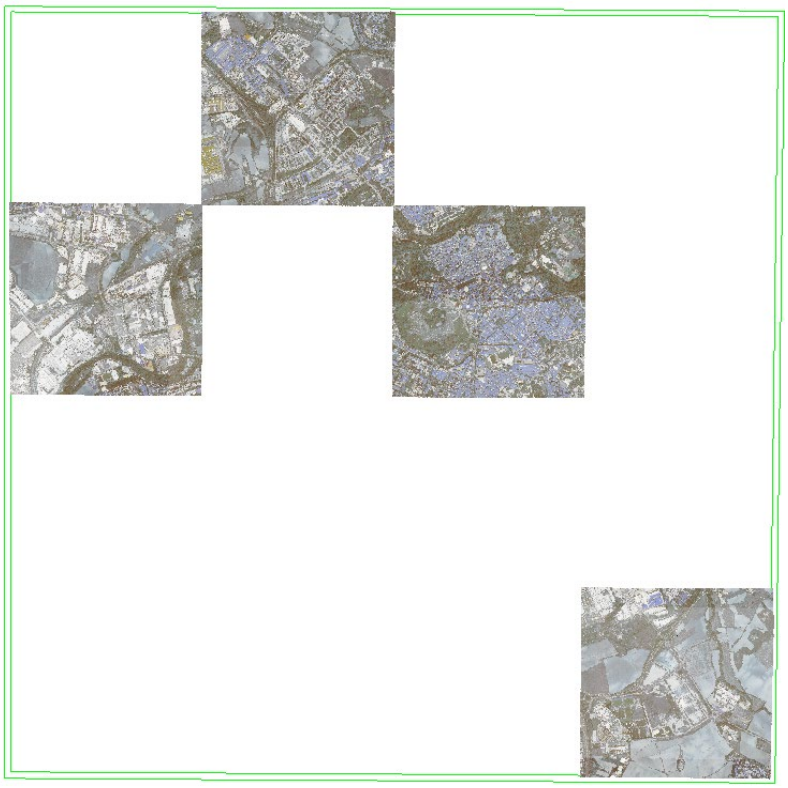

Ilustración 36. Vista cenital del dataset II.

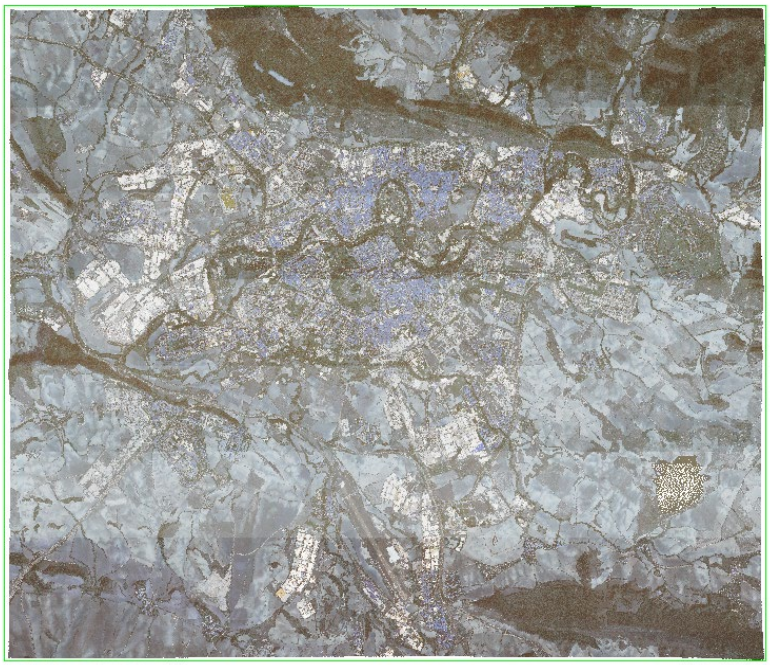

Ilustración 37. Vista cenital del dataset III.

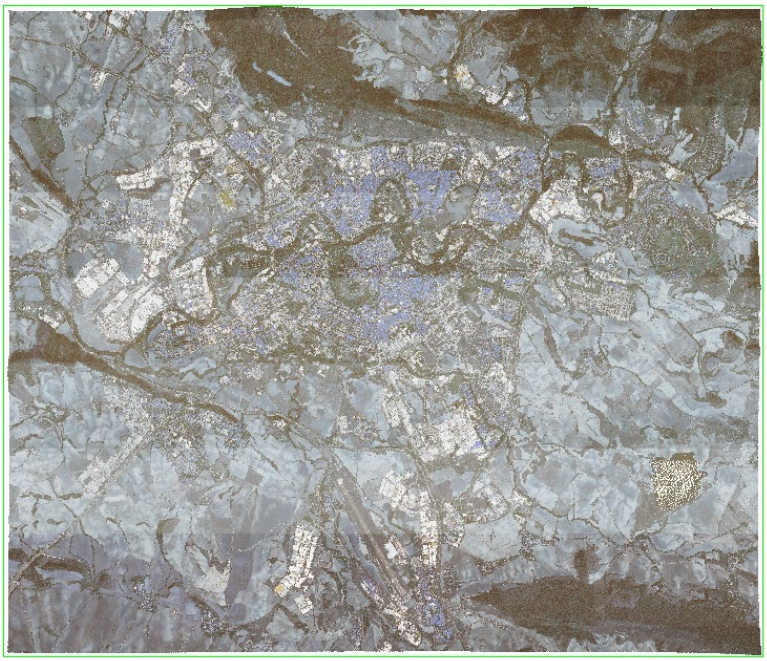

Ilustración 38. Vista cenital del dataset IV.

12.2 - Tests monousuario

En estos tests se realizará para cada uno de los datasets presentados anteriormente un proceso de inserción, variando el tamaño de máximo de datablock (el número de puntos que tendrá como máximo los ficheros que se generen y se asocien a cada nodo del octree) y el tamaño máximo de profundidad del octree. Para cada configuración se lleva a continuación un proceso de petición de ficheros, buscando llegar a un presupuesto de 20 millones de puntos, de tal forma que una vez se descarguen tantos ficheros como sea necesario para superar ese umbral, se detendrá la fase de descarga.

Como resultado de la inserción se obtendrán los siguientes datos:

1. Tiempo total utilizado en la inserción del dataset
2. Número de nodos generados
3. Máxima profundidad del octree

4. Tamaño ocupado por la base de datos.

Como resultado de la petición de ficheros se obtendrán las siguientes medidas:

1. Tiempo total de descarga empleado
2. Número de ficheros descargados
3. Tiempo medio de descarga de fichero
4. Número de puntos descargados
5. Tiempo medio de descarga por punto

12.2.1 - Dataset I

Tamaño máximo datablock	10.000	100.000	1.000.000	5.000.000
Tiempo de inserción (s)	1361,7755	210,7225	62,2726	34,3417
Nº de datablocks generados	9181	941	94	20
Profundidad máxima	7	5	3	1
Tiempo de descarga empleado (s)	23,7434	3,2786	1,4129	1,052
Nº de ficheros descargados	6351	606	53	14
Tiempo medio descarga fichero (s)	0,0037	0,0054	0,0267	0,0751
Nº de puntos descargados	20000883	20024282	20210074	20693902
Tiempo medio descarga punto (s)	1,19E-06	1,97E-07	6,99E-08	5,08E-08
Tiempo medio descarga 1-M puntos (s)	1,19	0,20	0,07	0,05
Tamaño ocupado (MBs)	235	202	198	174

Tabla 9. Resultados del test monousuario sobre el dataset I.

12.2.2 - Dataset II

Tamaño máximo datablock	10.000	100.000	1.000.000	5.000.000
Tiempo de inserción (s)	12530,91	2035,5732	823,0036	634,4885
Nº de datablocks generados	86840	8070	899	154
Profundidad máxima	8	6	5	3
Tiempo de descarga empleado (s)	28,7477	3,7792	1,7437	1,4329
Nº de ficheros descargados	7378	594	62	11
Tiempo medio descarga fichero (s)	0,0039	0,0064	0,0281	0,1303
Nº de puntos descargados	20005752	20087909	20298978	22211365
Tiempo medio descarga punto (s)	1,44E-06	1,88E-07	8,59E-08	6,45E-08
Tiempo medio descarga 1-M puntos (s)	1,44	0,19	0,09	0,06
Tamaño ocupado (MBs)	2162	1893	1875	1880

Tabla 10. Resultados del test monousuario sobre el dataset II.

12.2.3 - Dataset III

Tamaño máximo datablock	10.000	100.000	1.000.000	5.000.000
Tiempo de inserción (s)	22757,5924	3751,1987	1833,3941	1463,196
Nº de datablocks generados	171184	17489	1872	365
Profundidad máxima	8	6	4	3
Tiempo de descarga empleado (s)	26,4823	4,2302	1,8186	1,875
Nº de ficheros descargados	6344	644	74	5
Tiempo medio descarga fichero (s)	0,0042	0,0066	0,0246	0,375
Nº de puntos descargados	20000198	20002592	20458993	23688035
Tiempo medio descarga punto (s)	1,32E-06	2,11E-07	8,89E-08	7,92E-08
Tiempo medio descarga 1-M puntos (s)	1,32	0,21	0,09	0,08
Tamaño ocupado (MBs)	5062	4313	4233	4209

Tabla 11.Resultados del test monousuario sobre el dataset III.

12.2.4 - Dataset IV

Tamaño máximo datablock	10.000	100.000	1.000.000	5.000.000
Tiempo de inserción (s)	41183,7195	7975,1991	3936,6701	3377,5501
Nº de datablocks generados	360506	35081	3478	736
Profundidad máxima	8	7	5	4
Tiempo de descarga empleado (s)	33,0441	3,9259	1,7505	1,3365
Nº de ficheros descargados	8048	661	66	7
Tiempo medio descarga fichero (s)	0,0041	0,0059	0,0265	0,1909
Nº de puntos descargados	20006210	20075168	20638700	20127133
Tiempo medio descarga punto (s)	1,65E-06	1,96E-07	8,48E-08	6,64E-08
Tiempo medio descarga 1-M puntos (s)	1,65	0,20	0,08	0,07
Tamaño ocupado (MBs)	9615	8392	8274	8250

Tabla 12.Resultados del test monousuario sobre el dataset IV.

12.2.5 -Discusión de resultados

Para la discusión de resultados, se van a mostrar algunas gráficas generadas a partir de los datos expuestos en las tablas anteriores.

Los tiempos de inserción dependen claramente del tamaño del dataset y el tamaño máximo de datablock: si debemos procesar una nube de puntos de gran volumen y el tamaño de partición del nodo es reducido, el octree que se genere será más complejo, con un mayor número de nodos, aumentando el número de llamadas a las operaciones de sampleado y partición basadas en LASTools, las cuales son bastante costosas computacionalmente.

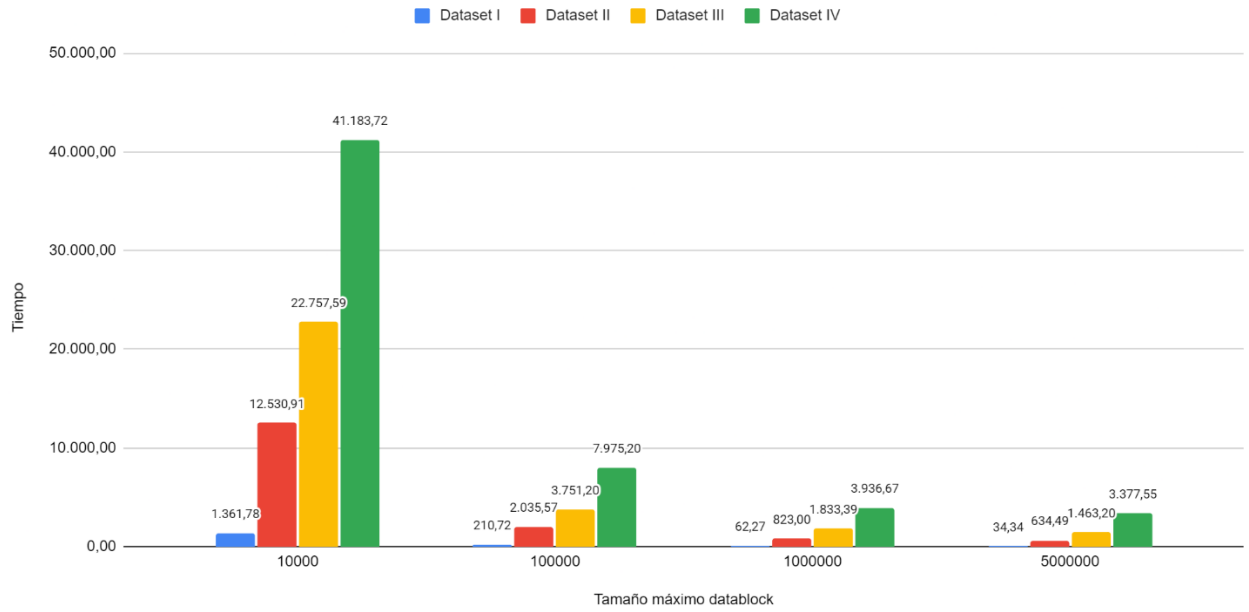

Ilustración 39. Gráfica comparativa de los tiempos de inserción entre datases.

Dentro de los dos parámetros que afectan al aumento de los tiempos de inserción, el comportamiento es distinto. En la siguiente gráfica se muestra la evolución del tiempo de inserción solo para del dataset III en función al tamaño máximo de datablock.

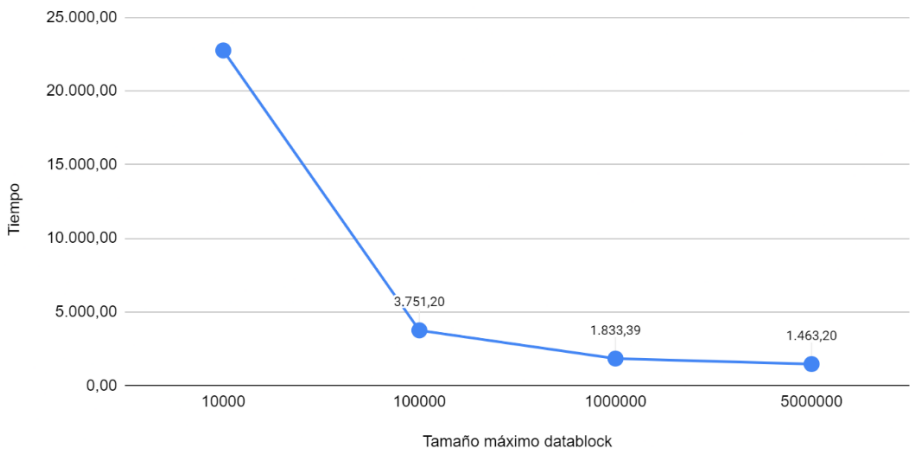

Ilustración 40. Evolución de los tiempos de inserción en el dataset III para tamaño variante de datablock.

En ella se puede apreciar que el tamaño de datablock aumenta considerablemente al usar valores pequeños. Si bien el salto de un tamaño de 1 millón de puntos a 100.000 supone duplicar el tiempo empleado, el salto de 100.000 a 10.000 provoca que el tiempo de inserción sea casi 10 veces mayor.

En la siguiente gráfica se representa la evolución de los tiempos de inserción en función del tamaño del dataset. Los resultados son los obtenidos para un tamaño de datablock máximo de 100.000. Aquí sí que podemos ver que el incremento es prácticamente lineal: si el tamaño de la nube de puntos se duplica, el tiempo de inserción necesario lo hace en la misma proporción.

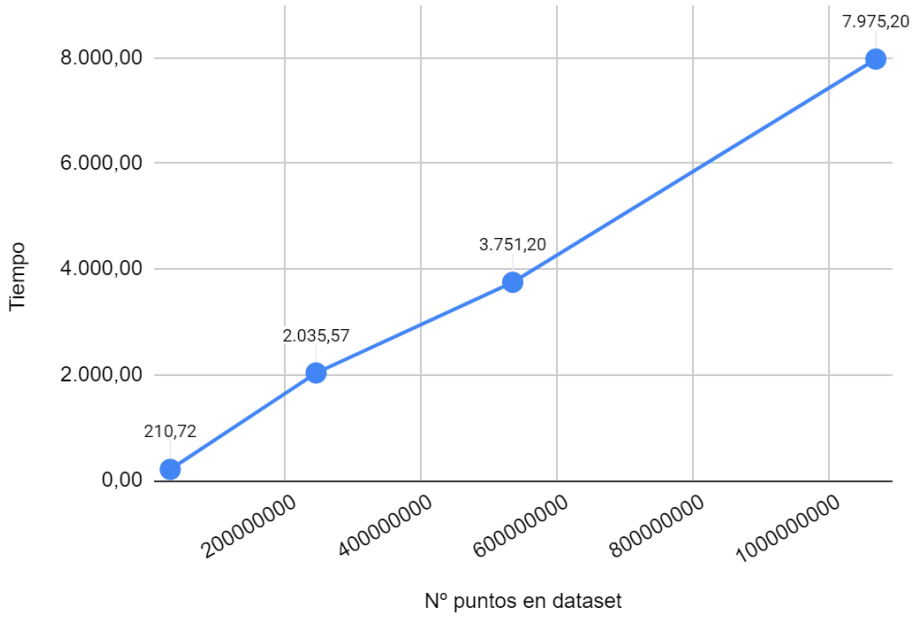

Ilustración 41. Evolución de los tiempos de inserción en función del tamaño de la nube de puntos.

A continuación, vamos a comparar el almacenamiento ocupado por cada dataset en función del tamaño de datablock empleado. En la siguiente gráfica podemos ver los datos presentados en las tablas para todos los datasets:

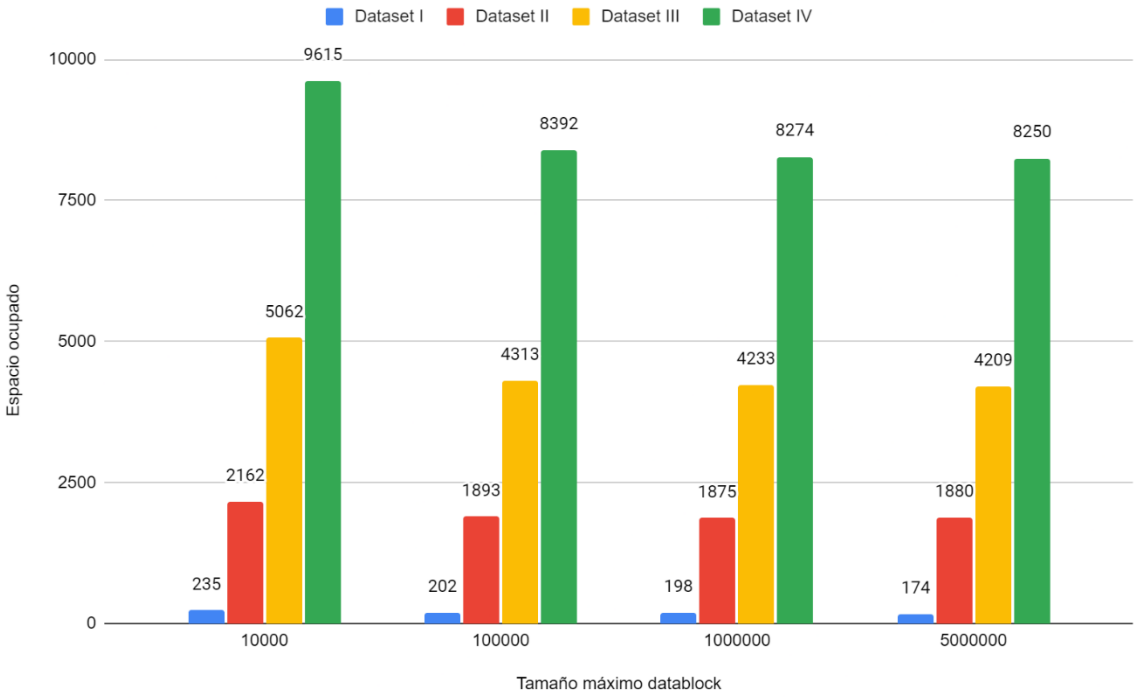

Ilustración 42. Gráfica comparativa del espacio ocupado en Mongo.

Los valores de almacenamiento se mantienen relativamente constantes. En todos los datasets se sigue una lógica similar, de tal forma que para tamaños de datablock 100.000, 1.000.000 y 5.000.000 el espacio ocupado va por lo general aumentando de forma muy ligera a medida que decrece el tamaño máximo de datablock. Para el tamaño de datablock de 10.000, el espacio requerido vuelve a aumentar, esta vez de forma algo más abrupta. Este incremento supone en todos los datasets un incremento de en torno al 15% con respecto al espacio ocupado para el tamaño de 100.000.

En cuanto a la obtención de ficheros, los principales aspectos que nos interesan son definir cuál es el tamaño más óptimo de los utilizados, además de comprobar que el sistema funcione de manera correcta tanto en datasets pequeños como en aquellos que tengan asociados un gran número de puntos. En la siguiente gráfica se muestran los valores obtenidos:

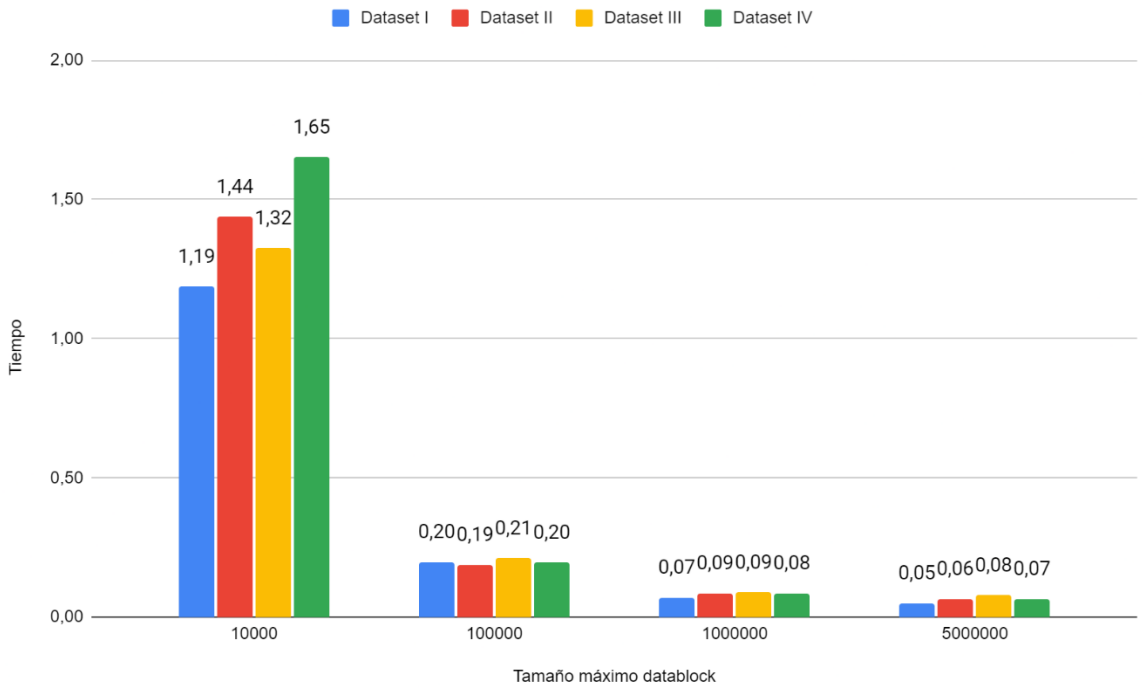

Ilustración 43. Gráfica comparativa de los tiempos de recuperación de 1 millón de puntos.

Los tiempos por lo general se mantienen bastante parejos independientemente del dataset. El dataset I, que es el que menos puntos tiene, tiende a ser el que mejor rendimiento ofrece. Esto se puede explicar debido a que las colecciones generadas en Mongo para los datablock son más reducidas y, en consecuencia, los tiempos de respuesta de la base de datos son mejores. Aun así, el incremento de tiempos parece bastante asumible y solo notorio cuando se usa un tamaño máximo de datablock de 10.000. El tamaño de datablock parece el principal causante del incremento en los tiempos. Cuanto más pequeño sea este valor, más ficheros habrá que solicitar al servidor para llegar a la cifra del millón de puntos, que en consecuencia supone un mayor número de accesos a la base de datos y un mayor número de peticiones al servidor.

12.3 - Tests multiusuario

Los siguientes tests buscan analizar la capacidad del sistema de soportar diferentes cargas de usuarios a la hora de solicitar ficheros. Para ello, se ha utilizado la herramienta *Loadtest* (https://www.npmjs.com/package/loadtest), un paquete de NodeJS para llevar a cabo pruebas de concurrencia.

Loadtest es una utilidad de linea de comandos en la que podemos especificar el número de peticiones totales a realizar y el número de peticiones concurrentes, mostrando como resultado una serie de datos estadísticos sobre el rendimiento del sistema. Los comandos de ejecución siguen el formato *loadtest -n requests -c concurrency URL*, donde en -n especificamos el número total de peticiones, en -c el número de usuarios concurrente, y en URL el endpoint a testear.

Se llevarán a cabo dos experimentos, probando con 10 usuarios concurrentes y posteriormente con 100. Se realizará un número de peticiones que simule que cada uno de estos usuarios alcanza un millón de puntos. Todos los tests se van a realizar en el primer dataset, variando el tamaño máximo de datablock empleado en 10.000, 100.000 y 1.000.000, de tal forma que la petición de fichero vaya dirigida a recuperar el nodo raíz de una de las tres celdas del grid generadas.

Las medidas obtenidas son los siguientes:

1. Tiempo medio de petición
2. Tiempo máximo empleado en una petición
3. Productividad del sistema (número de peticiones por segundo)
4. Tiempo total necesario para completar todas las peticiones

Peticiones	10 usuarios concurrentes, 1-M puntos cada uno		
Nº de puntos en datablock	10.000	100.000	1.000.000
Nº de peticiones	1.000	100	10
Tiempo medio de respuesta	0,0276	0,206	1,603
Tiempo máximo de respuesta	0,058	0,378	1,628
Productividad (peticiones x segundo)	359	49	6
Tiempo total	2,78	2,06	1,629

Tabla 13. Resultados del test multiusuario para 10 usuarios concurrentes.

Peticiones	100 usuarios concurrentes, 1-M puntos cada uno		
Nº de puntos en datablock	10.000	100.000	1.000.000
Nº de peticiones	10.000	1.000	100
Tiempo medio de respuesta	0,2518	2,204	15,77
Tiempo máximo de respuesta	0,489	3,807	15,96
Productividad (peticiones x segundo)	396	42	6
Tiempo total	25,27	23,89	15,99

Tabla 14. Resultados del test multiusuario para 100 usuarios concurrentes.

A partir de los datos obtenidos podemos generar algunas conclusiones. Por ejemplo, a nivel de productividad, el número de peticiones por segundo atendidas se mantiene es bastante similar, independientemente de que haya 10 usuarios o 100 realizando peticiones, lo cual nos puede indicar que el sistema puede funcionar bien en situaciones con una carga alta de usuarios:

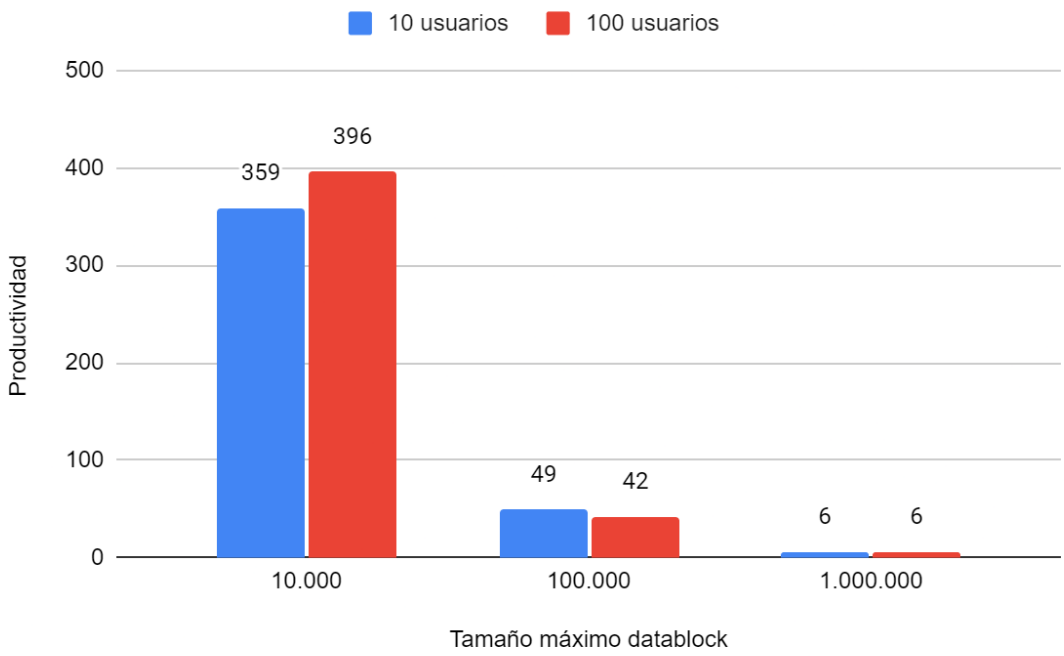

Ilustración 44. Gráfica comparativa del número de peticiones atendidas por segundo del sistema.

Si nos fijamos en el tiempo total empleado en satisfacer todas las peticiones, vemos que con 100 usuarios los valores se multiplican por aproximadamente 10 en comparación con los valores obtenidos para 10 usuarios. Si calculamos el tiempo medio en completar las peticiones de un usuario, entendido como el tiempo total necesario dividido entre el número de usuarios, vemos que los valores se mantienen muy similares, independientemente del número de usuarios.

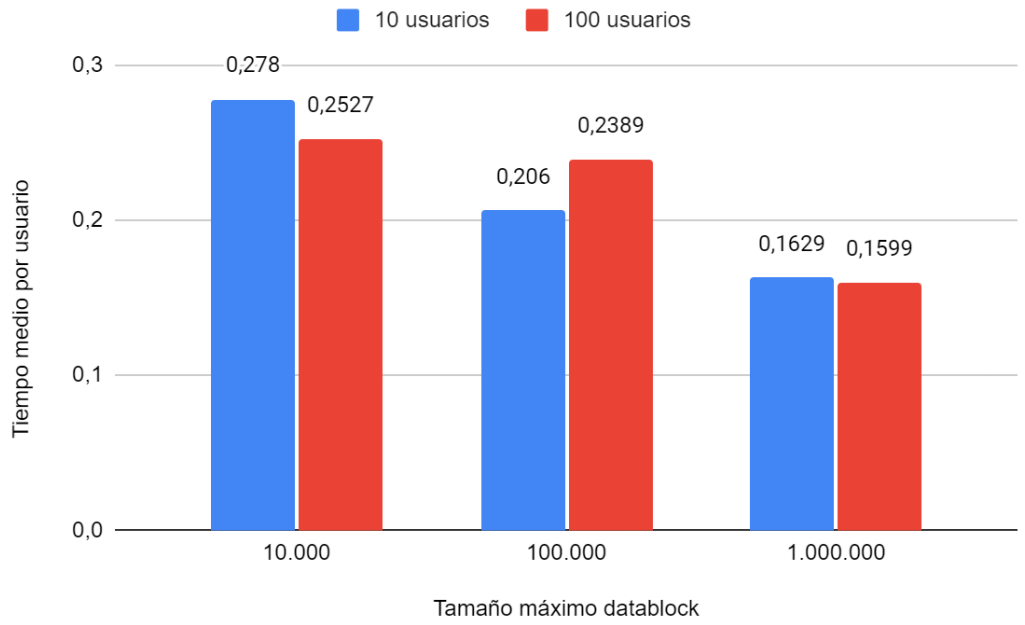

Ilustración 45. Gráfica comparativa del tiempo medio para completar las peticiones de un usuario.

13- Trabajos futuros

Una vez constituido este primer prototipo de la aplicación, podemos pensar en distintos puntos sobre los que trabajar de cara al futuro, tanto a nivel de experimentación como de optimización del sistema y funcionalidad.

Una de las ideas propuestas consiste en experimentar con diferentes sistemas de bases de datos, que nos permitan llevar a cabo distintos estudios de rendimiento del sistema utilizando otras tecnologías análogas a Mongo. De este modo, se podría modificar la capa de persistencia, añadiendo nuevos repositorios que implementen las interfaces definidas en la aplicación para bases de datos noSQL como Cassandra o experimentando con bases de datos relacionales como PostgreSQL. Si bien por la relación del problema que estamos tratando, encuadrado dentro del Big Data, se presupone que el rendimiento deberá ser mejor utilizando bases de datos noSQL, puede merecer la pena realizar esta comparación y analizar si en algunos casos específicos (por ejemplo, octrees de poca profundidad que no impliquen tablas/colecciones de gran volumen) una base de datos SQL podría tener un mejor rendimiento.

Otro de los puntos donde existe un margen de mejora pasa por la inserción de los datasets. Como quedó expuesto durante la fase de experimentación, los tiempos necesarios para procesar un dataset y convertirlo en la estructura final de ficheros que queda almacenada en Mongo pueden ser muy altos al trabajar con grandes volúmenes de puntos y tamaños de datablock pequeños. Pese a que las LASTools son un kit de herramientas muy robusto y sólido, se requieren una gran cantidad de invocación de procesos distintos para generar cada fichero que necesitamos. Se podría plantear generar una implementación propia en C++, utilizando la propia librería LASLib que emplea LASTools, que trate de sintetizar algunas operaciones en una sola llamada. Del mismo modo, si en un futuro se amplían las funcionalidades ofrecidas por alguna de las librerías implementadas en Java para ficheros LAZ/LAS, se podría incorporar al sistema la escritura de ficheros mediante estas.

Finalmente, se propone de cara al futuro transicionar la aplicación desarrollada a una arquitectura basada en microservicios. Si bien la aplicación actual envuelve funcionalidades lo suficientemente relacionadas entre sí como para justificar que constituyan una aplicación única, se puede plantear una mejora del diseño que mejore la productividad del sistema y su escalabilidad, integrando algunas de las herramientas del módulo de Spring Cloud que permitan replicación y descubrimiento de servicios de forma automática. Por ejemplo, podríamos tener distintos microservicios para la inserción de datasets y recuperación de ficheros, de tal forma que, en función de la demanda, se pongan en marcha nuevas instancias que den soporte a los usuarios de forma automática.

Anexo I - Instalación y despliegue

Para desplegar la aplicación en un nuevo equipo se ha utilizado Docker. Se ha generado una imagen propia basada en Ubuntu 20.04 que incorpora una instalación de LASTools, Java 11 y Python 3.8, a la que se le añaden los archivos correspondientes a los dos scripts de Python necesarios para acceder a la bounding box de un fichero y al número de puntos, junto al .jar de la aplicación en Java. El Dockerfile definido para generar esta imagen se ha incluido junto al resto del código fuente presente en la entrega de este TFM. Este Dockerfile está diseñado para ejecutarse desde la carpeta raíz de la aplicación, utilizando el comando docker run. Previamente se habrá debido generar el .jar correspondiente a partir de Maven.

Una vez ha sido generada la imagen correspondiente a la aplicación principal, la cuál ha sido nombrada *spslidarserver*, se debe ejecutar un archivo docker-compose.yml, que se encarga de gestionar dos contenedores, uno correspondiente a la ya mencionada aplicación y otro para la base de datos, en este caso Mongo en su versión 4.2.10. El fichero docker-compose.yml tendrá la siguiente forma (también ha sido añadido junto al resto del código fuente):

```
version: "3"
services:
 mongo-database:
   image: "mongo:4.2.10"
   container_name: "mongo-database"
   ports:
     - "27017:27017"
 spslserver:
   image: "spslidarserver"
   container_name: "spslidarserver"
   ports:
     - "8080:8080"
   links:
     - "mongo-database"
```

Utilizando docker-compose up ponemos en marcha la aplicación. A partir de este momento podemos empezar a interactuar con el sistema.

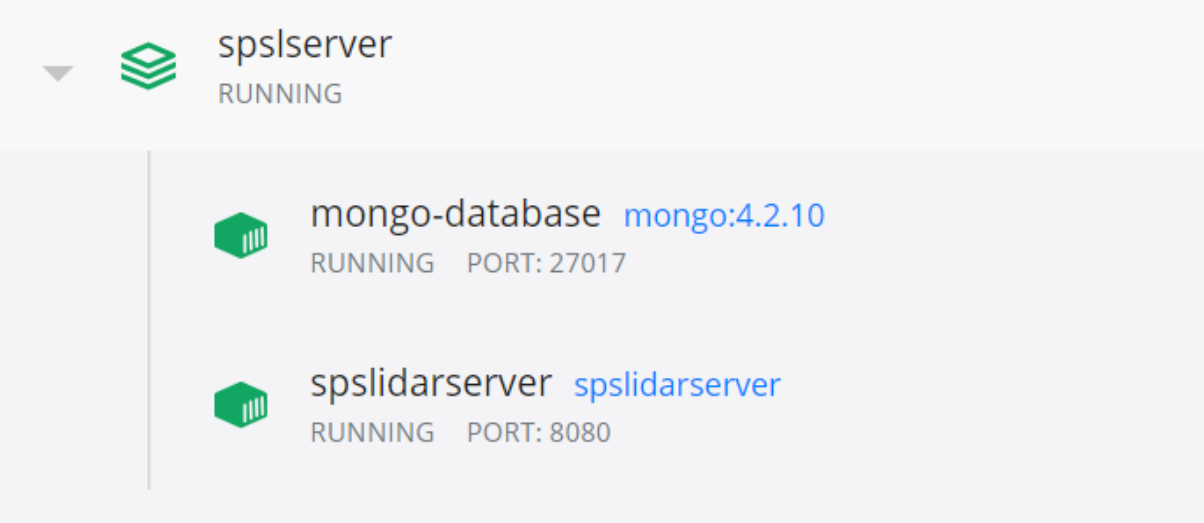

Ilustración 46. Vista de la aplicación ejecutándose desde Docker Desktop.

Anexo II - Diagramas

En este anexo se pretenden incluir distintos diagramas UML que permitan comprender de mejor manera la arquitectura y funcionalidades del sistema.

Diagrama de paquetes

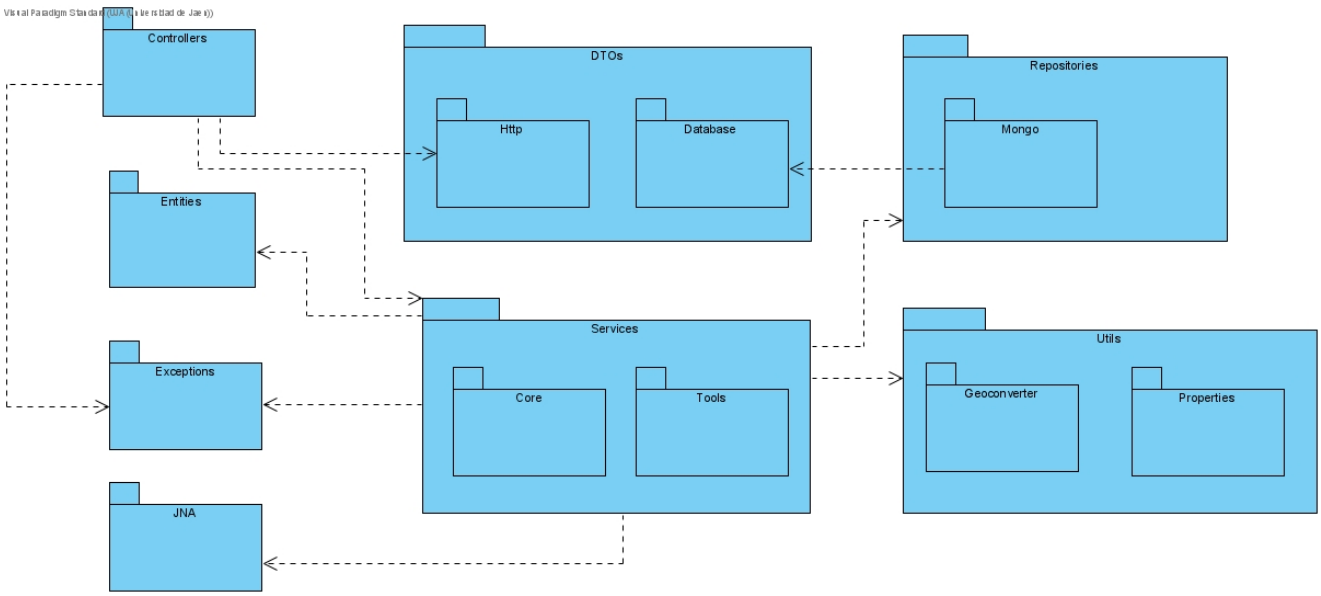

Ilustración 47. Diagrama de paquetes del sistema.

Diagrama de clases

Diagrama de clases del paquete Controllers

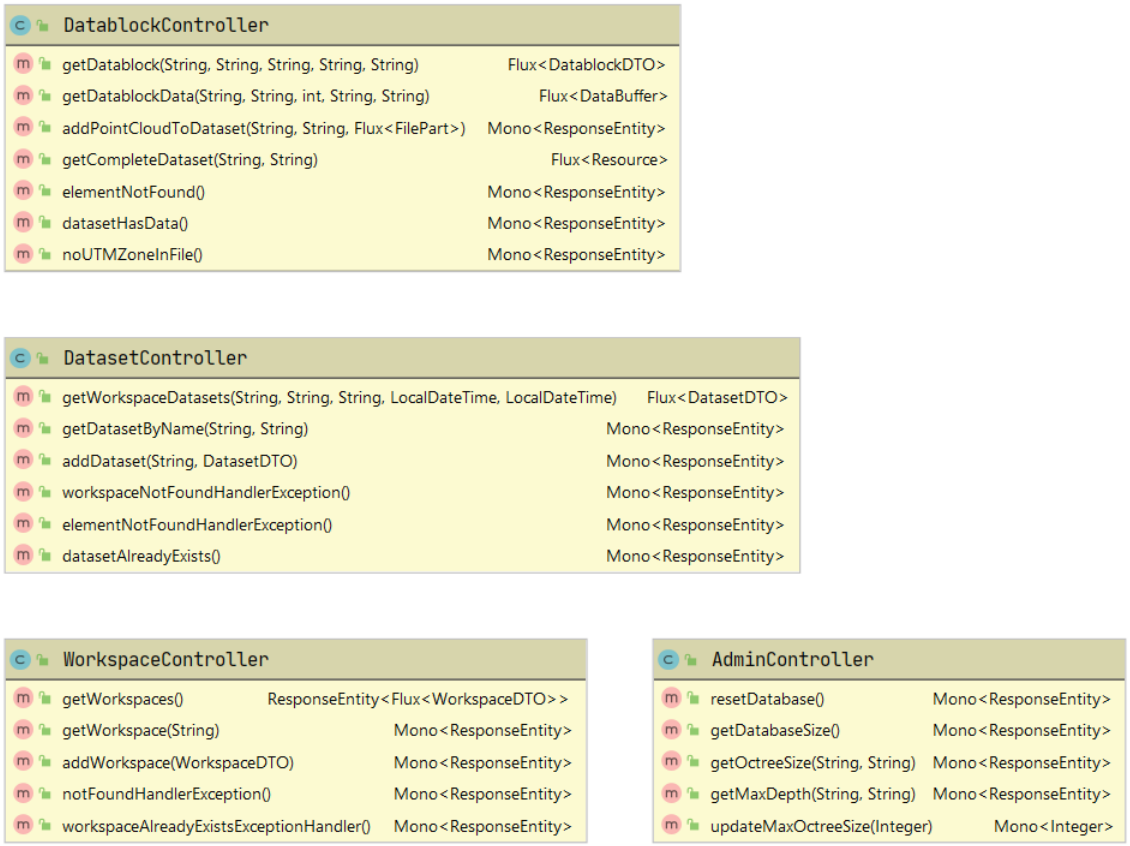

Ilustración 48. Diagrama de clases del paquete spslidar.controllers.

Diagrama de clases del paquete Repositories

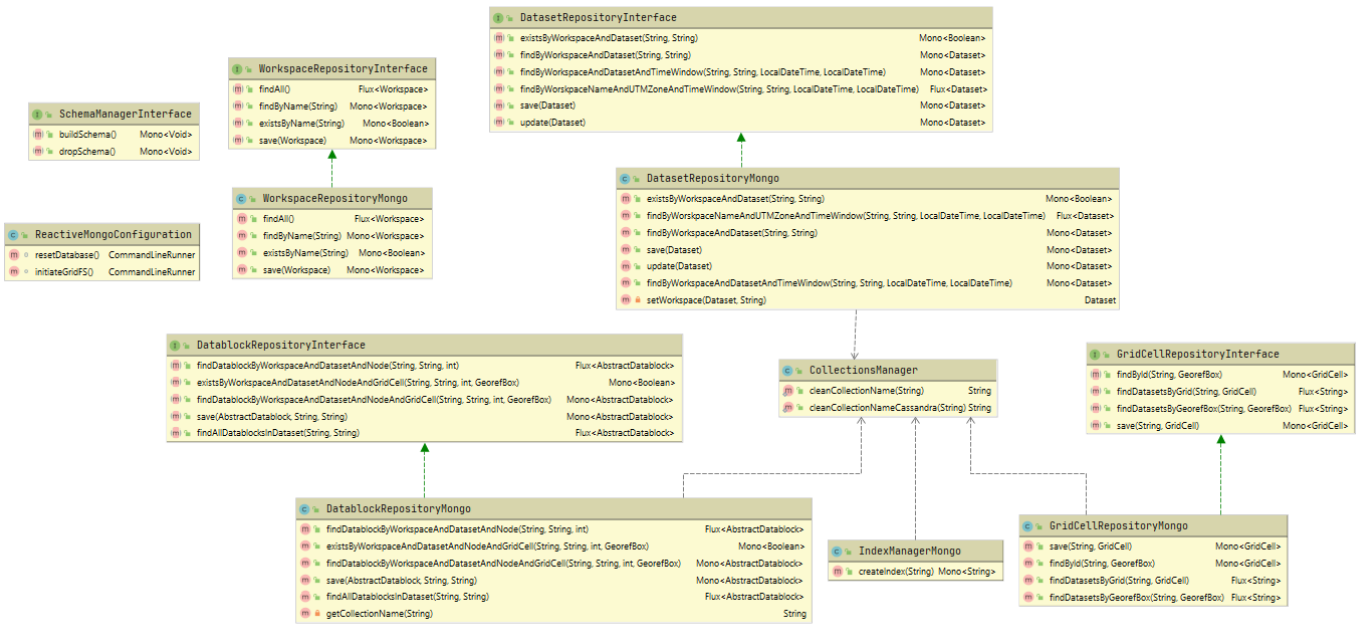

Ilustración 49. Diagrama de clases del paquete spslidar.repositories.

Diagrama de clases del paquete Services

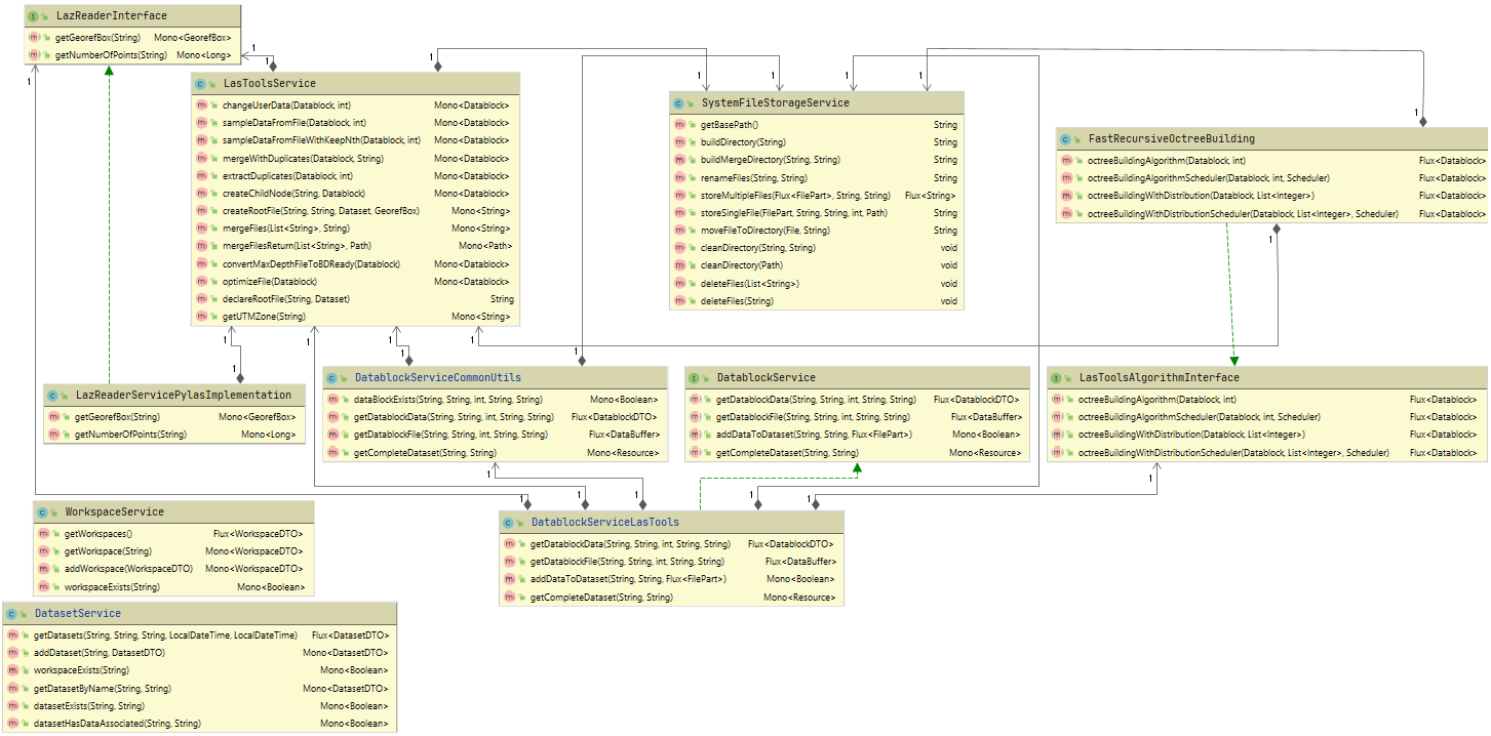

Ilustración 50. Diagrama de clases del paquete spslidar.services.

Diagrama de clases del paquete Entities

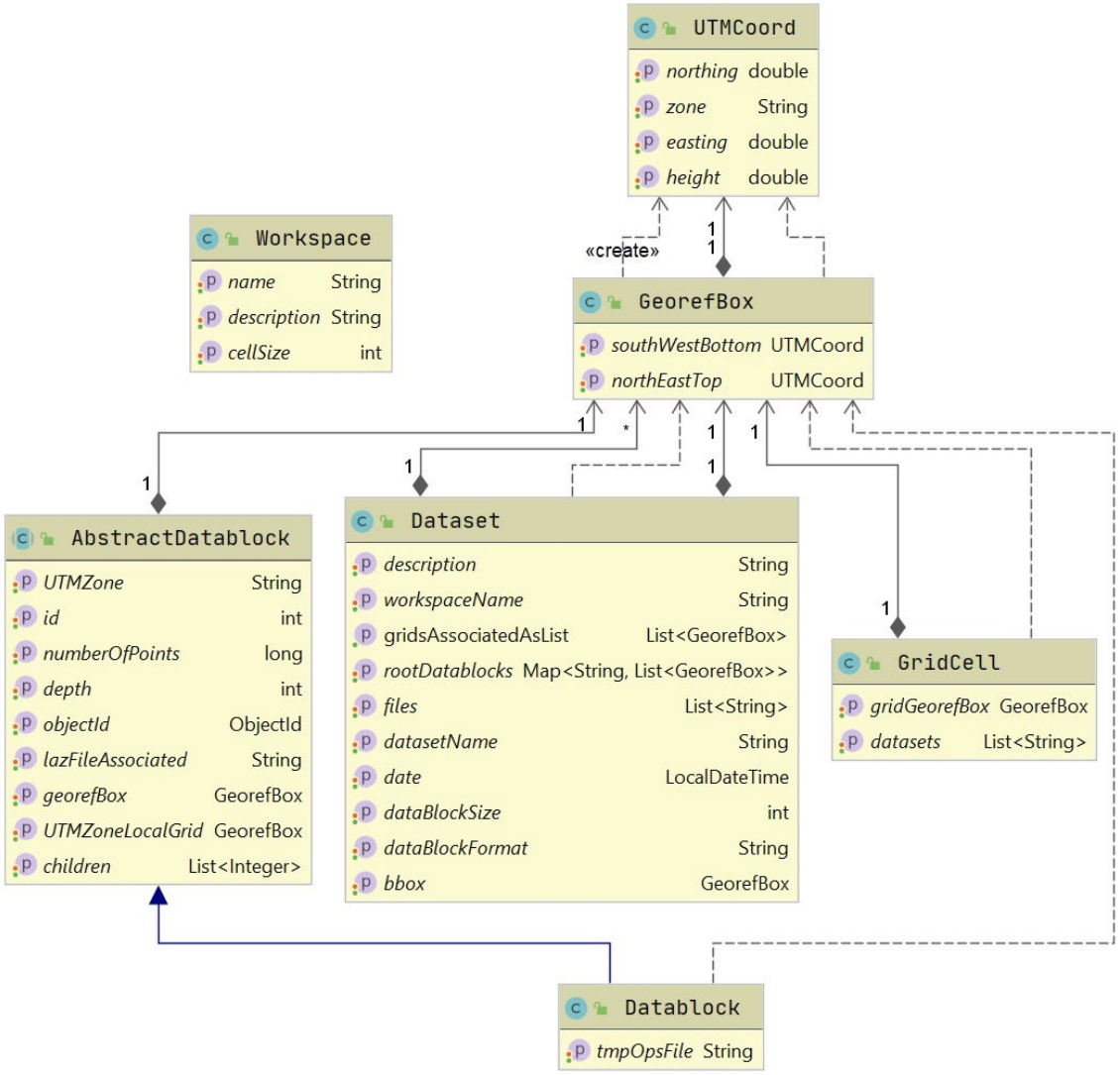

Ilustración 51. Diagrama de clases del paquete spslidar.entities.

Diagrama de clases del paquete Tests

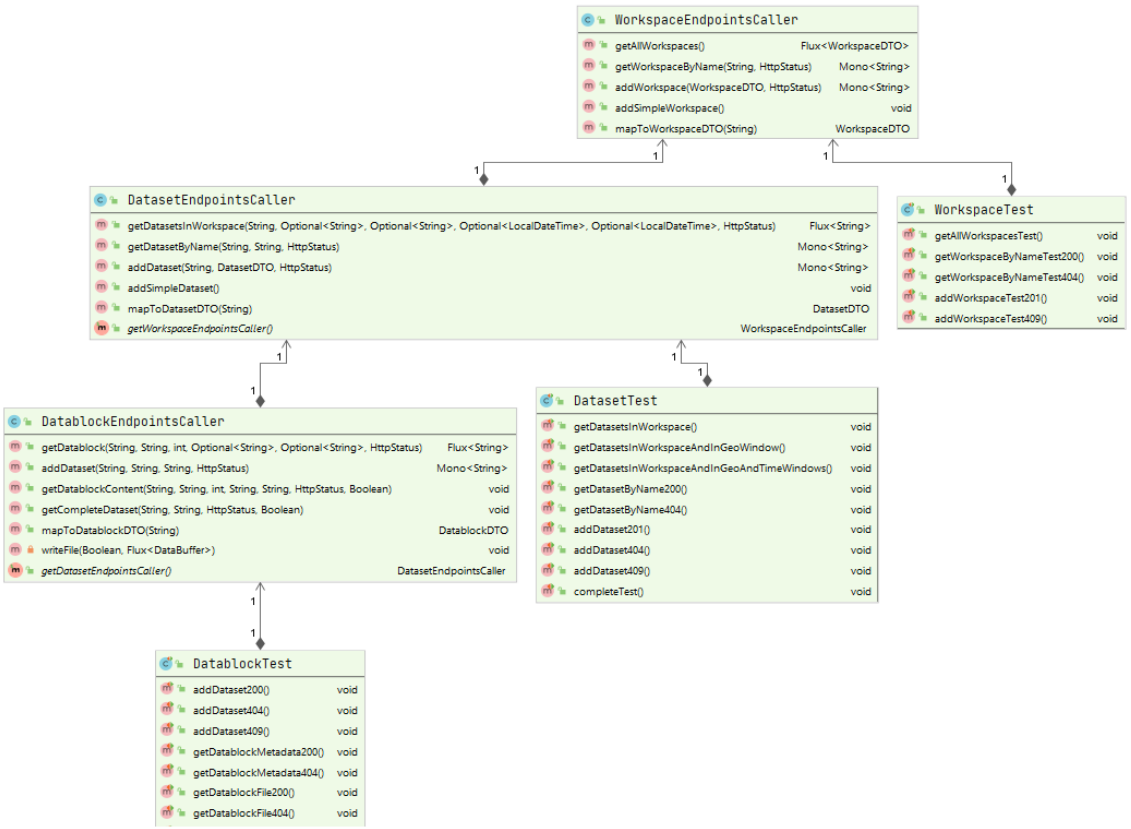

Ilustración 52. Diagrama de clases del paquete tests.

Anexo III - Spring Webflux

En este anexo se pretende hacer una breve introducción a Spring Webflux y presentar algunos de los elementos claves que aparecen recurrentemente a lo largo de la implementación.

Spring Webflux utiliza Project Reactor como implementación del estándar Reactive Streams. Se utiliza un modelo publicador-subscriptor, de manera que un publicador emite elementos y un subscriptor los consume. Por lo general, la subscripción se llevará a cabo cuando un cliente invoque alguno de los métodos expuestos en el controlador mediante una llamada HTTP. Un publicador solo comienza a emitir elementos cuando haya ocurrido la subscripción. Por ejemplo, en el método para recuperar todos los workspaces del sistema, un usuario hará la petición correspondiente, provocando la suscripción que provocará que se realice una consulta sobre Mongo y el método del repositorio que gestiona la entidad workspace comience a publicar los elementos que correspondan.

En Reactor se define una interfaz denominada *Publisher*, de la que heredan dos clases, *Mono* y *Flux*. Mono se utiliza para especificar que se emitirá como máximo 0 ó 1 elementos, mientras que Flux se utiliza para definir que el flujo o stream contendrá un número indefinido de elementos en el rango 0 a N. Cada una de estas dos clases expone una gran cantidad de métodos para operar sobre los elementos del stream. En este TFM se han utilizado principalmente métodos como *map, flatMap* o *flatMapMany*. Estos se utilizan para llevar a cabo transformaciones sobre los elementos del stream. Por ejemplo, en el método de recuperación de workspaces, se realiza una operación de tipo map sobre todos los elementos para transformarlos en un *workspaceDTO* que contendrá la estructura que finalmente se enviará al cliente. Por su parte, flatMap y flatMapMany se utilizan en situaciones en las que la operación que se va a llevar a cabo a su vez devuelva otro publicador, es decir, operaciones asíncronas. Por ejemplo, las interacciones con la base de datos se realizan por lo general dentro de un flatMap, puesto que los métodos para interactuar con Mongo

siempre devuelven el resultado embebido en un Publisher. Otras operaciones que aparecen a lo largo del código son de tipo combinativo, como *concatWith* o *zipWith*. La primera se utiliza para combinar dos publicadores del mismo tipo en uno solo. Por ejemplo, en la creación del octree out-of-core, todos los datablocks que se van generando se van a añadiendo a un Flux mediante esta operación que al que se le aplicarán posteriormente en el código una operación para almacenar en la base de datos cada uno de estos elementos. Por otro lado, zipWith se utiliza por lo general para combinar dos publicadores de distinto tipo en un mismo publicador, utilizando una tupla como elemento intermedio. Por ejemplo, a la hora de insertar una nube de puntos, uno de los primeros pasos pasa por recuperar de la base de datos el dataset al que se va a asociar y almacenar los ficheros enviados por el cliente; estas dos operaciones son asíncronas y mediante zipWith se combinan en un solo publicador parametrizado a un objeto de tipo tupla, que tendrá en cada posición los valores de los publicadores combinados.

El último aspecto que se va a presentar es el de la concurrencia. Con Reactor podemos publicar los elementos de un flujo en distintos hilos virtuales, utilizando alguna de las implementaciones ya por defecto de *Schedulers*. En este proyecto la concurrencia se ha empleado de forma bastante básica, utilizando una de las implementaciones ofrecidas, *boundedElastic*, diseñada para tareas largas y bloqueantes (como es la generación de los ficheros en la implementación out-of-core del octree).

Bibliografía

Artículos científicos

Cao, C., Preda, M., Zaharia, T. (2019). *3D Point Cloud Compression: A Survey.*

Sugimoto, K., Cohen, R.A., Tian, D., Vetro, A. (2017). *Trends in efficient representation of 3D point clouds.*

Florent, P. (2019) *The Smart Point Cloud: Structuring 3D intelligent point data.*

American Society for Photogrammetry and Remote Sensing (2019), *LAS Specification Version 1.4 - R15.*

Isenburg, M. (2011), *LASzip: lossless compression of LiDAR data*.

Evans, M.R., Oliver, D., Zhou, X., Shekhar, S. (2014), *Spatial Big Data: Case Studies on Volume, Velocity and Variety,* en *Big Data: Techniques and Technologies in Geoinformatics*, capítulo 8.

Kramer, M., Senner, I. (2015), *A modular software architecture for processing of big geospatial data in the cloud.*

Boehm, J., Liu, K. (2015), NOSQL *For Storage and Retrieval of Large LiDAR Data Collections.*

Deibe, D., Amor, M., Doallo, R. (2018), *Big data storage technologies: a case study for web-based LiDAR visualization.*

Venkatraman, S., Fahd, K., Kaspi S., Venkatraman R. (2016), *SQL Versus NoSQL Movement with Big Data Analytics*.

Davoudian, A., Chen, L., Liu, M. (2018), *A survey on NoSQL Stores*.

Documentación técnica

Spring.io. (2020), *Web on Reactive Stack*. Disponible en: https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html

ProjectReactor.io (2020), *Reactor 3 Reference Guide*. Disponible en: https://projectreactor.io/docs/core/release/reference/

MongoDB (2020), *The MongoDB 4.2 Manual*. Disponible en: https://docs.mongodb.com/v4.2/