TESIS

Universidad de Jaén

Escuela de Doctorado

Estrategias para la gestión a gran escala de nubes de puntos

Juan Antonio Béjar Martos

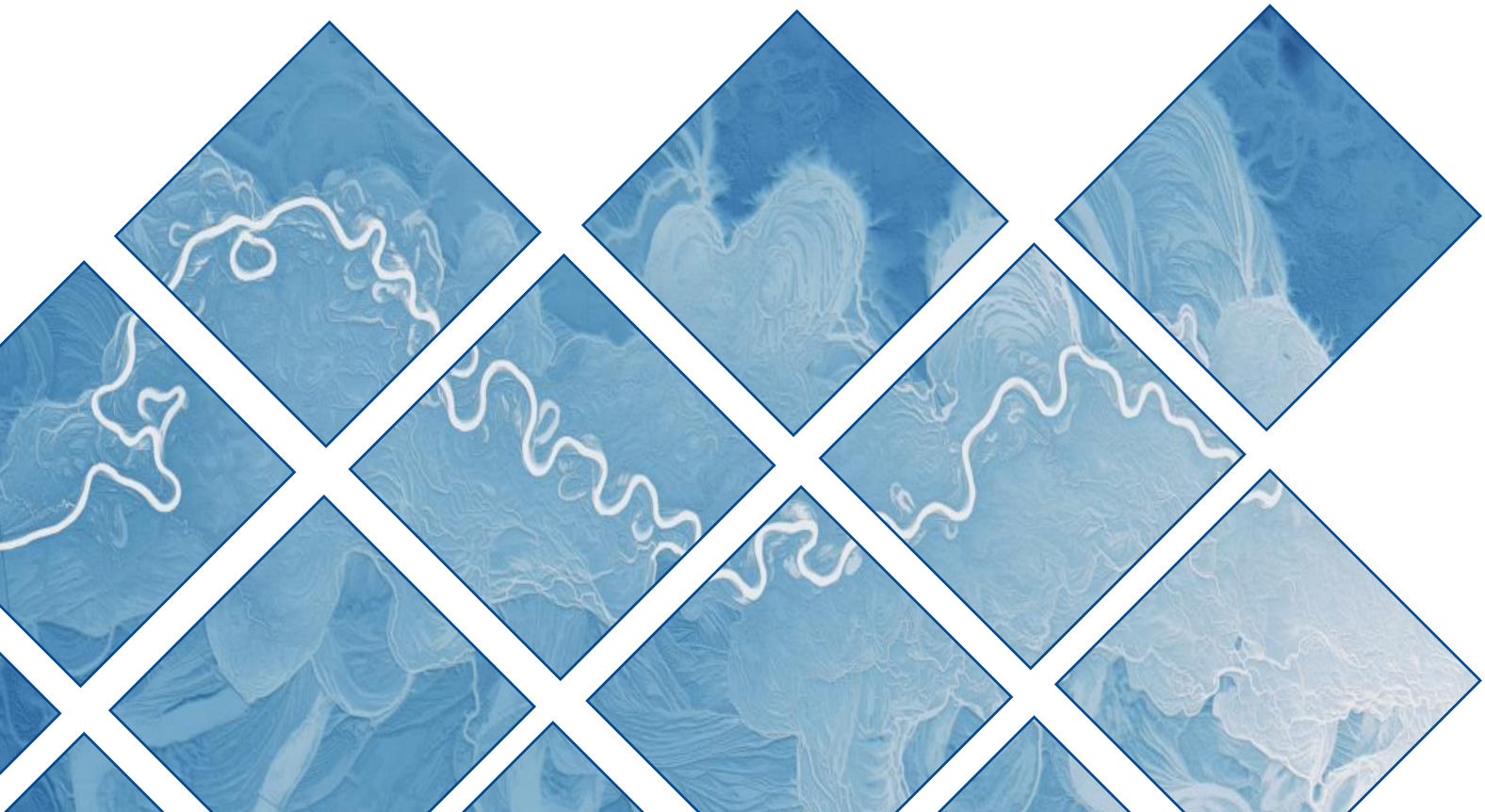
Antonio Jesús Rueda Ruíz
Carlos Javier Ogáyar Anguita

Departamento de Informática

Fecha: 25/05/2024

ESTRATEGIAS PARA LA GESTIÓN A GRAN ESCALA DE NUBES DE PUNTOS

Juan Antonio Béjar Martos



CONTENIDOS

RESUMEN

Los modelos de nubes de puntos se han convertido en una herramienta de gran importancia en multitud de dominios. Las mejoras asociadas a las tecnologías de escaneo han generado una amplia disponibilidad de modelos de alta resolución. Ante esta situación, la necesidad de definir nuevas estrategias que faciliten la organización, almacenamiento y posterior acceso a estos volúmenes masivos de datos es cada vez más apremiante. Cualquier solución que se plantee debe atender a múltiples de las necesidades de los usuarios que trabajan con esta información: poder llevar a cabo consultas en base a criterios espaciotemporales, visualización progresiva, tareas de procesamiento y análisis, edición colaborativa, etc. En esta tesis tratamos de proponer una nueva alternativa con la que resolver múltiples de estas problemáticas. Presentamos SPSLiDAR, un modelo sencillo y flexible que permite registrar modelos de nubes de puntos a escala global y a lo largo del tiempo, capaz de modelar distintas estructuras en base a las cuáles organizar la información de una nube de puntos. Incluye una interfaz que estandariza las interacciones con estos datos y facilita el acceso ubicuo a ellos. Trabajamos en la integración de estos conceptos en una solución real en forma de repositorio de nubes de puntos, analizando distintas estructuras de datos adaptadas a diferentes casos de uso y tipos de datos además del comportamiento de distintos sistemas de almacenamiento dentro de este contexto.

Palabras clave: Nubes de puntos; LiDAR; Servicios web; Sistemas de información geográfica; Estructuras de datos; Grids; Octrees; Estructuras de datos anidadas; Procesamiento de nubes de puntos; Almacenamiento de nubes de puntos; Bases de datos no-relacionales; Bases de datos SQL

ABSTRACT

Point cloud models have become a valuable source of information in multiple domains. The improvements related with scanning technologies throughout the last decade have made easier the access to detailed high-resolution point cloud models. However, this increase in the volume of information has posed challenges with regards to the storage, structuring, and access to this information. New solutions must consider the needs of the users that work with this type of data: querying by spatial and temporal criteria, progressive visualization, processing, analysis or collaborative editing are some of these use cases. In this thesis we propose SPSLiDAR, a simple and flexible model that allows to register point cloud models acquired at a global level at any time. This model allows to structure a point cloud in multiple ways, being able to adapt to different use cases. It includes an interface that standardizes the access to this information and enables ubiquitous access to it. We work on the integration of all these concepts in a real solution, building a point cloud repository. We propose multiple data structures compatible with SPSLiDAR that fit different scenarios. We also analyze different storage systems for the persistence of point clouds and its related information.

Key words: Point clouds; LiDAR; Web Services; Geographic Information Systems; Data Structures; Grids; Octrees; Nested data structures; Point cloud processing; Point cloud storage; Relational databases; Non-relational databases

AGRADECIMIENTOS

Llevar a cabo este doctorado ha sido todo un desafío. Durante mis estudios de grado nunca imaginé que acabaría acometiendo un proyecto de este tipo. Este documento es el resultado de muchos meses de sacrificio, tratando de compaginar la vida laboral fuera del marco de trabajo de la Universidad con el desarrollo de este proyecto; sin la ayuda de muchas personas esto habría sido imposible de realizar.

Primero de todo quiero darle las gracias a Antonio por haber hecho en gran parte esto posible. Primero por sus aportaciones, innumerables revisiones y comentarios que han contribuido enormemente a esta tesis y toda la línea de investigación sobre la que se sostiene. Y segundo, por haberme dado la oportunidad de formar parte del proyecto SPSLiDAR, sin la cual no habría sido posible haber llegado hasta aquí.

Del mismo modo quiero agradecer sinceramente a Carlos sus muchas contribuciones tanto a esta tesis, así como su ayuda a la hora de afrontar las dificultades encontradas durante este camino. Quiero extender estos agradecimientos a todas las personas con las que he trabajado durante estos años dentro del marco de este proyecto y han permitido que hoy sea realidad.

Por último, porque sin su apoyo esto no hubiera sido posible. Gracias a mamá, papá y María.

INTRODUCCIÓN

En este capítulo se realiza una breve introducción al trabajo realizado que sirve para presentar las motivaciones que originaron la investigación, así como los objetivos planteados. Además, se incluye un breve resumen del contenido de cada capítulo, para proporcionar un esquema de la estructura general de esta tesis doctoral.

Motivación

La presente tesis se ha llevado a cabo dentro del ámbito del proyecto de investigación "Estrategias para el procesamiento y segmentación a gran escala de grandes nubes de puntos. Aplicaciones (SPS-LIDAR) - RTI2018-099638-B-I00" financiado por el Ministerio de Ciencia, Innovación y Universidades en su convocatoria de Proyectos I+D+i - Retos Investigación de 2018. Las líneas de investigación desarrolladas en la tesis han estado motivadas por los objetivos fijados en dicho proyecto.

La tecnología LiDAR permite obtener representaciones de gran precisión sobre todo tipo de objetos físicos en forma de modelos de nube de puntos. Los sensores que implementan esta tecnología han evolucionado de manera notable en los últimos años, incrementando la calidad de la información obtenida y reduciendo su coste, lo que ha facilitado el acceso a estos dispositivos. Si hablamos de LiDAR de medio alcance, algunos de los sensores más punteros en la actualidad son capaces de generar más de 2 millones de puntos por segundo, con rangos de precisión de 5 mm a 40 m de distancia. Además, la tecnología LiDAR se está extendiendo a dispositivos de consumo como los propios teléfonos móviles, como es el caso de los iPhone de la serie Pro a partir del modelo 12.

Debido a estas mejoras en el ámbito técnico, la popularidad del LiDAR es cada vez mayor. Múltiples disciplinas ven en esta tecnología una herramienta con la que ofrecer nuevos servicios y mejorar sus flujos de trabajo, generando volúmenes de información crecientes en forma de modelos de nube de puntos.

Estos modelos de nube de puntos se definen como una colección de puntos identificados por sus coordenadas (x, y, z) en base a un sistema de coordenadas preestablecido. Normalmente, se incluyen otra serie de atributos adicionales que pueden representar aspectos como, por ejemplo, el color o la clasificación del punto en una serie de categorías. Para guardar esta información existen formatos de fichero específicos, desde simples archivos de texto como formatos más específicos y optimizados como es el caso del estándar LAS y su versión comprimida LAZ.

El uso masivo de modelos de nube de puntos, con densidades cada vez mayores, varias versiones tomadas en distintos instantes temporales, y múltiples atributos adquiridos por punto plantea importantes desafíos a la hora de organizar la información, tanto por el volumen como por el número de ficheros implicados. Por lo general, los ficheros se estructuran en jerarquías de carpetas, utilizando los nombres de los diferentes niveles del sistema de archivos a modo de identificadores. Cuando el número de archivos es reducido, esta forma de trabajar puede ser asumible, pero cuando la volumetría de los datos se incrementa y aparecen requisitos para recuperar información de estos en base a distintos tipos de criterios (espaciales, temporales, etc.),

los flujos de trabajo dejan de ser eficientes y es necesaria una política consistente y rigurosa para mantener la información organizada correctamente ante cualquier tipo de transformación.

Otro tanto ocurre cuando es necesario compartir esta información a nivel interno en una organización o de forma pública en Internet. Habitualmente se utilizan carpetas compartidas a nivel de intranet, servidores FTP o colecciones de enlaces en páginas web para descarga, de manera directa (Hackel et al., 2017) o mediante algún mecanismo de búsqueda básico por coordenadas o selección en un mapa (PNOA, 2024) . No existen mecanismos directos para la descarga parcial del modelo o la obtención de distintos niveles de detalle.

La naturaleza y complejidad de estos problemas son propios de big data y se ven afectados por los cinco principios fundamentales a considerar dentro de esta área (las 5 Vs): volumen, velocidad, valor, veracidad y variedad. Las particularidades de la información asociada a las nubes de puntos y sus necesidades específicas en términos de almacenamiento y procesamiento han llevado a algunos autores a hablar de Big Data Geoespacial (Evans et al., 2014; Poux, 2019).

El propósito de esta tesis es plantear un modelo de repositorio centralizado que mejore el almacenamiento, organización y acceso a grandes volúmenes de información en forma de modelos de nube de puntos. Este objetivo está alineado con las ideas planteadas por el Open Geospatial Consortium (OGC), dentro del documento de trabajo “Testbed-14: Point cloud data handling engineering report” (Boehler et al., 2018) , que recomienda el uso de repositorios accesibles a través de servicios web para facilitar la diseminación y explotación de los modelos de nube de puntos.

Por tanto, el trabajo realizado durante esta tesis se ha centrado en el diseño y la implementación de una repositorio denominado SPSLiDAR (Rueda-Ruiz et al., 2022) que se adhiere a múltiples de los requisitos citados en el informe de la OGC, planteando un modelo flexible capaz de gestionar información de distintas fuentes de datos bajo múltiples formatos, la capacidad de llevar a cabo consultas de tipo espaciotemporales y ofreciendo distintos niveles de detalle.

Puesto que nuestro ámbito de trabajo se centra en la investigación de soluciones orientadas a servicios, es fundamental incluir el diseño de una API que permita a aplicaciones clientes (escritorio, web, móviles) interactuar con la solución que se proponga. Esta API debe tener en cuenta la naturaleza del modelo y al mismo tiempo ser lo suficientemente flexible para adaptarse a diferentes cambios y estrategias que se lleven a cabo a nivel interno sin afectar la manera en la que los consumidores interactúen con ella.

Estas estrategias pueden hacer referencia a cómo se organizan los datos de los modelos de nube de puntos. Existen múltiples estructuras de datos abordadas en la literatura que proponen distintas formas de organizar la información de una nube: octrees, quadtrees, grids, estructuras anidadas que combinen algunas de las ya mencionadas, etc. (Ogayar-Anguita et al., 2023a; Scheiblauer & Wimmer, 2011; Schütz et al., 2020). El uso de una u otra dependerá de cada caso de uso, pero es fundamental que la solución que se plantee pueda dar soporte a distintas estructuras mediante una API agnóstica. Una de las líneas de trabajo que se propone pasará por evaluar múltiples de estas estructuras de datos para entender cómo se adecúan a la solución final y analizar de forma comparativa su rendimiento ante distintas casuísticas.

Del mismo modo, también se considera cómo gestionar el almacenamiento de esta información de forma eficiente. Como ya se ha mencionado previamente, el tratamiento de este tipo de información se enmarca dentro del big data, y por ello requiere de soluciones adaptadas a su naturaleza. Cada vez se apuesta más por alternativas que aportan mayor flexibilidad que los tradicionales ficheros LAS/LAZ o en formatos propietarios, como el uso de bases de datos NoSQL. A priori este tipo de bases de datos presenta ventajas como son la posibilidad de organizar la información espacial de una manera eficiente y la capacidad para escalar mejor conforme aumenta el volumen de datos a gestionar. El diseño de soluciones adaptadas a este tipo de bases de datos y la ejecución de un análisis comparativo entre distintos tipos de implementaciones es también uno de los puntos que aspiramos a tratar (Béjar-Martos et al., 2022; Boehm, 2014; Deibe et al., 2018; Khan et al., 2023).

A raíz de las necesidades encontradas y los avances ya realizados, esta tesis pretende plasmar los resultados obtenidos, avanzar en el desarrollo de esta investigación y plantear líneas futuras de trabajo. Puesto que el trabajo se realizó dentro del citado proyecto del Programa Estatal de I+D+i Orientado a los Retos de la Sociedad, consideramos que se adecúa de forma satisfactoria a las estrategias de investigación nacionales.

Objetivos

Los objetivos principales planteados en el origen de esta tesis son los siguientes:

1. Definir un esquema conceptual para la estructuración de modelos de nube de puntos a gran escala. Diseñar las entidades y la arquitectura de un sistema capaz de gestionar información de nubes de puntos georreferenciadas.

2. Estudio de una solución que traslade el esquema definido a un sistema que permita la gestión de modelos de nube de puntos a través de servicios web.

A nivel específico los objetivos que se plantean son:

A. Estudio bibliográfico para entender las necesidades y adecuar los objetivos generales al estado del arte actual, incluyendo los estándares, recomendaciones y documentos de trabajo del OGC.
B. Diseño de un modelo de clases que defina los elementos necesarios para dar soporte a los principales requisitos asociados a la gestión de modelos de nube de puntos a gran escala.
C. Diseño de una API REST que declare los servicios que debe ofrecer un repositorio orientado a modelo de nubes de puntos.
D. Estudio tecnológico e implementación del repositorio definido en el objetivo principal 2, dando soporte a los distintos servicios declarados en la API REST.
E. Diseño e implementación de distintas estructuras de datos para la organización de nubes de puntos.

F. Diseño e implementación del esquema de datos adecuado para distintos sistemas de gestión de bases de datos SQL/NoSQL.
G. Experimentación y análisis comparativo de rendimiento.

Estructura de la memoria

A continuación se presenta un breve resumen de los contenidos de cada capítulo.

Capítulo 1 - Situación y últimos avances

En este capítulo se presenta un resumen de los temas principales tratados en esta tesis, junto con una discusión sobre los trabajos más relevantes de la literatura científica relacionados con dichos temas.

Capítulo 2 - Definición del modelo: especificación SPSLiDAR.

En este capítulo se presenta el modelo conceptual que conforma la arquitectura para el almacenamiento y consulta de nubes de puntos. Este modelo teórico también se implementa como prototipo para su validación y se presenta en el capítulo siguiente.

Capítulo 3 - Implementación de SPSLiDAR en un repositorio para nubes de puntos

En este capítulo se presentan los resultados de la implementación del modelo teórico mediante una solución backend para su utilización vía web. También se presentan los resultados de las experimentaciones, que validan la utilidad del modelo desarrollado.

Capítulo 4 - Sistemas de almacenamiento y escalabilidad en el contexto de SPSLiDAR.

Este capítulo trata sobre la problemática de trabajar a bajo nivel con distintos sistemas de gestión de bases de datos (SGBD) para dar soporte a la implementación del modelo en lo referente a la persistencia. No es un aspecto trivial elegir un SGBD que maximice las prestaciones del sistema, ya que hay muchas variables que imposibilitan resolverlo a nivel teórico. Aquí se presentan los resultados ya publicados sobre una comparativa de rendimiento entre algunos de los SGBD más conocidos actualmente.

Capítulo 5 - SPSLiDAR en el contexto de los servicios web y los estándares de la OGC

Este capítulo presenta los estándares del Open Geospatial Consortium existentes para el acceso a la información geográfica, así como sus últimas propuestas. Además, se ofrece una comparativa con las novedades presentadas en esta tesis, ya que comparten objetivos y filosofía.

Capítulo 6 - Conclusiones y trabajos futuros.

En este capítulo se presentan las conclusiones de la investigación realizada, así como una exposición de los temas que se pretenden desarrollar en el futuro.

Capítulo 1 - SITUACIÓN Y ÚLTIMOS AVANCES

1.1. Introducción

En los últimos años, el uso de escáneres 3D de medio y largo alcance se ha generalizado en numerosos ámbitos. En pocos minutos, estos dispositivos permiten obtener un modelo basado en puntos de gran precisión de un entorno exterior o interior. En el entorno exterior, la fotogrametría desde plataformas convencionales o más recientemente con el uso de sistemas aéreos no tripulados (UAS) o remotamente pilotados (RPAS), conocidos comúnmente como drones, han posibilitado el escaneado de bajo coste de múltiples estructuras, así como de terrenos completos. En el entorno interior, el uso del escaneado basado en imagen estéreo con dispositivos móviles permite obtener información razonablemente detallada del entorno en muy poco tiempo. Las disciplinas que más se han beneficiado de esta tecnología son la geomática (topografía y cartografía), arquitectura, ingeniería civil, planificación urbana, arqueología o gestión medioambiental, entre otras. La disponibilidad de información 3D precisa sobre el entorno (natural o urbano) facilita tareas como el control ambiental y de la masa forestal, control de la erosión del suelo, clasificación del fondo marino, monitorización de costas, lagos y ríos, mediciones de superficie a efectos catastrales y de lindes, gestión de mapas de ruido para el control de la contaminación acústica, planificación del riesgo de desastres, gestión de emergencias, modelado urbano en entornos complejos, monitorización y conservación de carreteras y vías férreas, control de redes de distribución de recursos, etc. La información sobre edificios, monumentos y otras estructuras de menor envergadura (que no complejidad) permite mejorar tareas como el control de edificaciones y reformas, o la catalogación, reconstrucción, rehabilitación, conservación y difusión del patrimonio histórico y cultural (Figura 1.1).

Figura 1.1. Modelo de una iglesia obtenido a partir de LiDAR terrestre (Kadobayashi et al., 2004)

A bajo nivel, la información que se obtiene mediante el escaneado 3D es un conjunto de puntos definidos por sus coordenadas en el espacio (en muchos casos georreferenciadas) junto a una serie de atributos adicionales (color, intensidad, etc.), que en conjunto se denomina modelo de nube de puntos o simplemente nube de puntos (point cloud). Una nube de puntos puede describir la geometría y aspecto de los elementos capturados de una forma bastante precisa, siempre que la resolución del escáner sea suficiente y el proceso de escaneado se haya realizado adecuadamente. Las nubes de puntos pueden obtenerse desde múltiples orígenes, como conversiones de representación desde mallas de polígonos, muestreos de distintos tipos, generación procedural, etc. Sin embargo, el tipo de conjunto de datos que más nos interesa es el

Figura1.2. Escena escaneada en París usando un Trimble TX8 para el proyecto Terra Mobilita. ©Trimble.

procedente de captación de datos del entorno real (Figura 1.2), ya sea mediante escaneado, fotogrametría o Structure from Motion (SfM).

El resultado de un proceso de captación de puntos del entorno real raramente puede utilizarse directamente, siendo necesario un posprocesamiento con programas como LAStools, CloudCompare, Trimble RealWorks, o la suite de Terrasolid (TerraScan, TerraModeler, TerraStreet, etc.), entre otros. Varias de las operaciones más comunes que se realizan son el registrado en el mismo sistema de coordenadas de referencia de diferentes nubes de puntos, el filtrado y eliminación de elementos no deseados, la realización de mediciones de interés, la obtención de secciones transversales, la generación de superficies de interpolación de los puntos o la clasificación semántica de los mismos. Algunas de estas tareas plantean bastantes dificultades, debido a las características propias de las nubes de puntos procedentes de escaneado LiDAR o en su caso las derivadas de procesos fotogramétricos. En primer lugar, a nivel geométrico no existen relaciones de conectividad entre puntos, lo que dificulta el diseño de algoritmos eficientes de edición, filtrado y sobre todo generación de superficies. En segundo lugar, el gran tamaño que tienen típicamente los modelos de nube de puntos penaliza todas las operaciones, con especial impacto en la visualización interactiva del modelo. A nivel de investigación estos problemas han sido abordados con mayor o menor éxito, especialmente la generación de superficies, la edición básica y la visualización. En general, aunque existen muchas propuestas para abordar parte de estos problemas, su aplicación, sin embargo, es muy concreta y resultan difíciles de aplicar en las situaciones reales citadas. Además, por motivos de diseño, estructuras de datos e implementación, son muy difíciles de compatibilizar en el mismo software.

Esta tesis continúa la labor de un equipo de investigadores que ha trabajado en la mayoría de los problemas indicados anteriormente. En el proyecto Scancyr (Rueda-Ruiz et al., 2009), ejecutado en el marco de un contrato con las empresas Sacyr y Aplitop, se desarrolló un software que facilitaba el trabajo relacionado con el tratamiento de los datos escaneados en una oficina

técnica de una empresa de obra civil. En un escenario de tratamiento de datos manual, muy particionado y poco eficiente, se consiguió obtener una herramienta de edición integrada con prestaciones gráficas de última generación, que permitía realizar en tiempo real tareas como la segmentación de puntos, la detección automática de elementos normalmente no deseados (cableado, árboles, andamiaje, personal de obra) (Gonzalez et al., 2012), el filtrado de puntos o el cálculo de secciones transversales. Como punto negativo, no abordaba el problema del almacenamiento y gestión de grandes colecciones de nubes de puntos ni el tratamiento out-of-core de los datos. Además, el dataset debía ser muestreado parcialmente en caso de falta de memoria, y no había control de versiones o comparación de nubes de puntos tomadas en distintos momentos temporales. El proyecto Las-Roads, desarrollado posteriormente junto a la constructora Sando y otras empresas, supuso una evolución del software anterior, añadiendo la extracción automática de información de interés empleada en el diseño y control de infraestructuras relacionadas con la obra civil, como, por ejemplo, la detección automática de marcas viales. No obstante, esta aplicación seguía teniendo las mismas limitaciones en cuanto a la gestión masiva de datos.

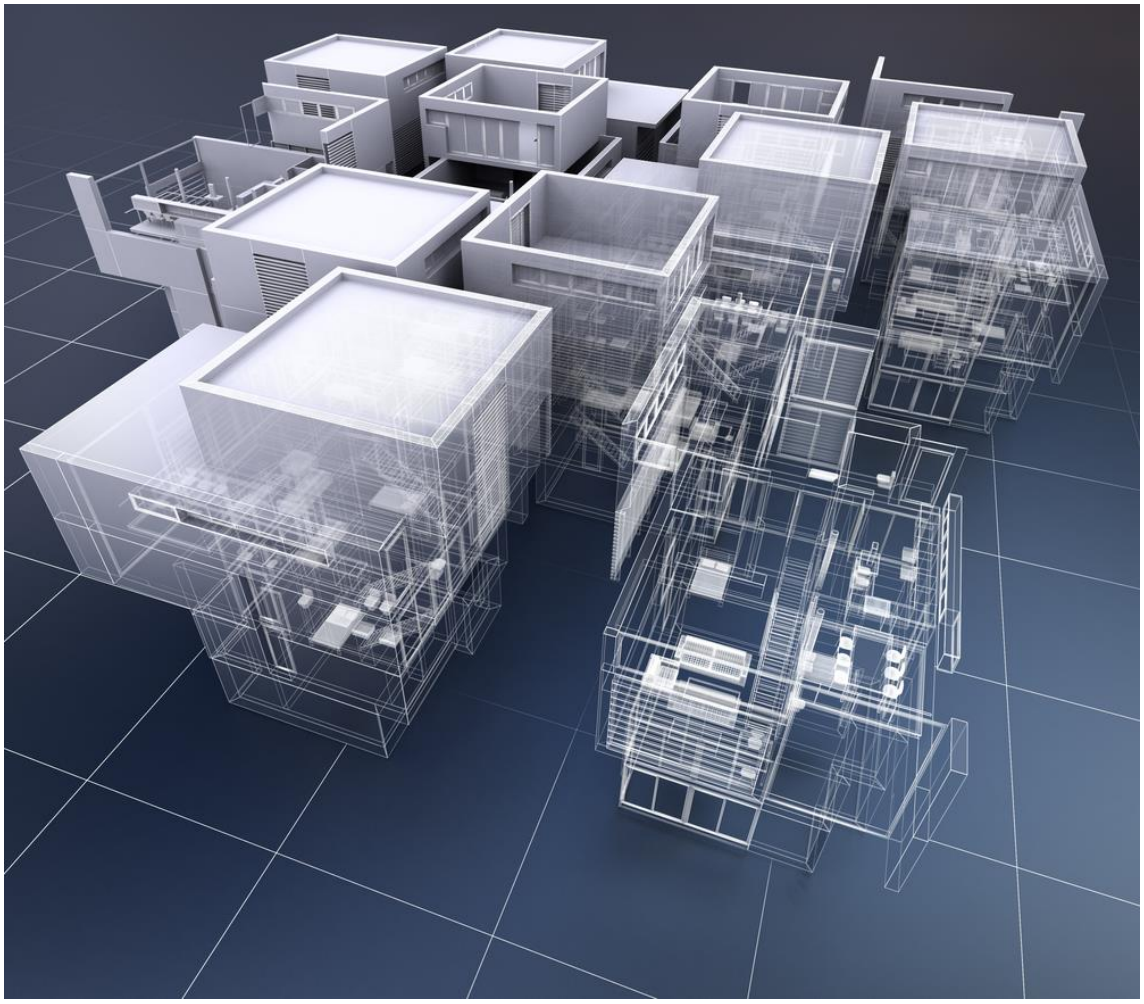


Figura 1.3. Mediante el proceso Scan-to-BIM se pueden crear modelos 3D de una infraestructura o inmueble a partir de nubes de puntos. © Shutterstock.

Como se ha comentado anteriormente, el principal problema que surge en el procesamiento de las nubes de puntos son los grandes volúmenes de datos implicados, problema que se acentúa con la continua aparición de hardware de captura más preciso y potente. Además, la tendencia futura es a utilizar el escaneado de forma más generalizada y masiva, utilizando una amplia variedad de dispositivos que incluyen desde equipos LiDAR profesionales de altísima precisión hasta sensores integrados en dispositivos de bajo coste como los teléfonos móviles inteligentes, y pasando desde la escala de un objeto, edificio o infraestructura concreta a zonas amplias del territorio que cubren muchos kilómetros. Existen ejemplos muy ilustrativos en ámbitos como la arqueología y preservación del patrimonio. Según la UNESCO, 48 lugares arqueológicos que son patrimonio de la humanidad están en peligro debido a guerras o catástrofes naturales, especialmente en el medio oriente. Esto ha motivado una carrera para preservar este patrimonio, al menos de manera digital, utilizando el escaneado LiDAR a gran escala y la fotografía de alta definición como herramientas. En lugares como Petra esto implica una tarea ingente (*Cyber-archaeology, big data and the race to save threatened cultural heritage sites ð phys.org*, s. f.). Otro ámbito en el que se está produciendo esta transición es el de la obra civil, gracias a la implantación del Building Information Management (BIM), que usa el escaneado LiDAR para capturar el modelo fiel (As-Built) de una infraestructura durante el progreso de la obra o a su finalización (Figura 1.3). A pesar de que aún no es de carácter obligatorio en todo el territorio español, el BIM está entre las recomendaciones de las administraciones tanto nacionales como regionales, y desde 2019 es un requisito para licitaciones públicas de edificación e infraestructuras. En los países nórdicos como Dinamarca, Noruega y Finlandia, además del Reino Unido, esto ya es una realidad. Lo mismo ocurre en las aplicaciones relacionadas con la gestión y planificación urbana y medioambiental, donde se precisa trabajar con nubes de puntos de gran tamaño. El volumen de información generado por este tipo de escaneados a gran escala impone un desafío sin precedentes a los medios convencionales actuales, tanto en lo que se refiere a hardware como software. Esto obliga al uso de estrategias que intentan paliar el problema, trabajando solo con una parte del dataset o con un pequeño muestreo del mismo.

En general, se requieren soluciones integrales y eficientes para procesar los datos LiDAR. Esta tesis es parte de un proyecto donde se proponen soluciones para algunos de los problemas que quedan por resolver, en concreto, se plantea el diseño e implementación de un sistema para el almacenamiento y gestión de volúmenes masivos de datos procedentes de escaneado LiDAR y otras fuentes, así como su gestión y procesamiento en un entorno virtual común basado en la nube. Otra necesidad planteada por los usuarios que habitualmente trabajan con este tipo de datos es un control de versiones que facilite la edición del modelo, especialmente cuando varios técnicos trabajan simultáneamente con el mismo, y por otro lado integrar la evolución temporal del modelo, aspecto de especial interés en áreas como la construcción, el urbanismo, o el seguimiento de procesos naturales o eventos sociales. En la literatura existen trabajos que resuelven algunos de estos problemas, total o parcialmente; sin embargo, en muchos casos no se pueden aplicar o no se adaptan bien al tratamiento de nubes de puntos. Además, como ya se ha expuesto anteriormente, en cuanto a diseño, estructuras de datos e implementación, presentan incompatibilidades entre ellas, lo que las hace muy difíciles de integrar en una sola herramienta. Lo que sí queda claro es que la gran cantidad de usos de los datos LiDAR requieren de un sistema que soporte datasets masivos con un acceso lo más eficiente posible, y con esta tesis se busca ofrecer soluciones de base a este problema.

1.2. Modelos de nube de puntos

Como se ha mencionado anteriormente, un modelo de nube de puntos es un conjunto discreto de puntos en el espacio que puede representar una forma u objeto 3D. La posición de cada punto se define mediante sus coordenadas en un sistema de referencia preestablecido, que puede ser cartesiano o esférico. En relación con las tecnologías de adquisición de los datos queremos destacar que, a priori, no son un objetivo en esta tesis, si bien es evidente que se consideran los diferentes métodos de captura (LiDAR o fotogramétricos) y su resolución, que está condicionada por la tecnología y el tipo del sensor LiDAR, ya sea aéreo (ALS) o terrestre (TLS). Los formatos de almacenamiento de las nubes de puntos como LAS/LAZ (ASPRS, 2019; Isenburg, 2013), SPD (Bunting et al., 2013) o HDF5 (OGC, 2019; The HDF Group, 2024) también tienen un interés secundario para la investigación. Los objetivos planteados en esta investigación sobre estrategias de organización espacial de los datos, almacenamiento y acceso van más allá de lo que puede aportar un formato u otro de almacenamiento. A ello debemos añadir que ninguno de los formatos existentes soporta la multirresolución intrínseca que precisamos en la organización de la información; así, el formato SPD carece de soporte de multirresolución o almacenamiento distribuido, soportando únicamente una indexación espacial básica mediante una rejilla, insuficiente para muchas aplicaciones que requieren del manejo de grandes modelos. De modo similar, para el estándar LAS existe la posibilidad de utilizar ficheros basados en el formato LAX para indexar información.

El modelado basado en puntos es una alternativa a los sistemas clásicos de modelado para la representación de superficies, tanto para el tratamiento como para la visualización de modelos

Figura 1.4. Ejemplo de modelado basado en puntos. Autodesk.

complejos, lo que ha sido objeto de intensa investigación en la última década. Esto ha dado lugar a la creación de un área dentro de la informática gráfica denominada point-based graphics (Akenine-Möller et al., 2008; Gross & H. Pfister (eds.), 2007). No obstante, la investigación en esta área se ha centrado principalmente en aspectos relacionados con la visualización realista, y siempre con modelos de complejidad moderada (Günther et al., 2013; Wand et al., 2008).

En los modelos basados en puntos no es necesario mantener información topológica global, lo que facilita el tratamiento de objetos cuya forma cambia dinámicamente. Una representación basada en puntos puede ser considerada como un muestreo de una superficie continua, que incluye opcionalmente vectores normales, colores y otras propiedades asociadas. Una de las ventajas de esta representación es que permite obtener propiedades asociadas a un punto cualquiera a partir de los puntos circundantes pertenecientes al modelo. El principal inconveniente de utilizar modelos basados en puntos es la falta a priori de conectividad, lo que hace más complicadas las consultas de tipo geométrico y topológico (Figura 1.4).

Tal y como se ha mencionado anteriormente, las nubes de puntos suelen elaborarse a partir de datos obtenidos con escáneres 3D (ver Figuras 1.2 y 1.4) o utilizando métodos de reconstrucción basados en imágenes (Barrile et al., 2019). A la información básica de posición, color o vector normal, se suelen añadir datos como la intensidad, el NIR (Near Infra-Red), la geolocalización, etc. (Chen et al., 2014). La precisión de la geometría que puede obtenerse dependerá de la resolución del escáner y de la corrección del proceso de escaneado. En cualquier caso, los puntos obtenidos contienen cierta cantidad de ruido debido a estos factores, que puede reducirse mediante técnicas de filtrado. Además, los puntos pueden disponerse en el espacio con o sin un patrón conocido, es decir, estar o no estar organizados. Los datos organizados (organized cloud dataset) se presentan siguiendo una estructura tipo matriz, y por tanto divididos en filas y columnas. Esto es típico cuando se trabaja con cámaras estéreo o de tiempo de vuelo, además de los escáneres basados en láser. La gran ventaja de este subesquema es que las operaciones de cálculo de los vecinos más cercanos de un punto son más rápidas, lo que impacta de forma directa en el rendimiento de la mayoría de los algoritmos que operan sobre nubes de puntos. Además, este conocimiento puede aprovecharse para almacenar los datos escaneados en una imagen ráster donde cada celda corresponde a un retorno del láser. Estas imágenes ráster se pueden guardar y consultar de manera eficiente, simplificando así el problema de almacenamiento de la nube de puntos. Los fabricantes de escáneres terrestres han introducido sus propios formatos para almacenar puntos de esta manera.

Figura1.5. Ejemplo de nube de puntos masiva MGGP Aero.

A pesar de lo anterior, se debe suponer que el caso general con el que hay que trabajar es el de nubes de puntos desorganizadas, en las cuales no existe relación de vecindad entre los elementos (H. Huang et al., 2009). Por ejemplo, en el caso de un modelo obtenido tras el registrado de varias tomas de escaneado 3D independientes no es posible utilizar la correlación espacial entre puntos vecinos en el resultado final, aunque los datos de cada una de las tomas se almacenen de forma consecutiva. Como consecuencia, ya no es posible representar las nubes de puntos como simples imágenes ráster, y en su lugar deben almacenarse las coordenadas 3D reales para cada punto. Es este tipo de nubes de puntos el que plantea los mayores desafíos por el volumen de datos que puede implicar, particularmente con modelos que en muchas aplicaciones actuales rozan o superan los 1000 millones de puntos (Figura 1.5).

El procesamiento de grandes modelos de nube de puntos implica una serie de problemas que no están resueltos, o cuya resolución con los métodos actuales no es óptima. Los datos escaneados deberán ser visualizados y examinados desde distintos puntos de vista en tiempo real. Su gran volumen hace muy difícil el obtener una respuesta interactiva en las aplicaciones que necesitan trabajar con este tipo de datos. Además, dependiendo de la aplicación, pueden realizarse distintas operaciones sobre el modelo de puntos, como mediciones de interés (alturas, distancias, áreas, etc.), segmentación y clasificación de partes de los datos (Grilli et al., 2017), así como la obtención de secciones transversales en cualquier zona.

Otra de las operaciones comunes al trabajar con nubes de puntos es la conversión a otros esquemas de representación, con el objeto de aprovechar sus ventajas de cara a la ejecución de diversos algoritmos o la realización de procesos relacionados con el almacenamiento y la visualización. La conversión a mallas de triángulos es una de las opciones más frecuentes, lo que permite más capacidades como la visualización más realista de los datos, la edición de los mismos en un programa CAD, o la realización de simulaciones de distinta naturaleza (resistencia estructural, erosión, etc.). Este problema de gran interés ha recibido mucha atención por parte de la comunidad científica (H. Huang et al., 2009; Turk & Levoy, 1994; X. Wang et al., 2013). A pesar de la cantidad de trabajos existentes relacionados con este tema, pocos abordan el problema

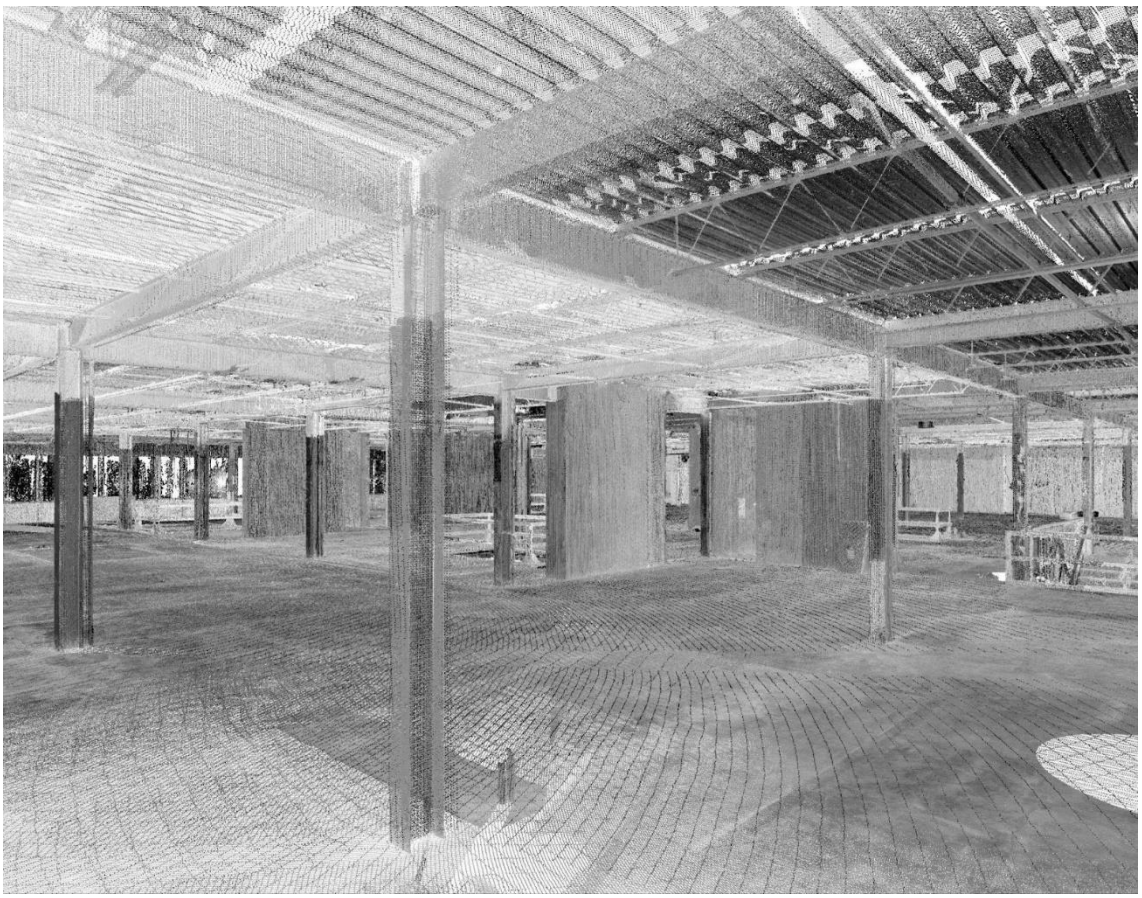


Figura 1.6. Ejemplo de uso de escaneados para evaluar el progreso de construcción. © 3DVDT.

de la reconstrucción a partir de modelos de partida de calidad media-baja, que son comunes en las sesiones de escaneado de exterior por la presencia de multitud de factores ambientales.

Como puede verse, hay una gran cantidad de temas y aplicaciones abiertos a la investigación en el tratamiento de nubes de puntos y su conversión a otros esquemas de representación. Teniendo en cuenta todo lo anterior, especialmente el volumen de los datos a tratar, sería necesario disponer de una gestión óptima de datasets masivos, tanto en aspectos de edición como de almacenamiento y transmisión por red. Relacionado con lo anterior, sería importante contar con estructuras de datos compatibles con GIS y BIM (ver Figura 1.6) que permitieran organizar los datos espacialmente, por capas, agrupamientos con una semántica determinada, incorporar un histórico de evolución del entorno, además de opciones como accesos restringidos o compresión de datos para una visualización eficiente, tanto en compresión sin pérdida (Schnabel & Klein, 2006) como con pérdida (de Queiroz & Chou, 2016; Morell et al., 2014), siendo esta última la más habitual. El concepto de computación en la nube en este sentido cobra gran importancia, ya que probablemente sea necesaria el uso de tecnologías escalables compatibles con la nube (microservicios, bases de datos NoSQL, soluciones serverless) para proporcionar elasticidad en función de las necesidades del sistema derivadas del gran volumen de información a procesar.

1.3. Organización espacial de nubes de puntos

La estructura de datos espaciales utilizada para la organización de una nube de puntos tiene una influencia determinante en todos los procesos que se pueden llevar a cabo. La mayoría de las propuestas de investigación definen una estructura de datos que está fuertemente ligada a una aplicación específica, dejando de lado otros usos importantes. Esto es muy común en sistemas orientados al renderizado (Deibe et al., 2019; Goswami et al., 2012; Preiner et al., 2012; Richter et al., 2015; Schütz, 2016; Schutz & Wimmer, 2019), que en algunos casos no son adecuados para otros fines como el almacenamiento óptimo o el análisis de datos. Por otro lado, existen varios trabajos que se centran en el almacenamiento masivo en memoria secundaria, en la red o en la nube (Baert et al., 2013; Deibe et al., 2019; Richter et al., 2015; Scheiblauer & Wimmer, 2011; Schütz et al., 2020).

La mayoría de los autores proponen soluciones para la indexación espacial basadas principalmente en grids y quadtrees para datos planares (Boehm et al., 2016; Deibe et al., 2019; Hongchao & Wang, 2011), u octrees para datos volumétricos (Elseberg et al., 2013; L. Huang et al., 2021; B. Lu et al., 2019; Scheiblauer & Wimmer, 2011; Schön et al., 2013; Schütz et al., 2020; Tian et al., 2019). Estas soluciones de indexación espacial son bastante básicas, teniendo en cuenta la creciente complejidad de los conjuntos de datos y los procesos a realizar sobre ellos. La estructura de datos más utilizada para el almacenamiento y renderizado de nubes de puntos es el octree. Otros enfoques se basan en el k-d tree, que optimiza la partición del espacio (Goswami et al., 2012), una estructura de vóxel dispersa (Baert et al., 2013; Kämpe et al., 2013; Poux, 2019) o una cuadrícula 2D (Hongchao & Wang, 2011). Existen otras estructuras de datos que funcionan mejor con objetos completos, como el r-tree. Normalmente se utilizan con mallas poligonales, aunque hay trabajos que utilizan con éxito algunas variantes con nubes de puntos (Gong et al., 2012), así como otras que podrían adaptarse para este fin (Y. Wang et al., 2020). La Figura 1.7 muestra ejemplos de algunas estructuras espaciales básicas.

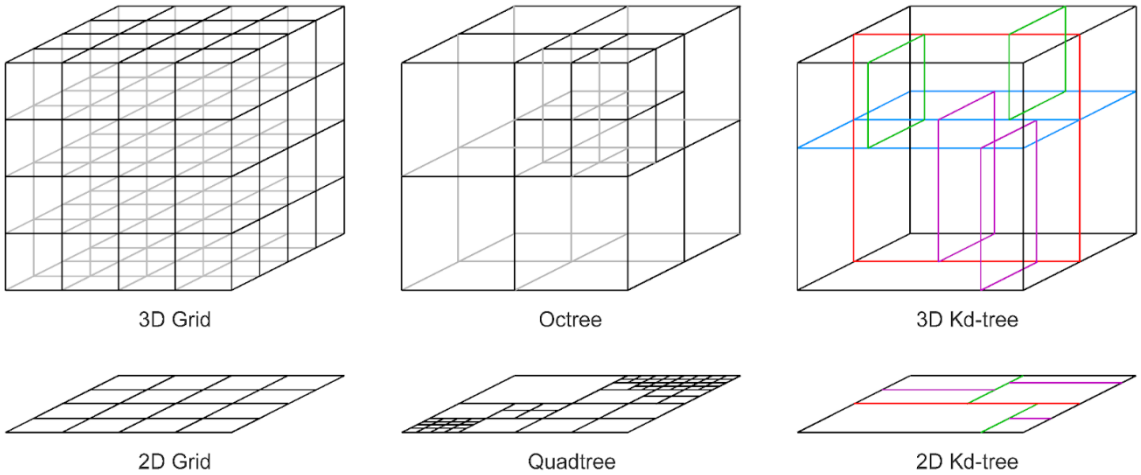


Figura1.7. Ejemplos de estructuras de datos comunes.

Algunas de las propuestas citadas incluyen el uso de una estructura espacial anidada o híbrida, orientada principalmente al renderizado (Deibe et al., 2019; Schütz, 2016; J. Yang & Huang, 2014) y a la optimización de las operaciones de búsqueda (B. Lu et al., 2019; Scheiblauer

& Wimmer, 2011). La mayoría de estos métodos tienden a mantener la estructura de datos espaciales lo más simple posible, al tiempo que permiten trabajar con niveles de detalle (level of detail o LOD) (Deibe et al., 2019; Schütz, 2016). El principal inconveniente es que la indexación espacial no se adapta bien a todas las distribuciones espaciales de puntos posibles. El caso más relevante es la mala adaptación del octree a la distribución espacial de puntos procedentes de captura LiDAR de grandes extensiones del territorio. En estos casos, las estructuras planares del grid y el quadtree suelen ser más apropiadas (Boehm et al., 2016; Deibe et al., 2019; Hongchao & Wang, 2011). Sin embargo, si un dataset tiene múltiples patrones de distribución de puntos, ninguna estructura podrá ajustarse de manera óptima en todas las zonas, es decir, con el menor número de nodos posible.

1.4. Almacenamiento de nubes de puntos

Algunos de los artículos mencionados anteriormente solo abordan la indexación de nubes de puntos en la memoria, mientras que otros también lo hacen en el nivel de almacenamiento out-of-core. Los enfoques más simples utilizan el almacenamiento directo en la memoria secundaria usando archivos locales (Scheiblauer & Wimmer, 2011; Schütz et al., 2020), mientras que otros dependen del uso de sistemas de archivos distribuidos (Deibe et al., 2018; Krämer & Senner, 2015), generalmente indexados a través de una estructura espacial, que se implementa en un archivo de índice maestro o almacenado en una base de datos. Existen varios formatos de archivo conocidos para almacenar nubes de puntos y datos LiDAR. Las opciones más comunes son los formatos ASCII no estándar, el formato LAS de la Sociedad Americana de Fotogrametría y Teledetección (ASPRS) (ASPRS, 2019), su variante comprimida LAZ (Isenburg, 2013), SPD (Bunting et al., 2013), PCD (Rusu & Cousins, 2011), HDF5 (OGC, 2019; The HDF Group, 2024) y otros formatos generales de archivos de datos 3D como OBJ (Wavefront Technologies, 1990) o STL (3D Systems, 1988). Desde un punto de vista práctico, los formatos más interesantes son aquellos que permiten la compresión de datos (Cao et al., 2019; Isenburg, 2013; Sugimoto et al., 2017). Cualquiera de estas opciones es válida para almacenar grupos de puntos contenidos en los nodos de una estructura espacial utilizada en la memoria secundaria.

Aunque el almacenamiento orientado a archivos es la opción más sencilla y común, el uso de sistemas de gestión de bases de datos (SGBD) es la solución más versátil (Boehm, 2014; Deibe et al., 2018; Hongchao & Wang, 2011; Lee & Kang, 2015; Schön et al., 2013). La forma original de almacenar un modelo de nube de puntos en una base de datos es utilizar una fila para cada punto en una base de datos relacional, incluidos todos sus atributos. En el pasado, este enfoque era común entre las aplicaciones destinadas a GIS, donde se almacenaban conjuntos de puntos de tamaño moderado en una base de datos espacial para soportar consultas espaciales, principalmente 2D. Sin embargo, este esquema no escala bien, y por tanto no es válido para gestionar los modelos de nube de puntos que son comunes hoy en día (Boehm, 2014). Por ello la siguiente solución lógica es almacenar un grupo de puntos en cada fila, como lo hacen sistemas como Oracle Spatial o PostGIS. Además de esto, existen otras soluciones basadas en bases de datos relacionales para trabajar con nubes de puntos e información espacial, principalmente utilizando la base de datos como indexador de conjuntos de datos almacenados como bloques binarios o como archivos externos (Lee & Kang, 2015; Schön et al., 2013).

Por otro lado, las bases de datos NoSQL tienen algunas ventajas sobre las bases de datos relacionales en aplicaciones de big data. Entre ellas, las bases de datos orientadas a documentos

(MongoDB, Cassandra, Couchbase, etc.) han ganado popularidad debido a su capacidad para manejar grandes volúmenes de datos de manera eficiente y escalable mediante el uso de sharding (fragmentación). Debido a esto, suelen ser la opción preferida para almacenar nubes de puntos a gran escala, tanto conjuntos de datos semiestructurados como no estructurados (Boehm, 2014; Deibe et al., 2018; Hongchao & Wang, 2011). No obstante, el éxito de los enfoques basados en bases de datos NoSQL no debe hacer pensar que las bases de datos relacionales han perdido relevancia en aplicaciones de big data, particularmente considerando su madurez, eficiencia y adopción generalizada. Hay varios enfoques que utilizan SGBD relacionales para almacenar y procesar conjuntos de datos LiDAR (Cura et al., 2015; Schön et al., 2013).

En la descripción del modelo SPSLiDAR (Rueda-Ruiz et al., 2022) que se realiza en esta tesis se propone una implementación de referencia que aprovecha las ventajas del uso de formatos específicos para nubes de puntos, así como el uso de sistemas de bases de datos para almacenarlos e indexarlos. La parte más relevante del esquema propuesto es el modelo de datos del SGBD, que define la estructura lógica y determina cómo se pueden almacenar, organizar y recuperar los datos.

1.5. Procesamiento de nubes de puntos

Aunque no es objetivo de esta tesis, el procesamiento de nubes de puntos es uno de los aspectos principales que motivan su desarrollo, y por tanto merece una mención en este capítulo con objeto de encuadrar mejor el trabajo. En el proyecto en el que se integra esta tesis se trabaja sobre todo con el procesamiento inteligente de nubes de puntos, orientado a la segmentación y al aprendizaje profundo.

Segmentación de nubes de puntos

Los métodos de segmentación de nubes de puntos son operaciones destinadas a agrupar subconjuntos de puntos geométricamente continuos que comparten una o más características en común (geométricas, radiométricas, etc.) (Tóvári & Pfeifer, 2005). El proceso de segmentación permite la clasificación o asignación de dichos segmentos a clases específicas de acuerdo a diferentes criterios y suele ser un paso necesario en muchas aplicaciones para explotar la información intrínseca en la nube de puntos, ya que permite identificar los elementos relevantes. El problema de la segmentación es un área de investigación muy activa en la actualidad, debido por un lado a la gran importancia que ha cobrado el trabajo con nubes de puntos en multitud de campos y aplicaciones, y por el otro a la complejidad y variedad de las nubes de puntos debido a los distintos tipos de muestreo, densidades variables, distintos elementos a identificar, etc. (Grilli et al., 2017).

En los últimos años se han desarrollado numerosos métodos de segmentación, que de acuerdo a distintos autores (Grilli et al., 2017; M. Li, 2018; A. Nguyen & Le, 2013), se pueden clasificar en distintas categorías atendiendo a la estrategia usada, siendo las principales la detección de límites o fronteras, el crecimiento de regiones, el ajuste a modelos y el uso de aprendizaje automático. Los métodos basados en límites o fronteras primero detectan los límites entre regiones cuyos puntos tienen distintas propiedades (normales, gradientes, curvatura, etc.) y después agrupan los puntos entre los límites definidos (Rabbani et al., 2006). Existen diversas

propuestas, aunque su principal inconveniente es la limitada precisión en la segmentación en caso de ruido o variabilidad en la densidad de escaneado (Grilli et al., 2017; A. Nguyen & Le, 2013). Los métodos de crecimiento de regiones están basados en definir una serie de puntos como semilla, a partir de los cuales se van construyendo regiones agrupando los puntos vecinos con características similares de orientación de la superficie, curvatura, etc. (Grilli et al., 2017), y que se diferencian de las regiones anexas con distintas propiedades. Están menos afectados por el ruido y otras distorsiones, a causa del uso de información global de las nubes de puntos (Liu & Xiong, 2008), pero son menos precisos a la hora de establecer límites (A. Nguyen & Le, 2013). Los métodos de ajuste a modelos se basan en que los objetos pueden ser descompuestos en primitivas espaciales como planos, conos, cilindros o esferas, de tal manera que los puntos que forman una primitiva dentro de una nube de puntos se etiquetan como un segmento. Hay dos algoritmos que han sido ampliamente empleados: la transformada de Hough, que permite detectar planos (Vosselman et al., 2004), o cilindros y esferas (Rabbani et al., 2006), y el conocido como RANSAC (RANdon SAmple Consensus) que detecta formas mediante la selección aleatoria de conjuntos mínimos de puntos para construir candidatos a primitivas, que son validados con la nube determinando qué porcentaje de puntos se ajustan bien al modelo de primitiva. Este proceso de generación de primitivas se repite preservando aquellas que han obtenido mejor ajuste. Este tipo de métodos son muy rápidos y robustos respecto a los valores atípicos u outliers (ruido), y su eficacia para la detección de formas ha sido comprobada en diversos trabajos (Bosché, 2012; Chen et al., 2014; Grilli et al., 2017; Schnabel et al., 2009), aunque presentan limitaciones para expresar formas complejas y poco definidas.

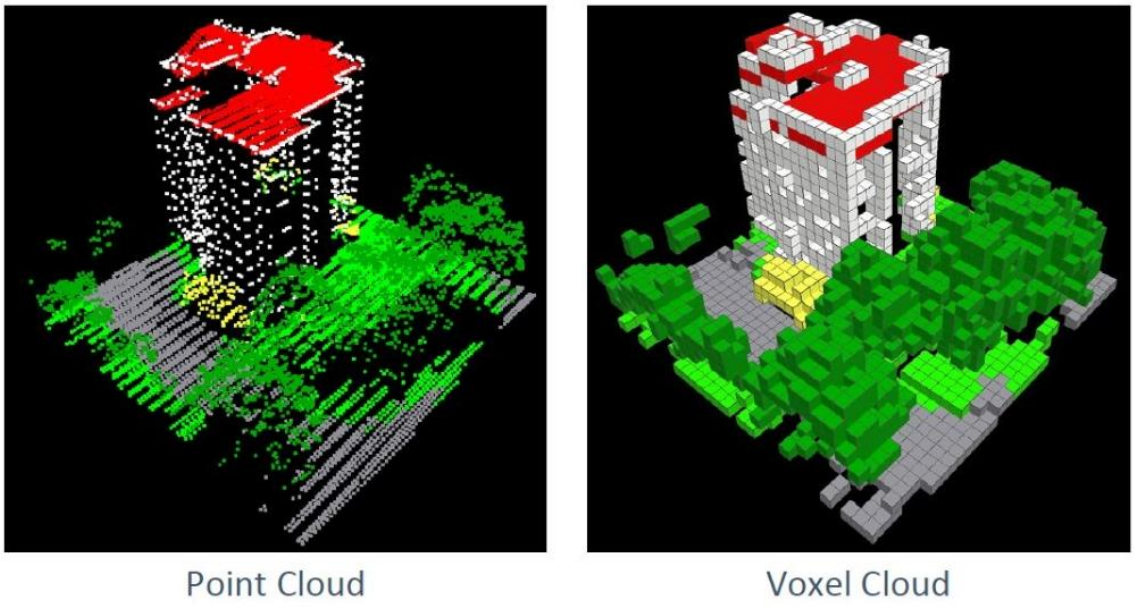


Figura 1.8. Segmentación de una nube de puntos mediante una CNN (*Convolutional Neural Network*) que toma una voxelización como entrada (Schmohl & Sörgel, 2019)

Figura 1.9. Resultado de una segmentación semántica con PointNet (Qi et al., 2017)

Un paso más allá son los métodos de aprendizaje automático (Grilli et al., 2017; M. Li, 2018), incluyendo los agrupamientos basados en k-means y agrupamientos jerárquicos basados en árboles de decisión. Recientemente las técnicas de aprendizaje profundo han comenzado a aplicarse obteniéndose resultados excelentes en datasets con información geométrica y/o componentes RGB (Dai et al., 2017; Du et al., 2021; Yoo et al., 2020; Zhang et al., 2018). De igual modo se han logrado avances en la comprensión semántica de las muestras 3D a partir de los mapas de profundidad (Song et al., 2016). En cualquier caso, el principal problema con el que se enfrentan estas soluciones radica en la necesidad de transformar el espacio de representación de las nubes de puntos a estructuras discretas (por lo general volumétricas, como en la Figura 1.8) (Maturana & Scherer, 2015; Zhou & Tuzel, 2017), que puedan servir como datos de entrada a las redes construidas. No obstante, también han aparecido numerosas arquitecturas de red orientadas a la clasificación y segmentación de nubes de puntos en su representación habitual (Boulch, 2020; J. Li et al., 2018; Y. Li et al., 2019; Qi et al., 2017; Y. Wang et al., 2018; Yan et al., 2020), como puede verse en la Figura 1.9.

Comparación de nubes de puntos

La comparación de modelos de nube de puntos tomados en diferentes momentos permite controlar la evolución de un modelo a lo largo del tiempo. Existen numerosas aplicaciones que usan la comparación de nubes de puntos, como el control ambiental y de la masa forestal, el control de la erosión del suelo, la monitorización de costas, lagos y ríos, el control de la contaminación acústica a partir de mapas de ruido o la documentación y rehabilitación del patrimonio histórico y cultural, entre otros. En general el procesamiento 4D de nubes de puntos no está todavía muy explotado, y solo se encuentran algunos trabajos que las utilizan de forma muy concreta como posprocesamiento automático (Józsa et al., 2013) o para detección de movimiento en disciplinas como la robótica, aunque recientemente se están aplicando en temas como planificación urbana (Tran et al., 2018) (Figura 1.10).

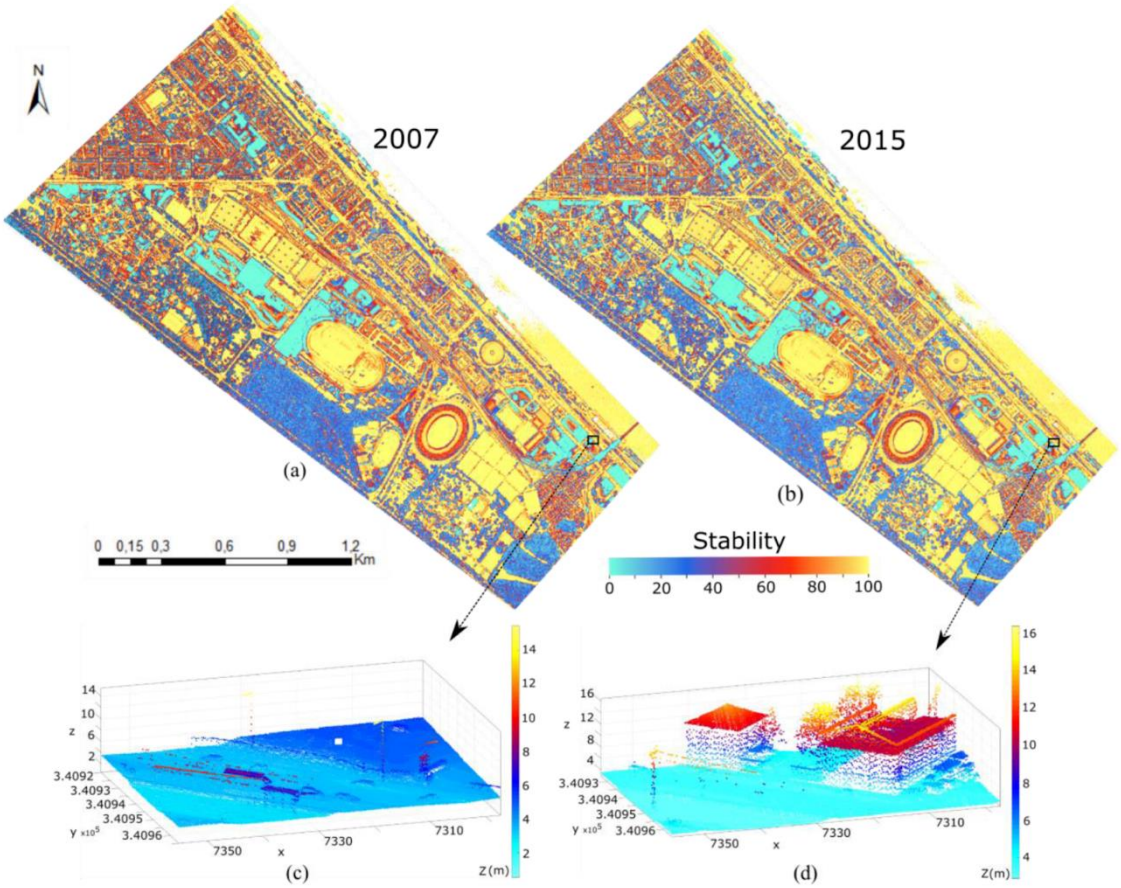


Figura1.10. Característica de estabilidad para la detección de cambios: (a) estabilidad mostrada para 2007 valor de color, que va de 0 (cian) a 100% (amarillo); (b) estabilidad mostrada para 2015; (c) nube de puntos 2007 mostrada como valor de altura en un área específica; (d) nube de puntos de 2015 mostrada como valor altura en un área específica; e) rango de valores de estabilidad en una zona específica de 2007; y (f) rango valores de estabilidad en un área específica de 2015 **(Qin et al., 2018)**.

La detección de cambios es uno de los principales temas en teledetección y, como resultado, ha recibido una considerable atención en los últimos 40 años. Tradicionalmente, la mayoría de los enfoques han abordado el problema utilizando imágenes 2D normales o con datos espectrales adquiridas con sensores satelitales o aéreos, aplicando una amplia gama de técnicas (D. Lu et al., 2004; Shi et al., 2020; Si Salah et al., 2019). Hay varios enfoques que se centran en convertir las nubes de puntos 3D en modelos 2D, conocidos como modelos digitales de terreno (DSM), donde cada píxel almacena la información de elevación en un punto determinado en el plano. Gracias a esto, se pueden aplicar métodos bien conocidos de comparación de imágenes 2D, que no obstante pueden perder una cantidad significativa de información durante la reducción de dimensionalidad e introducir errores de interpolación. Por esta razón, estos métodos apenas son aplicados a nubes de puntos densas y complejas producidas por escaneado 3D.

A diferencia de los métodos DSM, hay otros enfoques que abordan el problema procesando las nubes de puntos directamente. Qin et al. (2016) hizo una revisión exhaustiva de los métodos existentes y los organizó en dos categorías principales: los basados en comparación geométrica (distancia euclidiana, diferencias de altura o diferencias de proyección DSM), y los basados en análisis del espectro de la geometría (posrefinamiento de métodos DSM, fusión directa de características y posclasificación). Una de sus principales conclusiones es que los métodos

basados en diferentes métricas son difíciles de comparar, debiendo tenerse en cuenta el uso que se le va a dar a los resultados a la hora de elegir un método u otro. Otra contribución importante a este campo la hizo de Gélis et al. (2021), que presentan una comparación experimental de seis métodos diferentes de detección de cambios en la nube de puntos. Los autores dividen los métodos según la evolución histórica de esta área de trabajo y la técnica utilizada según una taxonomía que comprende las siguientes categorías: métodos de detección basados en DSM, métodos directos y métodos de detección basados en aprendizaje automático.

Evidentemente, siempre existe la posibilidad de estudiar los cambios de los datos de forma manual, es decir, mediante la gestión por parte del usuario de distintos archivos sobre los que se calculan por pares las diferencias con programas como CloudCompare. El reto en este aspecto es la integración de esta funcionalidad en una herramienta de mayor envergadura. Otro aspecto poco tratado es el de la visualización simultánea de varios datasets pertenecientes a una serie temporal, de forma cómoda y funcional, que proporcione un significado preciso de las diferencias percibidas de los datos. El sistema propuesto en esta tesis simplifica enormemente la construcción de un sistema que permita la gestión de series temporales, ya que permite almacenar distintas versiones de múltiples datasets en el mismo servidor.

Generación de geometría

Un modelo basado en puntos puede ser suficiente para muchas aplicaciones. Sin embargo, otras como la visualización realista de alta calidad o la simulación científica (resistencia estructural, erosión, etc.) requieren un modelo representado por una superficie, normalmente una malla de triángulos (Turk & Levoy, 1994). La generación de modelos de superficie a partir de una nube de puntos ha recibido una gran atención por parte de la comunidad científica, existiendo al menos un centenar de artículos al respecto (H. Huang et al., 2009; V. S. Nguyen et al., 2014; Nina et al., 2007; X. Wang et al., 2013). La mayoría de estos métodos requieren nubes de puntos de partida de alta calidad escaneadas en un entorno controlado. Sin embargo, la calidad de un modelo obtenido en una escena exterior puede ser muy variable, debido a factores ambientales como la existencia de obstáculos y la inaccesibilidad de ciertas zonas.

En general las mallas obtenidas suelen ser muy complejas por lo que son candidatas a un proceso de simplificación, que suele facilitarse por el hecho de que la distribución de los puntos suele ser regular (Figura 1.11). La tecnología LiDAR utiliza una estrategia de barrido que genera una nube de puntos ordenada, mientras que un escáner de mano permite realizar pasadas en direcciones arbitrarias, lo que genera discontinuidades e irregularidades en la nube final obtenida. En el caso de disponer de nubes de puntos desordenadas (el caso general, y la presunción más prudente), la obtención de geometría es un proceso complejo, ya que no existe información sobre conectividad entre puntos ni tampoco información sobre la normal a la superficie muestreada en cada uno. La aplicación de un método clásico de generación de superficies a un modelo de este tipo produce resultados en general pobres, por lo que es conveniente trabajar en esta línea adaptando métodos existentes (V. S. Nguyen et al., 2014; B. Yang et al., 2016), o desarrollando métodos nuevos, ya sea automáticos o con cierta asistencia por parte del usuario.

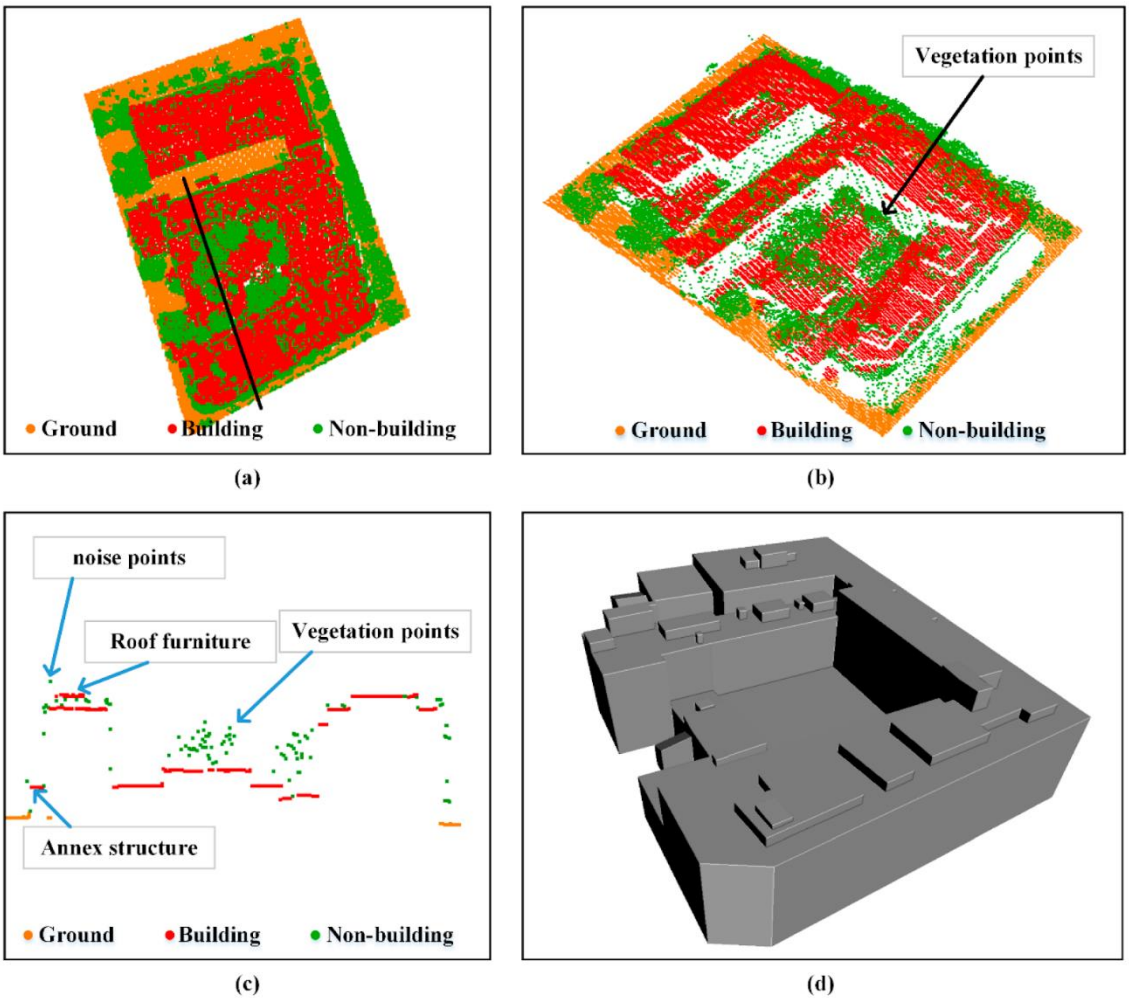


Figura1.11. Resultado de la detección de un edificio. (a) vista superior del resultado de la detección de edifi (b) vista lateral; (c) sección transversal de la línea negra en (a) una descripción detallada del resultado de l detección, donde se preservan los mbles del techo y las estructuras anexas, y se eliminan los puntos de vegetación y los puntos de ruido; (d) modelo del mobiliario obtenido tras la reconstrucción correspondiente (B. Yang et al., 2016).

Además de la generación de superficies, en muchas aplicaciones, una de las principales utilidades del modelo obtenido tras el escaneado, es el cálculo de perfiles o secciones transversales del modelo en diferentes ubicaciones, es decir, el resultado de la intersección de un plano de corte arbitrario con la nube de puntos o su triangulación. En el caso de la nube de puntos, realmente esta intersección debería ser un conjunto vacío puesto que el modelo no es una superficie. En cualquier caso, el objetivo es calcular una aproximación de la proyección del modelo basado en puntos obteniendo una polilínea que puede ser cerrada o contener agujeros.

Las secciones transversales facilitan la toma de medidas, el cálculo de áreas y volúmenes, la comparación y estimación de la desviación del modelo real respecto a un modelo teórico previo (Figura 1.12), la planificación de actuaciones de restauración, etc. También puede ser de utilidad como método de obtención de una malla de triángulos a partir del modelo de puntos (Park et al., 2007). Para este fin, existen algunos enfoques en la literatura que utilizan técnicas heurísticas sin una estimación clara del error cometido en la aproximación. Es motivo de estudio el abordar el problema mediante otros enfoques matemáticamente más rigurosos y precisos, como los contornos activos (Kass et al., 1988; Terzopoulos & Szeliski, 1992). Aparte de mejoras de

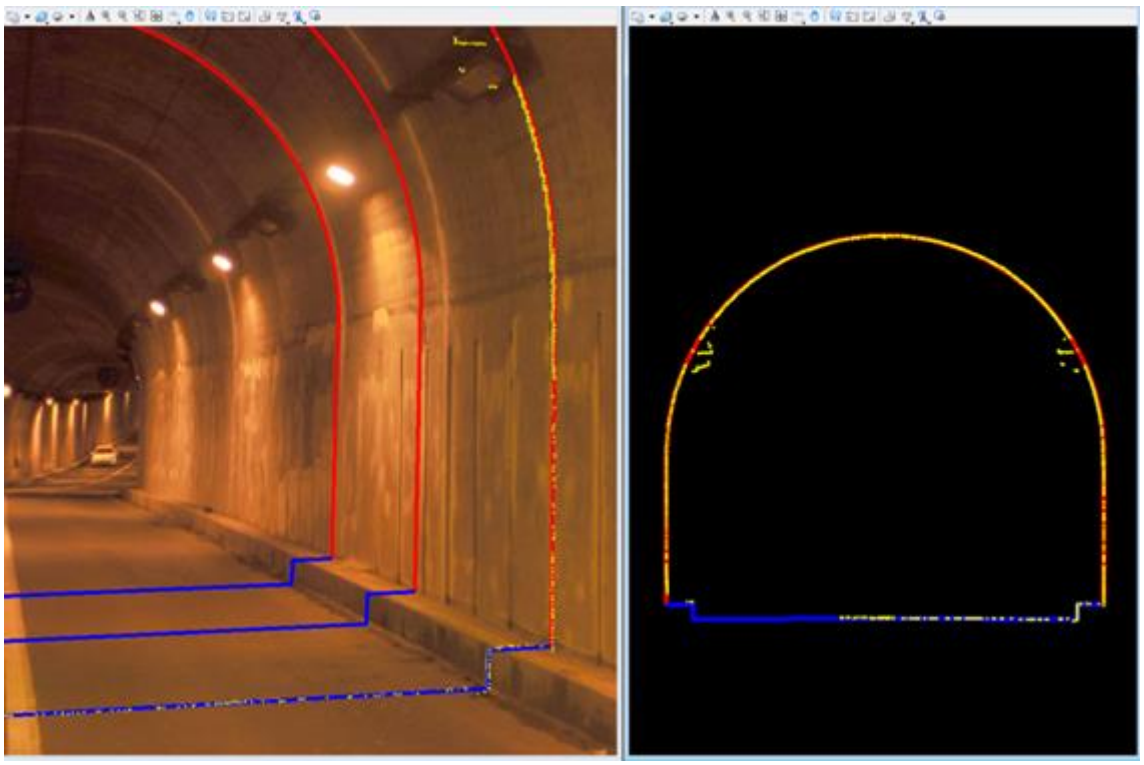


Figura1.12. Cálculo de sección transversal de un túnel. © Artescan.

implementación usando GPU, hebras y estructuras espaciales diversas, hay muchos enfoques nuevos que podrían experimentarse, ya que es un tema poco estudiado.

1.6. Sistemas de coordenadas de referencia

Los modelos de nube de puntos adquiridos mediante sensores representan partes del mundo real. El uso de los sistemas de coordenadas de referencia (SCR) permite georreferenciar estos puntos y define un marco común sobre el cuál interpretar los datos. Esta tesis no pretende profundizar en el ámbito de los SCR, al ser un tema complejo que involucra múltiples tipos de sistemas y modelos matemáticos tanto para la obtención de estos como las transformaciones entre distintos SCR. Sin embargo, consideramos necesario realizar una breve introducción a los SCR para poder contextualizar la información relacionada con este tema en capítulos posteriores.

Tradicionalmente los SCR se clasifican en dos tipos: los sistemas de coordenadas geográficos (SCG) y los sistemas de coordenadas proyectados (SCP). Los primeros utilizan una representación tridimensional de la Tierra, mientras que los segundos utilizan un modelo bidimensional. Un SCP se deriva a partir de un SCG: es el resultado de aplicar una serie de transformaciones para obtener una representación planar del SCG y adecuarlo a formatos como mapas o pantallas.

Un SCG está compuesto por tres elementos principales. El geoide especifica el aspecto real de la Tierra: un modelo sencillo define una superficie lisa, frente a uno complejo que tendrá en cuenta las irregularidades del planeta debidas a fuerzas gravitacionales. En la Figura 1.13 se

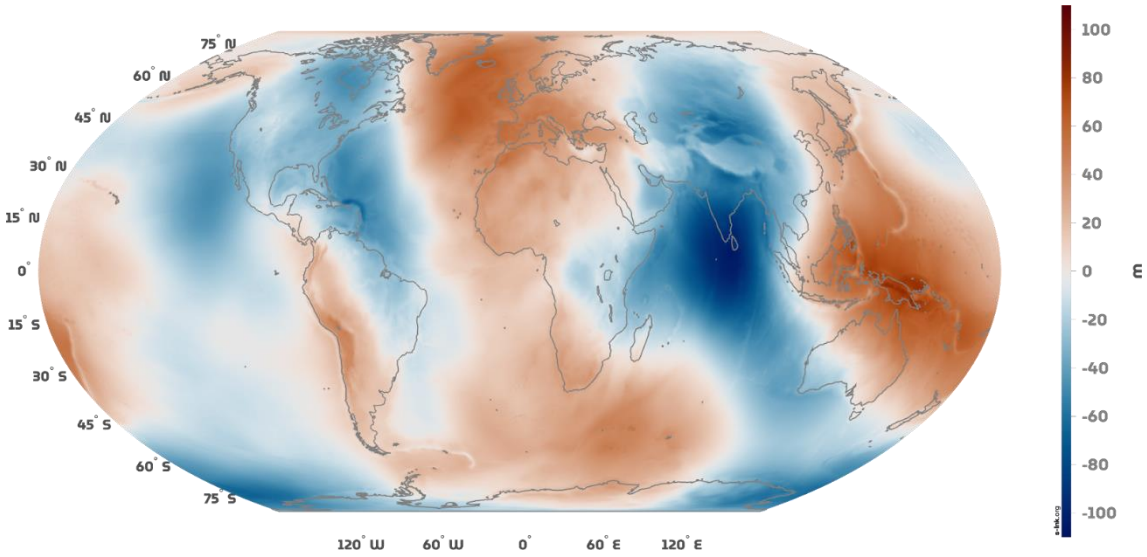


Figura1.13. Representación 2D del geoide de la tierra basado en EGM2008. El color indica la diferencia en en cada punto con respecto la superficie.

muestra el geoide con código EGM2008. En la práctica, estos modelos plantean dificultades a la hora de trabajar sobre ellos, así que se simplifica su forma definiendo una elipsoide, un cuerpo compuesto por los valores de los radios al ecuador y los polos. El tercer elemento es el datum, que determina cómo se alinean el geoide y el elipsoide: un SCG local suele utilizar un elipsoide definida para una región concreta y realiza el alineamiento entre esta y el geoide en base a un punto específico de la superficie terrestre; el resultado son modelos muy precisos en la región donde se ubica ese punto. Un SCG geocéntrico por su parte utiliza un elipsoide que representa el planeta en su totalidad y utiliza un punto en el centro del geoide para llevar a cabo el alineamiento. De este modo se define un modelo válido a nivel global, aunque la precisión pueda ser algo menor en comparación a un SCG local.

Dentro de los datums, se distinguen tres tipos principales: datums horizontales, datums verticales y datums tridimensionales. Un datum horizontal permite definir valores en X e Y sobre el elipsoide. Un datum tridimensional extiende el datum horizontal introduciendo la componente de altura, utilizando también como referencia el elipsoide. Un datum vertical es un datum específico construido para dar valores únicamente en el eje Z, utilizando puntos de referencia distintos al elipsoide, como pueden ser el nivel medio del agua. Un datum vertical y un datum horizontal se pueden combinar para poder definir valores para una coordenada en los tres ejes.

Un SCG utiliza unidades angulares. Esta es otra de las diferencias con respecto a los SCP. Los sistemas proyectados permiten utilizar coordenadas cartesianas, simplificando las operaciones aritméticas como pueden ser el cálculo de distancias y volúmenes. Existen múltiples técnicas para construir un SCP a partir de un SCG. En el contexto de esta tesis, nos centraremos en la proyección Universal Transverse Mercator (UTM). La proyección UTM es una de las más populares y se utiliza frecuentemente en modelos de nube de puntos. El modelo UTM particiona la Tierra en distintas zonas, utilizando un criterio longitudinal que define bandas de 6º numeradas del 1 al 60 de oeste a este, empezando en la línea internacional de cambio de fecha. Estas bandas son a su vez divididas en dos, utilizando el ecuador como referencia para separar entre norte y sur. Cada zona por lo tanto queda identificada por un patrón compuesto por el identificador numérico de la banda (del 1 al 60) y la letra N o S. Dentro de una zona se define un sistema de

coordenadas propio. El origen de coordenadas se asocia siempre al valor 500.000 metros. Este valor refleja el meridiano central de la zona UTM, situado 3º al este del borde de la banda. Este modelo permite evitar por lo general valores negativos: en el ecuador, lugar donde la zona tiene su máxima amplitud, el valor mínimo al oeste dentro del área de la banda sería 166.000 metros. Con respecto al eje vertical, se utiliza el ecuador como punto de referencia: para la zona N, el origen toma valor 0 en el ecuador, mientras que para la zona S toma el valor 10.000.000, con el objetivo de, al igual que en el eje horizontal, evitar el uso de valores negativos.

Con el modelo UTM, cada punto de la superficie estará asociado a una zona UTM: la banda se calcula como ¨Ω à é ¢ â è ¨Ω¨Ω‰ò°ê ¨Ω y la latitud es mapeada a N o S en función de si es negativa o no. Sin embargo, sigue siendo posible mapear un punto a cualquier otra zona UTM: por ejemplo, si tenemos una nube de puntos adquirida en una región que intersecta con dos zonas UTM, el mismo sistema de coordenadas puede ser utilizado para todos los puntos, incluso aquellos para los que la fórmula del cálculo de banda tuviese un valor distinto. Si las distancias no son grandes, el error asociado será asumible. Hay que tener en cuenta que incluso los puntos que caen de forma natural dentro de la zona UTM de referencia se ven a cierto nivel afectados por la distorsión conforme se alejan del meridiano central.

El modelo UTM además se puede ver ampliado con las zonas definidas por el Military Grid Reference System (MGRS). Este incluye un nivel de subdivisiones adicional, particionando las bandas a nivel de latitud en intervalos de 8º, identificadas por letras que van de la C a la X. Además, introduce un sistema de representación de datos partiendo de la definición de cuadrículas con distintos niveles de resolución, aunque en este trabajo nos ceñiremos únicamente al sistema de asignación de letras al ser utilizado frecuentemente junto al modelo UTM, ilustrado en la Figura 1.14.

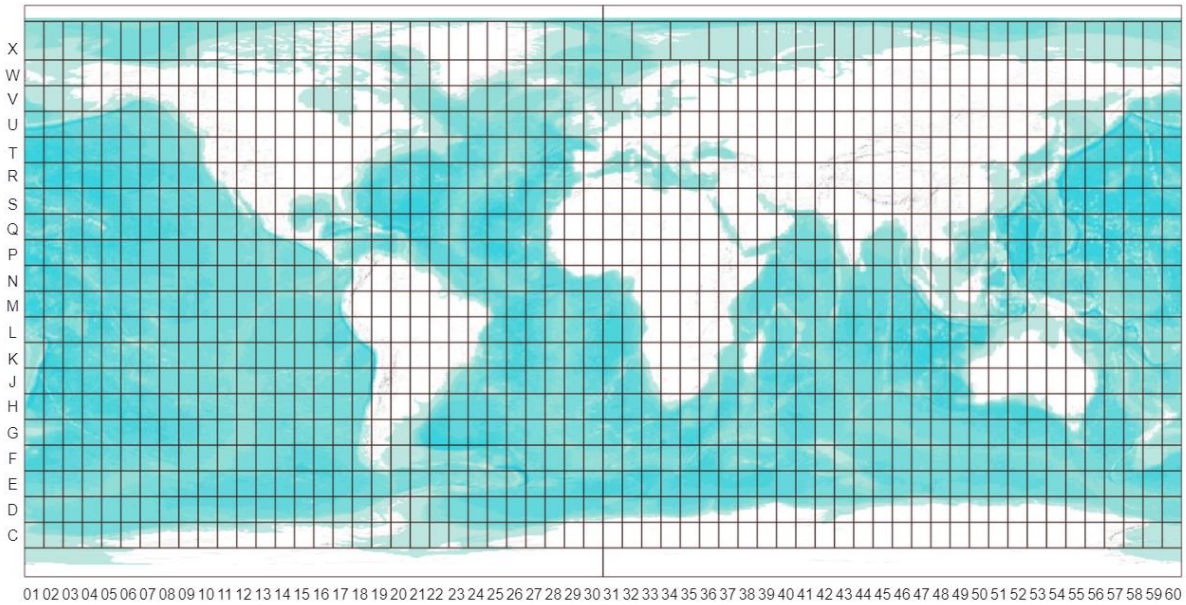


Figura1.14. Mapa que muestra las zonas definidas por UTM y MGRS. Las bandas longitudinales se extienden de 01 a 60 mientras que las latitudinales de las letras C a la X (excluyendo polos).

Los sistemas de coordenadas de referencias son conjuntos de información complejos, compuestos por múltiples atributos que requieren ser documentados bajo estándares para facilitar su interpretación. Existen múltiples formatos con los que representar estos datos: algunos como OGC WKT/WKT2 o proj4, son de tipo descriptivo: indican los distintos componentes del sistema con sus valores correspondientes. Otro de los estándares más populares es EPSG, un sistema más conciso basado en códigos numéricos que define identificadores únicos para los SCR más populares y sus componentes (elipsoides, datums…). A nivel interno, un código EPSG está asociado a una representación de OGC WKT que indica de forma detallada el sistema de coordenadas. En la Figura 1.15 se muestra un ejemplo con la forma de estos estándares.

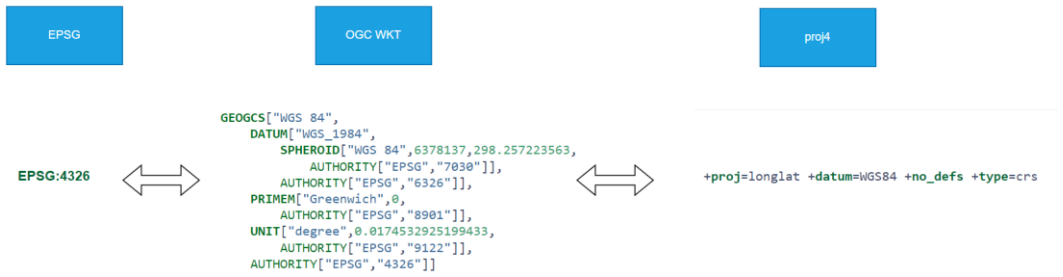


Figura1.15. Código EPSG:4326, asociado al SCG con datum WGS84, uno de los más populares a nivel global, y sus representaciones equivalentes en los estándares OGC WKT y proj4.

Capítulo 2 - DEFINICIÓN DEL MODELO: ESPECIFICACIÓN SPSLIDAR

2.1. Introducción

A pesar de la popularidad que han adquirido los modelos de nube de puntos, los flujos de trabajo en los que se integran presentan ciertas deficiencias. Los ficheros en los que se representan estos modelos necesitan ser procesados, almacenados y finalmente transmitidos a otras aplicaciones o usuarios para poder extraer valor de ellos. En muchas situaciones, la organización de estos datos no dista demasiado de la que se le podría dar a otro tipo de información presentes en un sistema de ficheros: los nombres de los archivos y los directorios que los contienen, la jerarquía de estos y algún documento de apoyo adicional en formato textual que incluya datos complementarios de la información presente son las únicas guías para poder encontrar el conjunto de datos de trabajo. A medida que aumenta el volumen de datos almacenado y la variedad de la información capturada, así como su distribución en el territorio y a lo largo del tiempo, el mantenimiento se complica, al no estar regulado por un proceso automático y ser susceptible al error humano. Del mismo modo, localizar la información necesaria se convierte en un proceso lento y poco preciso, que además se puede ver penalizado por la realización de operaciones adicionales de posprocesamiento sobre aquellos ficheros que contienen la información requerida, ya sea para reducir su tamaño y obtener una nube de puntos acotada a una región concreta del área total representada por el fichero original o aplicar una conversión a un formato diferente.

Con SPSLiDAR (Rueda-Ruiz et al., 2022) proponemos una solución genérica con la que estandarizar la integración de este tipo de información a nivel de software. SPSLiDAR define un modelo basado en tres entidades fundamentales con las que capturar los aspectos que consideramos clave a la hora de trabajar con modelos de nube de puntos. Este modelo organiza la información de las nubes de puntos a varios niveles. A nivel global, aporta información sobre qué representa una nube, cómo fue obtenida, su relación con otras nubes adquiridas, a qué región del espacio está asociada o cuándo fue llevado a cabo el proceso de adquisición. A nivel interno de cada una de estas nubes, el modelo permite definir estructuras con las que organizar la volumetría total de puntos que tiene asociada: de este modo, en vez de trabajar sobre el modelo de nube de puntos como una única entidad de gran tamaño, esta se descompone en submodelos, con una volumetría menor, controlada y generadas a través de un proceso de particionamiento preestablecido. Esta estructura puede ser explorada para localizar y recuperar de forma parcial solamente aquellas secciones del contenido original que sean de interés. Para ello, se define una entidad propia sobre la cual establecer relaciones y facilitar esta navegación.

Para poder interactuar con este modelo se ha planteado un enfoque orientado a servicios, definiendo una API REST que permite acceder a las entidades del modelo y llevar a cabo operaciones de consulta, análisis o ingesta de datos. El uso de una API REST se debe a dos motivos. Primero, es un estilo de arquitectura para APIs definido sobre el protocolo HTTP que destaca por su simpleza, legibilidad y popularidad, hasta el punto de haber sustituido completamente el antiguo estilo de servicios web basados en SOAP, que era el utilizado en los primeros estándares de servicios geoespaciales como WMS y WCS. En segundo lugar, el enfoque REST se adhiere de forma natural a la solución que planteamos, con un enfoque centrado en los recursos que definen el modelo subyacente y las relaciones entre ellos.

Esta API ha sido planteada para ser implementada por un repositorio que exponga servicios web para modelos de nube de puntos y debe ser interpretada dentro de dicho contexto. Si bien esta casuística será el principal foco de atención de esta tesis, el modelo de SPSLiDAR

puede ser utilizado en otro tipo de aplicaciones (como pueden ser los propios clientes que puedan a llegar a interactuar con el repositorio). Los métodos necesarios y la forma de la API para poder interactuar con el modelo de SPSLiDAR podrán variar en función de cada escenario.

2.2. Descripción del modelo SPSLiDAR

El modelo de SPSLiDAR propone tres entidades fundamentales: *Workspace*, *Dataset* y *Datablock.* En las siguientes secciones de este capítulo detallaremos el propósito de cada una de ellas, sus atributos y relaciones, así como otras entidades que serán necesarias para representar aspectos complementarios del modelo. En SPSLiDAR no existe una entidad para representar un punto de la nube. Un *datablock* será lo más próximo a nivel conceptual, puesto que representa un subconjunto de puntos relacionados en base a algún criterio, normalmente espacial. Los puntos que tiene asociado un *datablock* se almacenan usando formatos estándar de ficheros de nubes de puntos, como LAS, LAZ o E57, al que este *datablock* enlazará de forma directa. El *datablock* actúa como una abstracción de las características generales de este fichero, pudiéndose llevar a cabo consultas de filtrado sobre los *datablocks* para encontrar datos que satisfagan dichos criterios. Los ficheros, una vez construidos, se comportan como una caja negra, siendo en principio su interior opaco para la implementación del modelo. El uso de este enfoque a nivel de fichero permite optimizar múltiples procesos a nivel de almacenamiento y consulta de la información, aspectos sobre los que profundizaremos a lo largo de esta tesis.

En la Figura 2.1 se muestra un diagrama UML de clases con las entidades clave presentadas previamente, los atributos básicos de estas y las relaciones entre ellas. A lo largo de este capítulo se detallarán cada uno de estos elementos y su propósito, con el fin de proveer una visión completa del modelo, la necesidad de cada uno de sus componentes y cómo interactuar con ellos. Si bien múltiples de estos campos son primitivas comunes en la mayoría de lenguajes de programación, también existen algunas entidades adicionales propias del modelo que serán presentadas posteriormente.

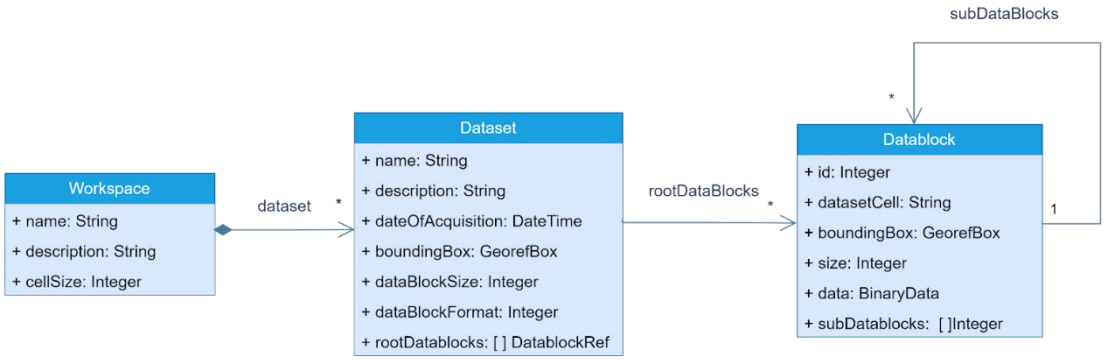


Figura2.1. Diagrama de clases básico de SPSLiDAR

Si bien SPSLiDAR pretende ser un modelo genérico y sencillo, en ciertos casos alcanzar este equilibrio puede ser complicado y requiere algunos compromisos. Por defecto, se ha apostado por proponer soluciones lo más concisas y simples posibles. También veremos en posteriores

secciones de este capítulo nuevas alternativas con el objetivo de ofrecer una solución más abstracta y parametrizable para aquellos escenarios que lo requieran.

Elementos del modelo

Workspace

La entidad *Workspace* marca el punto de entrada a un conjunto de nubes de puntos. Es la entidad de mayor nivel de abstracción. Actúa como un contenedor a nivel conceptual sobre el que basar el almacenamiento de distintos escaneos (representados por la entidad *Dataset*) que tienen algún tipo de relación entre sí. No se establece ninguna validación que determine o compruebe la naturaleza de estas relaciones; será el cliente que interactúe con el modelo quien deba decidir si múltiples *datasets* deben estar asociados a un mismo *workspace*. Algunos criterios posibles pueden ser el uso de una misma instancia para las nubes de puntos asociadas a un mismo proyecto o aquellas realizadas por una misma persona u organización. Sus atributos son:

- ***name*** (String)**:** este atributo es un identificador único dentro del sistema. Determina el nombre del *workspace*.

- ***description*** (String): debe ser una descripción a grandes rasgos del propósito del *workspace* y el tipo de información con el que se pretende poblar. Deberá de servir de apoyo a los usuarios para determinar si un *dataset* puede o no encajar dentro del *workspace*.

- ***cellSize*** (Integer): define el valor del tamaño de celda en la estructura para indexar datos a nivel de territorio.

Si bien los atributos *name* y *description* son fáciles de interpretar a nivel semántico, el atributo *cellSize* requiere una mayor contextualización. En la descripción realizada del modelo se introducen relaciones entre las distintas entidades, de manera que un *workspace* contiene múltiples *datasets*. Estas relaciones se implementan utilizando diferentes estructuras de datos, generalmente de tipo espacial, sobre las que profundizaremos a lo largo de esta tesis. En el caso de estructuras de tipo rejilla o grid, el atributo *cellSize* determina el tamaño de las celdas que lo componen. Para otras estructuras, este atributo puede ser omitido o complementado con otros campos adicionales: en el caso de usar un array, no sería necesario ningún atributo, mientras que en un quadtree se pueden especificar otras variables como el nivel de profundidad máximo que puede alcanzar.

Esta parametrización permite a los clientes definir un modelo adecuado a sus necesidades: si el *workspace* que definen ha sido ideado para *datasets* que albergan amplias extensiones de terreno, un *cellSize* de mayor tamaño (rango 10-100 km) puede ser una opción más interesante frente a uno más reducido (1-10 km). En posteriores capítulos se analizará cómo afecta este dimensionamiento a las estructuras de datos construidas.

Dataset

La entidad *Dataset* representa un modelo de nube de puntos desde una perspectiva general. Define el contexto de qué fue escaneado, la fecha en la que se obtuvo esta nube y la ubicación en la que fue obtenida. Por regla general, un *dataset* equivale a un objeto o lugar de interés representado por una o más nubes de puntos que han sido adquiridas en zonas adyacentes durante un periodo de tiempo cercano. Por ejemplo, si se lleva a cabo una campaña de adquisición de puntos en una región específica y como resultado el hardware y software de escaneado utilizados dan lugar a varios archivos, estos pueden ser considerados como parte de un mismo *dataset* al estar estrechamente relacionados a nivel conceptual, espacial y temporal (Figura 2.2). Esto es un marco de trabajo general y en última instancia los clientes que interactúan con el modelo tienen la libertad de decidir qué ficheros de nubes de puntos deben formar parte de un mismo *dataset*.

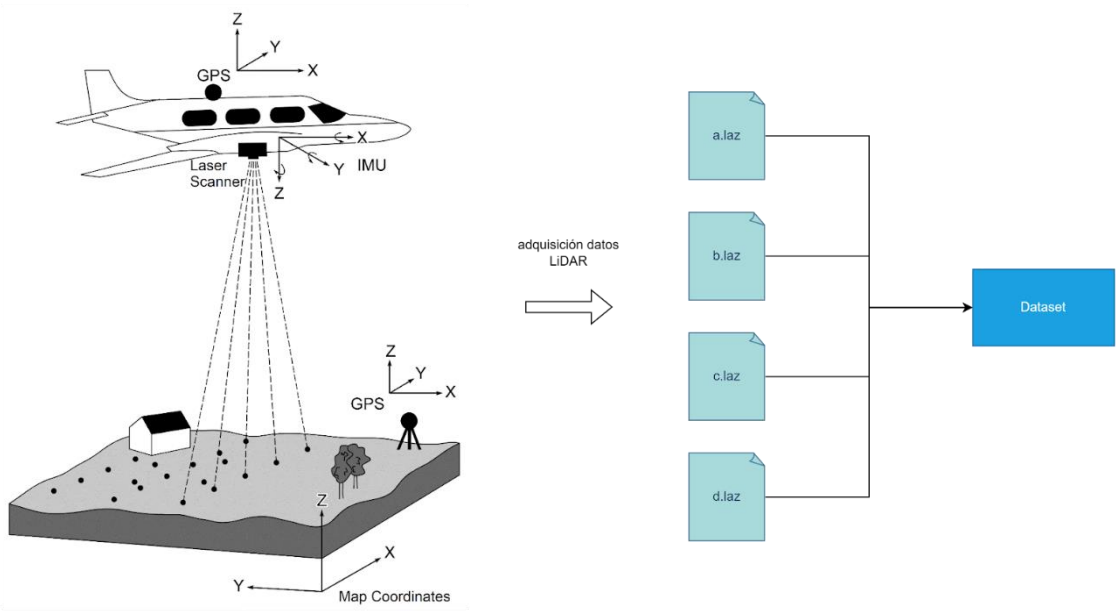


Figura2.2. Adquisición de datos LiDAR aérea. Los ficheros obtenidos son asignados al *dataset*.

Se compone de los siguientes campos:

- ***name*** (String): es un identificador nominal que debe ser único dentro del *workspace* al que esté asociado el *dataset*.
- ***description*** (String): descripción con la que aportar información adicional sobre el *dataset*, pudiendo indicar el contexto bajo el que fue adquirido, dispositivos utilizados, personas involucradas en la campaña de adquisición o cualquier otro aspecto que pudiera ser relevante.
- ***dateOfAcquisition*** (DateTime/String): representa el instante temporal al que está asociado el *dataset*. Utiliza la especificación ISO 8601 para indicar la fecha y hora asociada a esta nube de puntos. Se utilizará para dar soporte a operaciones de filtrado en la dimensión temporal. Aunque dentro de los formatos específicos para la representación de nubes de puntos normalmente encontramos un campo con una referencia temporal,

este campo debería ser imputado por el usuario de forma manual. La campaña de adquisición de una nube de puntos es un proceso continuo que se puede extender durante varias horas o días: el valor seleccionado deberá ser un momento de tiempo relevante dentro de dicho intervalo, como, por ejemplo, la hora de inicio, de finalización o un valor medio.

- ***boundingBox*** (*GeorefBox*): atributo que define la región que engloba los distintos ficheros de nubes de puntos que componen el *dataset*. Utiliza la entidad auxiliar *GeorefBox*, descrita posteriormente. Será utilizado para llevar a cabo consultas a nivel de territorio, de forma que puedan obtenerse los *datasets* localizados en un área dada.
- ***datablockSize*** (Integer): como hemos visto en la Figura 2.1, existe una relación entre un *dataset* y múltiples *datablocks*, de manera que estos últimos quedarán asociados a un *dataset*. Si bien entraremos en más detalle de la naturaleza de esta relación un poco más adelante, el propósito de este campo es definir un límite máximo en el número de puntos que podrán tener los *datablocks* que se asocien a este *dataset*.
- ***datablockFormat*** (String): define el formato de la información que se almacene en el campo *data* de la entidad *datablock*. En función de la implementación los valores permitidos podrán variar; algunas opciones habituales pueden ser LAS o LAZ.
- ***rootDatablocks*** ([ ] *DatablockRef*): contiene las referencias a los nodos raíz de la estructura de *datablocks* asociada al *dataset*. En caso de que todavía no tenga datos, será un array vacío.

La entidad *GeorefBox* define una región espacial. En esta versión de SPSLiDAR solo se utilizan representaciones sencillas en forma de caja definidas por dos puntos que constituyen sus únicos atributos:

- ***southWestBottom*** (*UTMCoordinate*): representa el valor mínimo en todos los ejes de la caja envolvente.
- ***northEastTop*** (*UTMCoordinate*): representa el valor máximo en todos los ejes de la caja envolvente.

Cada uno de estos puntos use representa por una entidad adicional, *UTMCoordinate*, que comprende los valores de estos puntos en el espacio y la zona UTM/MGRS a la que están asociados.

- ***easting*** (double): valor de la coordenada en el eje este-oeste.
- ***northing*** (double): valor de la coordenada en el eje norte-sur
- ***height*** (double): valor de la coordenada sobre la superficie.
- ***zone*** (String): zona UTM en la que está situada la coordenada.

El tipo *DatablockRef* define un subconjunto de los campos presentes en la entidad *Datablock*, compuesto por dos identificadores que permiten referenciarlo de forma unívoca.

- ***id*** (Integer): identificador del datablock que lo representa de forma unívoca dentro de la estructura de *datablocks* a la que pertenece.

- ***datasetCell*** (String): identificador de tipo textual asociado a la celda de la estructura que modela la relación *Workspace-Dataset* en la que está el *datablock*. Para una rejilla o grid 2D cada nodo puede quedar identificado por un identificador compuesto por la posición correspondiente de la celda en el eje horizontal y vertical (por ejemplo, un valor como "32-447"). A lo largo de esta tesis se presentarán distintos ejemplos concretos para las estructuras de datos definidas en este nivel.

Esta versión de SPSLiDAR es dogmática en su implementación de la gestión de la información geográfica y utiliza siempre coordenadas definidas en la proyección UTM. Algunos casos válidos pueden ser las versiones proyectadas de sistemas de coordenadas como WGS84 (con código EPSG:4326) o ETRS89 (con código EPSG:4258). Independientemente del seleccionado, el modelo no recoge ningún mecanismo para documentar el sistema de coordenadas de forma específica para los datos de tipo geográfico, por lo que se asumirá a nivel interno que han sido definidos en base al sistema de coordenadas de referencia de la implementación. Esto requiere que los clientes que interactúen con el modelo sean capaces de llevar a cabo estas transformaciones si fuera necesario. En posteriores secciones de este capítulo se presentará una extensión del modelo base que permite la definición de distintos sistemas de coordendas bajo una misma implementación y la posibilidad de llevar a cabo transformaciones entre ellos.

Datablock

Cuando se obtienen los ficheros con los puntos resultantes de un escaneo, su distribución puede ser arbitraria. El número de archivos, el volumen de puntos contenido en cada uno de ellos o los formatos utilizados podrán variar en cada caso. El carácter incierto de esta información puede afectar negativamente al procesamiento de esta por parte de los clientes; en el caso de que los ficheros sean demasiado grandes, la latencia asociada a la recuperación del fichero será mayor, y el cliente requerirá mayor capacidad de almacenamiento y procesamiento para poder trabajar sobre la nube de puntos asociada a este.

Con el fin de poder estructurar esta información y controlar cómo se organiza una nube de puntos surge la entidad *Datablock*. El modelo original da lugar a una serie de *datablocks* que dan cobertura a la totalidad de sus puntos. El tamaño de cada *datablock* puede ser predefinido, a fin de establecer unos tiempos de respuesta predecibles y una volumetría de puntos asumible por el cliente. Los *datablocks* se generan mediante un proceso algorítmico reflejado finalmente en una estructura, la cuál puede ser elegida para satisfacer un caso de uso concreto o por su adaptabilidad a la naturaleza de los propios datos. Ofrece además una serie de atributos sobre los que filtrar, de manera que, si bien no se define en el modelo un acceso directo a nivel de punto, aun así se puedan establecer criterios de filtrado y seleccionar o descartar *datablocks* que se adhieran a ellos.

Los atributos de esta entidad son:

- ***id*** (Integer): identificador numérico de un *datablock*. Debe ser único entre los *datablocks* generados dentro de la estructura de datos. No existe un criterio establecido para la numeración, dependerá en cada caso de los detalles de la implementación. En los casos que sea posible, recomendamos utilizar valores que sean lo más relevantes posibles: por ejemplo, en el caso de estructuras como quadtrees u octrees, podemos utilizar el identificador del nodo en la estructura. Por ejemplo, para un octree, el nodo raíz puede tener un *id* 0, sus hijos inmediatos *ids* en el rango 1 a 8, y así sucesivamente. De este modo se puede extrapolar información como el nivel de profundidad a partir de este identificador a diferencia de si fuese un número meramente aleatorio.

- ***datasetCell*** (String): identifica la celda de la estructura *Workspace-Dataset* bajo la cual está localizado este *datablock*. Este campo deberá tener el mismo valor en sus hijos (*subDatablocks)*.

- ***boundingBox*** (*GeorefBox*): representa la región espacial del *dataset* a la que corresponde este *datablock*.

- ***size*** (Integer): indica el número de puntos existentes dentro de la nube de puntos asociada a este *datablock*.

- ***data*** (BinaryData): bloque binario con el fichero de la nube de puntos en formato crudo. Otros atributos como *boundingBox* y *size* obtienen sus valores de la nube que registra. El campo *datablockFormat* de la entidad *dataset* indicará el formato al que corresponde el contenido binario.

- ***subDatablocks*** ([ ] Integer): array de enteros que referencia al campo *id* de los *datablocks* con los que mantiene una relación de parentesco. El número máximo de entradas es dependiente de la estructura de datos utilizada internamente para organizar los *datablocks*. Pese a que el campo *datasetCell* también es necesario para identificar un *datablock*, este valor deberá ser igual al presente en su padre y por lo tanto, no es necesario incluir esta información en el tipo utilizado en el array.

A diferencia de las entidades *Workspace* y *Dataset*, el número de *datablocks* necesario para abarcar una nube de puntos tenderá a ser elevado. Vendrá dado por múltiples factores: el tamaño de la nube original, el número máximo de puntos permitidos por *datablock* y la estructura de datos seleccionada. La generación de esta colección de *datablocks* deberá ser abordada a través de un proceso que se encargue de procesar la nube original, normalmente integrando la información procedente de varias fuentes (como se ilustra en la Figura 2.2), y generar como salida una colección de *datablocks* construidos en base a los parámetros especificados.

Frente a las entidades ya vistas del modelo, que se construyen a través de un método en la API en el que los datos definidos por el cliente reflejarán de forma prácticamente idéntica la instancia que se almacenará a nivel interno, la generación de *datablocks* será un proceso automático resuelto internamente por un servicio. Este proceso se lleva a cabo una sola vez por *dataset*: una vez generados los *datablocks* asociados a este, estos serán guardados en el sistema de almacenamiento definido y recuperados a posteriori bajo demanda durante las consultas.

Aunque el modelo SPSLiDAR está pensado fundamentalmente para la implementación de servicios para el almacenamiento y provisión de modelos de nubes de puntos en el lado del servidor, este modelo también es aplicable, al menos de forma parcial, en el lado del cliente. Concretamente el agregado *Dataset-Datablock* puede ser utilizado en la aplicación cliente para la representación del modelo que se está procesando. De esta forma el *dataset* del cliente haría de espejo del residente en el servidor, y la estructura de *datablocks* se iría construyendo progresivamente a partir de las consultas parciales realizadas al servidor durante la sesión de trabajo.

Relación Workspace-Dataset

Esta relación permite asociar un *dataset* con el *workspace* al que pertenece. Un *workspace* que contenga un número significativo de *datasets* debería definir una estructura que optimice la búsqueda de estos en base a algún criterio, principalmente espacial. En la Figura 2.3 se muestra la relación entre las distintas entidades del modelo, ilustrando el uso de una estructura de datos espacial a nivel de *workspace* para indexar múltiples *dataset*s.

Previamente se presentó el atributo *cellSize*, encargado de controlar el tamaño de las celdas usadas en la estructura de datos espacial. En función de la estructura utilizada, su significado podrá variar: para un grid regular define el tamaño de cada una de las celdas; para un quadtree especifica el tamaño de celda en el último nivel del árbol. Es importante reincidir en el hecho de que podrá incluso ser omitido, reemplazado o complementado por otros campos en función de la parametrización necesaria por la estructura a utilizar. Un aspecto importante de esta relación a tener en cuenta es que un *dataset* puede extenderse por una o más de estas celdas, como ocurre en el ejemplo de la Figura 2.3. En este ejemplo se implementa la relación *Workspace-Dataset* mediante un grid regular de dos niveles: uno primero basado en el sistema de zonas UTM y uno segundo dentro de cada zona, con celdas de menor tamaño. En el **Capítulo 3** se describe esta estructura de datos en detalle, pero de momento basta con observar que cada una de las celdas de esta estructura se puede representar unívocamente mediante un identificador que tenga en cuenta por su zona UTM y su posición en el eje este-oeste y norte-sur dentro de la zona. Este identificador será utilizado en al atributo *datasetCell* de la entidad *datablock* para indicar en cuál de las celdas que abarca el *dataset* se encuentra.

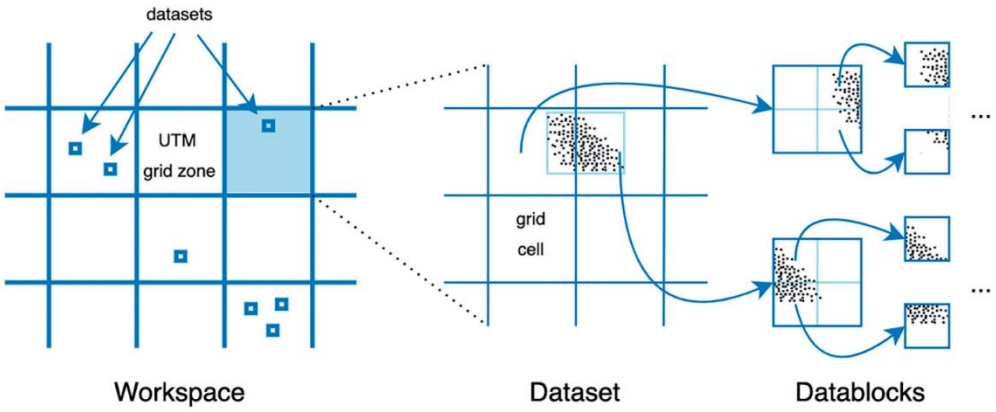


Figura 2.3. Relaciones entre las distintas entidades del modelo. En este ejemplo la relación *Workspace-Dataset* se ha implementado mediante un grid, mientras que la relación entre *Dataset-Datablock* se ha representado con un quadtree.

Relación Dataset-Datablock

La segunda relación clave en nuestro modelo es la planteada entre *datasets* y *datablocks*. El atributo *rootDatablocks* de la entidad *Dataset* define el punto de entrada en la estructura de *datablocks*, de manera que una vez localizado un *dataset*, se puede acceder inmediatamente a los *datablocks* asociados.

Esta relación se ve influenciada por la anteriormente descrita entre *workspaces* y *datasets*. Las celdas definidas a nivel de un *workspace* definen una guía sobre las que particionar la nube de puntos a partir de la que se construye la estructura de *datablocks*. En el caso de que la nube de puntos asociada al *dataset* quede distribuida por más de una celda, los puntos se distribuyen en base a estas y darán lugar a dos estructuras de *datablocks* distintas. En la Figura 2.3 se ilustra este proceso, de manera que un *dataset* asociado a dos celdas construye un quadtree para cada una de ellas, con los puntos asociados a la región definida por estas celdas. El nodo raíz de cada uno de estos dos quadtrees es referenciado en el atributo *rootDatablocks*. En la Figura 2.4 se muestra en más detalle: de las celdas denominadas #32/4/30S y #33/4/30S nace un quadtree construido de forma independiente. El identificador de cada datablock se compone por un valor numérico (#0, #1 …), que corresponde al *id* de la entidad *Datablock*, y un valor String que corresponde al *datasetCell* y que será igual al identificador de la celda.

Esta estrategia nos aporta dos ventajas fundamentales:

- Ofrece un mecanismo de control para *datasets* de gran tamaño. Para casos en los que la extensión del *dataset* es muy amplia, el fichero raíz desde el cual se construye la estructura puede ser muy grande y excesivamente costoso de procesar. Llevar a cabo una división inicial utilizando tamaños controlados agiliza el proceso y, además, permite paralelizar la ejecución del algoritmo de construcción de las estructuras de *datablocks*. Por ejemplo, si utilizamos un tamaño de celda de 10 km y tenemos una nube de puntos con una densidad media de 20 puntos/m^2, cada una de las celdas podría dar lugar a una estructura con un el número de puntos como máximo de 2.000 millones.

- Define un mecanismo nativo para agrupar ficheros en base a sistemas de coordenadas. Por ejemplo, en un proyecto concreto es posible que necesitemos trabajar con *datasets* que se extiendan por más de una zona UTM, y los ficheros con las nubes de puntos de partida pueden venir expresados en distintos sistemas de coordenadas, cada uno específico a la zona en la que fue adquirido, como EPSG:32630 y EPSG:32631. Para poder mantener el sistema de coordenadas nativo de cada uno de estos ficheros y evitar proyecciones a otras zonas que puedan causar una pérdida de precisión intolerable, este sistema de división de celdas permite que los ficheros en la zona 30S y en la zona 31S se agrupen y procesen por separado. Los ficheros deben de contar con información acerca del sistema de coordenadas bajo el cual están representados sus puntos para poder llevar a cabo esta operativa. En el estándar LAS se puede realizar a través de la inclusión de cabeceras.

Relación Datablock-Datablock

En el **Capítulo 1** de esta tesis se han presentado distintas estructuras espaciales jerárquicas utilizadas frecuentemente para organizar nubes de puntos. Cualquiera de estas estructuras puede ser integrada dentro del modelo de SPSLiDAR, asociando cada *datablock* a un nodo o celda de esta estructura. La relación entre *datablocks* no es más que un reflejo de las relaciones padre-hijo existentes entre los nodos de este tipo de estructuras. Se realiza a través del campo *subDatablocks* y permite que a través de la API, un cliente pueda navegar en la jerarquía de *datablocks* y componer nuevas llamadas utilizando los identificadores de los *datablocks* presentes en dicho campo. Esta relación puede ser ignorada en el caso de estructuras de datos no jerárquicas como las rejillas regulares o grids. En esta estructura de datos todos los *datablocks* generados están indexados directamente desde la entidad *dataset* que los contiene a través del campo *rootDatablocks*. En la Figura 2.5 se comparan las relaciones en estructuras jerárquicas y no jerárquicas.

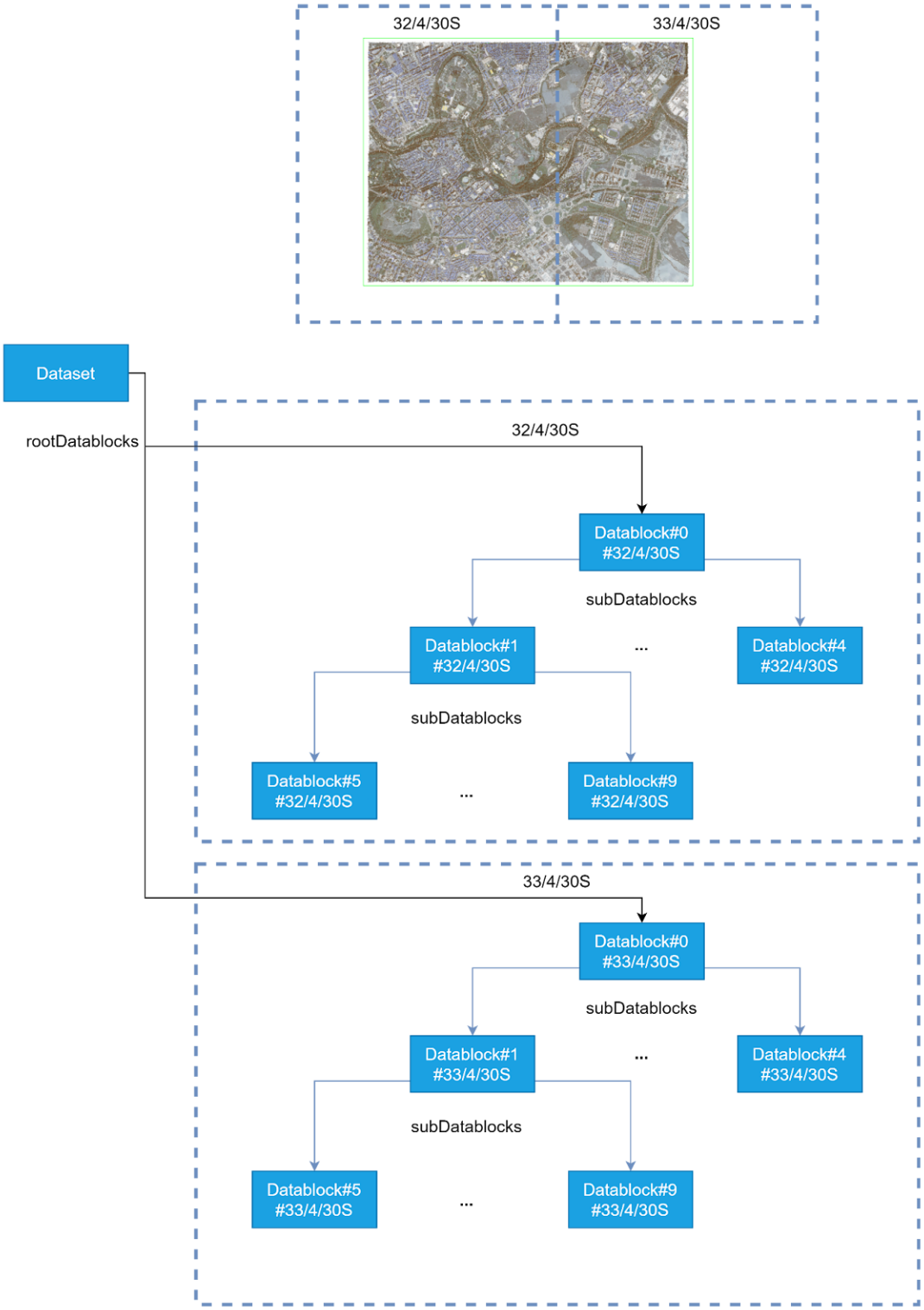


Figura2.4. Generación de múltiples estructuras a nivel de *datablock* en función del número de nodos en los que se indexa el *dataset*.

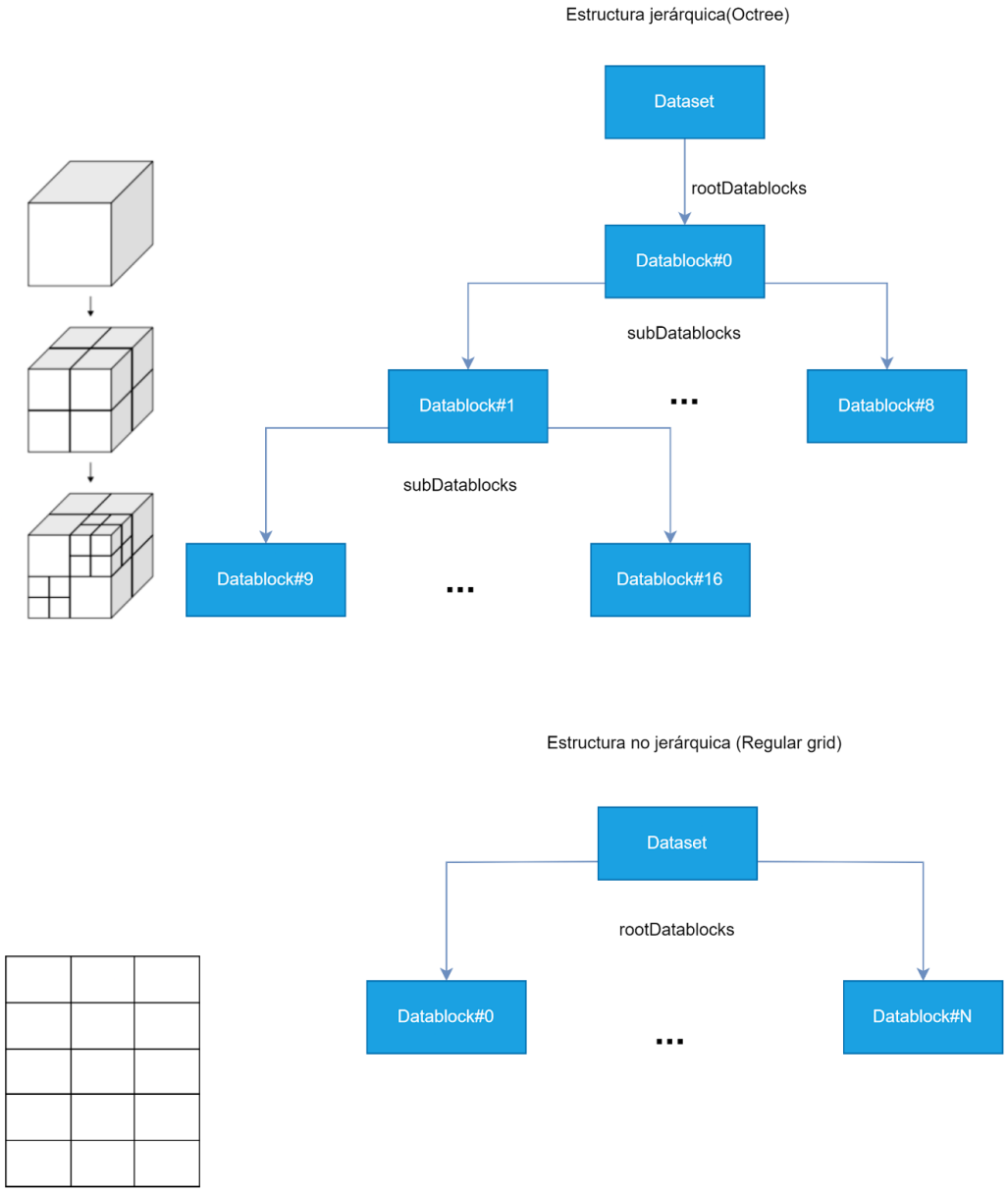


Figura2.5. Particionamiento de la estructura de *datablocks* en el caso de una estructura jerárquica (octree) y una no-jerárquica (grid regular).

2.3. API de SPSLiDAR

En la introducción de este capítulo se anticipa la necesidad de definir una API que permita interactuar con el modelo de SPSLiDAR y los motivos por los cuáles se ha decidido utilizar el paradigma REST para su diseño. La API que se presenta a continuación está orientada a repositorios de modelos de nube de puntos, definiendo los servicios clave que debe implementar un sistema con esta finalidad.

La API se estructura en torno a las tres entidades clave del modelo: *Workspace*, *Dataset* y *Datablock*. Debido a la jerarquía que se establece entre estas entidades, el acceso a un *dataset* requiere conocer previamente el *workspace* en el que está definido, y del mismo modo ocurre con

la entidad *Datablock* con respecto a las otras dos. Identificar unívocamente cada instancia es resultado de la agregación de los identificadores nominales parciales: por ejemplo, puede haber dos *datasets* que tengan el mismo nombre siempre y cuando estén en *workspaces* diferentes.

En cada nivel de la jerarquía es posible realizar búsquedas para encontrar instancias de las entidades subyacentes que satisfagan una serie de parámetros. Por ejemplo, dentro de un *workspace* concreto, es posible definir una consulta en base a criterios espaciotemporales y recuperar los *datasets* que se ajustan a ella (método **#4**).

Es importante destacar que mayoritariamente se han documentado los servicios necesarios para insertar y recuperar datos, haciendo un uso intensivo de los verbos GET y POST del protocolo HTTP. Una implementación más completa podrá incorporar servicios de modificación y borrado usando verbos PUT y DELETE sobre las mismas URIs definidas. En este caso hemos obviado este tipo de servicios puesto que a fines ilustrativos su relevancia es menor e incrementan considerablemente la extensión del documento.

ID	Verbo HTTP	URI	Descripción
#1	GET	**/workspaces**	Devuelve la lista de *workspaces*. Respuesta de ejemplo: `[ "/workspaces/CHJaen" ]`
#2	POST	**/workspaces**	Añade un nuevo *workspace*. Ejemplo del cuerpo de la petición: `{ "name": "CHJaen", "description" : "Cultural heritage sites in the province of Jaen (Spain)" , "cellSize" : 10000 }`
#3	GET	**/workspaces**/{workspace_name}	Recupera los metadatos de un *workspace*. La respuesta obtenida sigue el mismo esquema que el del método **#2** de la API.
#4	GET	**/workspaces**/{workspace_name} **/datasets** ?**southWest**={sw_coord} &**northEast**={ne_coord} &**fromDate**={from_date} &**toDate**={to_date}	Devuelve los *datasets* dentro del *workspace* especificado que satisfacen los criterios de ventana espacial y temporal definidos. `[ "/workspaces/CHJaen/datasets/CastilloStaCatalina" ,`

```
  "/workspaces/CHJaen/datasets/CatedralJaen"
]
```


#5	POST	**/workspaces**/{workspace_name} **/datasets**	Registra un nuevo *dataset*. Solo se incluyen los metadatos que lo identifican. Se le asociarán nubes de puntos a través del método **#9** de la API.


```
{
  "name": "CastilloStaCatalina",
  "description": "Castillo de Santa Catalina (Jaen)",
  "dateOfAcquisition": "2019-05-03T12:10:00Z",
  "boundingBox": {
    "southWestBottom": {
      "easting": 429522,
      "northing": 4180270,
      "height": 790,
      "zone": "30S"
    },
    "northEastTop": {
      "easting": 430638,
      "northing": 4180363,
      "height": 820,
      "zone": "30S"
    }
  },
  "data_blockSize": 10000,
  "data_blockFormat": "LAZ"
}
```


#6	GET	**/workspaces**/{workspace_name} **/datasets**/{dataset_name}	Recupera la información de un *dataset*. El campo *rootDatablocks* será un array vacío si todavía no se han añadido nubes de puntos al modelo.


```
{
  "name": "CastilloStaCatalina",
  "description": "Castillo de Santa Catalina (Jaen)",
  "dateOfAcquisition": "2019-05-03T12:10:00Z",
  "boundingBox": {
    "southWestBottom": {
      "easting": 429522,
      "northing": 4180270,
      "height": 790,
      "zone": "30S"
    },
    "northEastTop": {
      "easting": 430638,
      "northing": 4180363,
      "height": 820,
      "zone": "30S"
    }
  },
  "data_blockSize": 10000,
  "data_blockFormat": "LAZ",
  "rootDatablocks": [
    { "datasetCell": "4241830S",
"id": 0 },
```

```
    { "datasetCell" : "4341830S" ,
"id" : 0 }
  ]
}
```


#7	GET	/**workspaces**/{workspace_name} /**datasets**/{dataset_name} /**datablocks**/{datablock_id} ?**cell**={dataset_cell}	Recupera la información de un *datablock*. El parámetro *cell* no será necesario cuando el *dataset* solo tenga una celda asociada. Respuesta de ejemplo:


```
{
  "id" : 0,
  "datasetCell" : "4241830S" ,
  "bbox" : {
    "southWestBottom" : {
      "easting" : 429522,
      "northing" : 4180270,
      "height" : 790,
      "zone" : "30S"
    },
    "northEastTop" : {
      "easting" : 429999,
      "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** ?**cell**={dataset_cell}	Devuelve la información del fichero de nubes de puntos asociado al *datablock*. Respuesta en formato binario (application/octet-stream).
#9	PUT	/**workspaces**/{workspace_name} /**datasets**/{dataset_name} /**data**	Servicio para asignar una serie de ficheros de nubes de puntos a un *dataset*. Generará la estructura de *datablocks* asociada al *dataset*.
#10	GET	/**workspaces**/{workspace_name} /**datasets**/{dataset_name} /**data**	Devuelve la nube de puntos al completo.
#11	GET	/**workspaces**/{workspace_name} /**datasets**/{dataset_name} /**datablocks** ?**southWest**={sw_coord} &**northEast**={ne_coord}	Devuelve los *datablocks* de un *dataset* dentro de la caja envolvente definida.

Tabla2.1. Tabla con la definición de los servicios clave para una API REST a implementar por un repositorio de modelos de nube de puntos.

A continuación, se muestran dos diagramas de secuencia para ayudar a entender cómo combinar estos servicios para llevar a cabo flujos de trabajo completos. Estos recogen los dos procesos clave que soportaría una implementación basada en SPSLiDAR: la inserción de *datasets* y la posterior consulta.

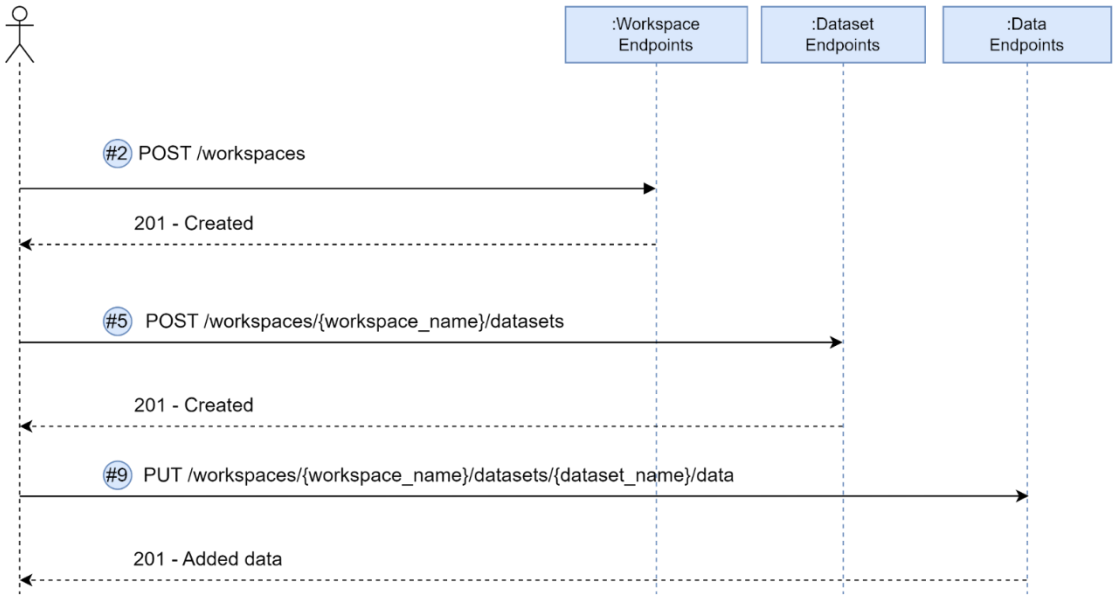


Figura2.6. Diagrama de secuencia con las operaciones necesarias para insertar un modelo de nube de punto

La Figura 2.6 define el proceso de inserción de datos. Para ello, se utilizan los métodos **#2** y **#5** para crear un *workspace* y un *dataset* respectivamente. En caso de que ya hubiese un *workspace* creado en el sistema en el que consideramos que el *dataset* pudiera tener cabida, podríamos omitir la llamada al método **#2**. Finalmente, con el método **#9** enviamos los ficheros de nubes de puntos que conformarán el *dataset* y que serán procesados en la estructura de *datablocks*.

En la Figura 2.7 se indican los pasos para realizar el descubrimiento de modelos de nube de puntos y la descarga iterativa de ficheros por parte de un cliente que carece de ningún conocimiento previo de los datos almacenados, y que, por lo tanto, necesita ir navegando por las distintas entidades del modelo de forma progresiva.

Con el método **#1** y **#3** se descubren los *workspaces* disponibles y se accede al detalle de uno de ellos. Con los métodos **#4** y **#6** se realiza el mismo procedimiento, pero esta vez sobre los *datasets* asociados a uno de los *workspaces* previamente recuperados. Con el método **#4** se pueden aplicar distintos criterios de filtrado adicionales a nivel espacial o temporal. Una vez encontrado el *dataset* deseado, el cliente puede iterar sobre la jerarquía de *datablocks* que tiene asociada, intercalando peticiones a los métodos **#7** y **#8** para recuperar los metadatos del *datablock* y descargar su fichero asociado respectivamente, Con los metadatos obtenidos en el **#7** se pueden buscar *datablocks* para los que mantenga la relación de parentesco y realizar una nueva llamada al mismo endpoint variando el campo *datablock_id*.

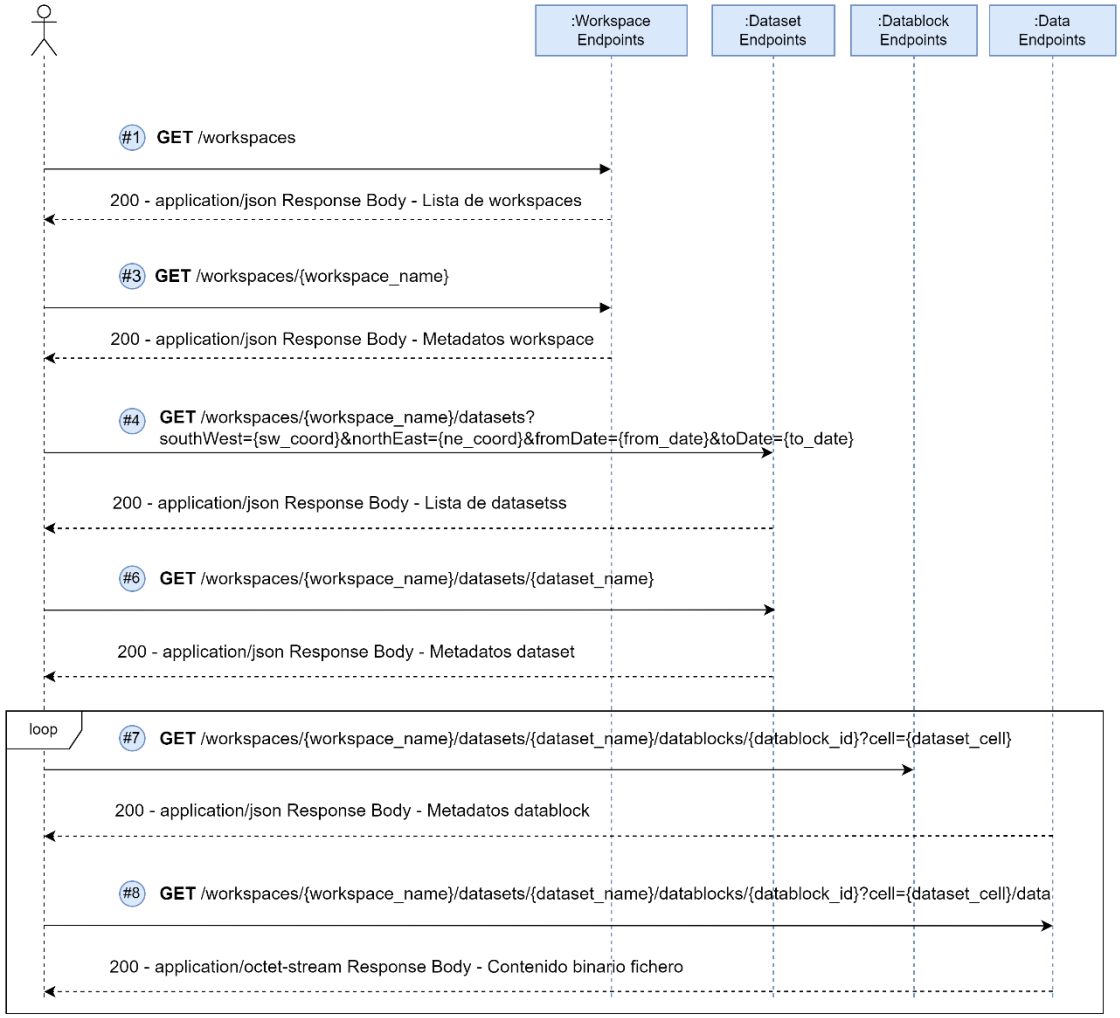


Figura2.7. Diagrama de secuencia con las operaciones para llevar a cabo una consulta de datos.

2.4. Extendiendo SPSLiDAR: SPSLiDAR 1.1

El modelo de SPSLiDAR original representa un marco general bajo el cual organizar información de modelos de nube de puntos a distintos niveles. Es un modelo sencillo y extensible, de manera que las entidades fundamentales puedan ser enriquecidas con nuevos atributos en función de cada caso de uso. A partir de la base de este modelo se ha definido SPSLiDAR 1.1, una extensión que recoge una serie de elementos adicionales que permiten a los clientes tener un mayor grado de decisión sobre cómo quieren almacenar y organizar la información, permitiendo aunar dentro de una misma implementación distintos tipos de configuraciones. Para ello, se han seguido dos líneas de trabajo principales, centradas en ámbitos concretos.

Por un lado, se proponen una serie de entidades con las que dar soporte a distintos sistemas de coordenadas. En el modelo original, el sistema de coordenadas es único dentro de una implementación: toda la información espacial recogida en la aplicación será tratada en dicho sistema, Además, este deberá ser un sistema basado en la proyección UTM. No es posible definir distintos sistemas de coordenadas ni llevar a cabo transformaciones entre ellos; si bien esta

filosofía nos aporta la ventaja de simplificar el procesamiento de este tipo de información geográfica, también supone una limitación para aquellos escenarios en los que sea necesario utilizar distintos sistemas de coordenadas.

El segundo ámbito en el que nos centraremos es la gestión de distintas estructuras de datos bajo una misma implementación. La versión base de SPSLiDAR permite definir ciertos parámetros con los que configurar distintas estructuras de datos. En el modelo presentado, utilizamos el campo *cellSize*, pero también recalcamos la posibilidad de eliminarlo o complementarlo con otros campos en función de la estructura a utilizar. Esta solución presenta inconvenientes, ya que requiere modificar las entidades para adaptarse a cada estructura. Otro aspecto relevante que presenta limitaciones con este modelo es el de poder ofrecer distintas estructuras con las que implementar las relaciones del modelo en un mismo sistema que lo implemente; no existe un mecanismo con el que poder elegir entre diversas alternativas ni con el que poder encapsular los campos que corresponden a la parametrización de cada una de ellas.

A lo largo de las siguientes secciones se presentarán una serie de entidades y servicios nuevos que complementan los fundamentos establecidos para poder resolver las dos problemáticas previamente mencionadas.

Soporte para distintos sistemas de coordenadas

Para permitir el uso de sistemas de coordenadas adicionales, debemos introducir en nuestro modelo nuevos atributos que permitan identificar cuál es el sistema utilizado. En la API propuesta en SPSLiDAR 1.0, existen dos tipos de operaciones que soportan la realización de consultas a nivel espacial (métodos **#4** y **#11**). Cuando se realiza una de estas consultas, es fundamental que estas se lleven a cabo sobre metadatos especificados en el mismo sistema de referencia, o de lo contrario, podrían dar lugar a inconsistencias. Para garantizar que las consultas se puedan realizar de manera eficiente, y teniendo en cuenta que la entidad de mayor nivel de abstracción que soporta este tipo de consultas es el *dataset*, hemos decidido definir el uso de un sistema de coordenadas de referencia a nivel de *workspace*. Esto implica que cuando se generan nuevas instancias de *datasets* y *datablocks*, sus cajas envolventes deberán de ser procesadas y transformadas al sistema establecido a nivel de *workspace*. En la implementación, será necesario desarrollar mecanismos que permitan llevar a cabo estas transformaciones, devolviendo un error cuando esta transformación no es posible.

Si bien las entidades del modelo necesitan estar especificadas en el mismo sistema de coordenadas, los ficheros asociados a los *datablocks* que contienen la nube de puntos mantendrán su sistema de coordenadas original para evitar introducir errores durante la conversión. La caja envolvente generada a partir de sus coordenadas será transformada al sistema del *workspace* al que pertenece, siendo el valor resultante el que quede almacenado en el campo que refleja esta información del *datablock*. Opcionalmente, es posible guardar como atributo adicional del *datablock* la caja envolvente original asociada al fichero.

CRS

Para modelar los sistemas de coordenadas utilizamos una entidad nueva denominada *CRS* (*Coordinate Reference System*, equivalente al concepto de SCR presentado en el **Capítulo 1**) con los siguientes atributos:

- ***format*** *(*String*)*: define el formato en el que se va a expresar el sistema de coordenadas de referencia. Una implementación podrá soportar distintos tipos de formato, como códigos EPSG, cadenas de texto basadas en OGC WKT/WKT2, proj4, etc.

- ***value*** (String)**:** constituye el sistema de coordenadas de referencia en sí, el valor especificado en el formato correspondiente. Por ejemplo, para un formato EPSG el valor asociado podría ser 4326. Describe de forma íntegra el sistema utilizado, tanto horizontal como verticalmente. En el caso de EPSG, si es necesario combinar dos códigos distintos para reflejar un sistema horizontal y uno vertical, se puede utilizar la sintaxis "4326+5714" (WGS84 para el horizontal y Mean-Sea Level para el vertical). La unidad de distancia utilizada será implícita al sistema utilizado.

- ***type*** (String)**:** sirve para especificar de forma explícita el tipo del sistema de coordenadas. Admite dos valores, GCS (Geographic Coordinate System) y PCS (Projected Coordinate System).

- ***forceProjectionTo*** (String): este campo solo se considerará cuando *type* sea de tipo GCS. Una de las problemáticas asociadas a los PCS, en el caso de la proyección UTM, es que quedan adscritos a una zona concreta. Si se desea trabajar con UTM (u otra proyección) a nivel global, utilizando la zona UTM más apropiada para cada *dataset*, se puede especificar un GCS en la definición del sistema de coordenadas e indicar en este campo la proyección a aplicar sobre este.

La entidad propone un mecanismo básico basado en información textual para referenciar un sistema. Por un lado, se define el formato: EPSG, WKT o proj4. Todas las implementaciones deberán dar soporte al menos a una de estas especificaciones e idealmente proveer las opciones disponibles a través de un servicio de la API. En el futuro se puede introducir soporte adicional para representar estos valores utilizando esquemas nativos en JSON, modelándolos como un objeto en vez de un String que encapsule el valor completo. Existen algunas especificaciones como PROJJSON para el estándar OGC WKT que pueden facilitar este proceso de integración.

Con los campos *type* y *forceProjectionTo* se busca mantener la compatibilidad con la versión base de SPSLiDAR. En esta, varios *datasets* pertenecientes a zonas UTM diferentes pueden convivir en el mismo *workspace*. El uso de un sistema de coordenadas proyectado adscrito a una zona UTM específica, como el representado por el código EPSG:32630 implica que toda la información en el *workspace* está asociado a ese sistema de coordenadas, el de la zona 30N. Es necesario definir un modelo bajo el cual se pueda utilizar la proyección UTM sin vincularla por defecto a una zona UTM, de manera que un *dataset* pueda estar en la zona UTM más adecuada para su situación geográfica. La Figura 2.8 muestra un ejemplo concreto de cómo distintas configuraciones de esta clase afectarían a la zona UTM seleccionada.

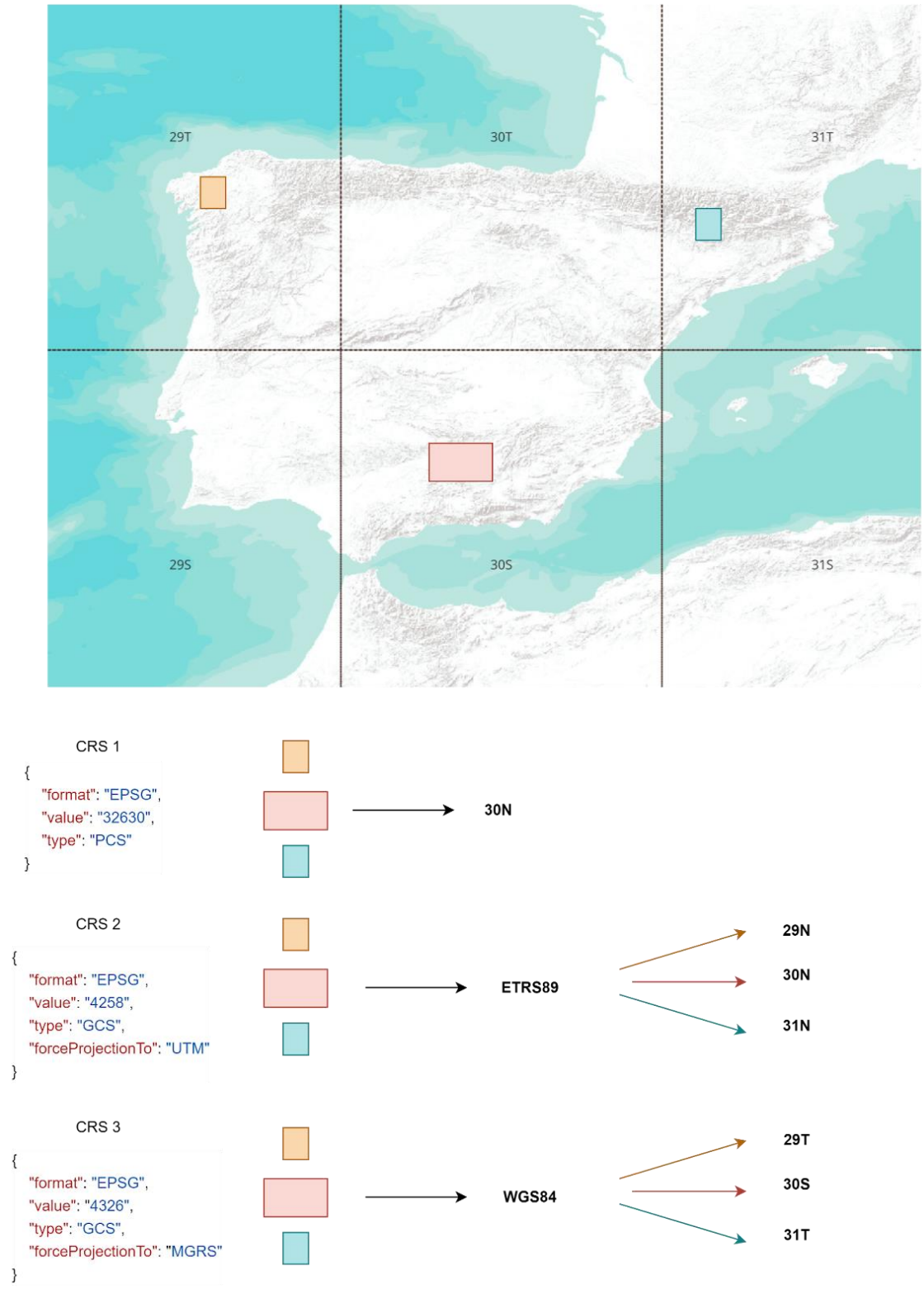


Figura2.8. Diagrama que muestra las zonas UTM a las que serían asociados un *dataset* en función del CRS utilizado.

En la Figura 2.9 se muestra la nueva entidad *CRS* y las relaciones establecidas con otras clases del modelo. La entidad *Workspace* mantendrá una instancia de la entidad *CRS* para establecer el sistema común. Definimos una entidad genérica llamada *Coordinate* y la entidad

UTMCoordinate definida en SPSLiDAR pasa a ser una extensión de esta que guarda la zona UTM asociada. Tanto *Coordinate* como *GeorefBox* pueden tener su propia instancia de la entidad *CRS*. En el caso de *Coordinate*, este será útil para aquellos casos en los que los clientes quieran definir una caja envolvente, por ejemplo, al crear un *dataset*, en la que las coordenadas estén representadas en sistemas distintos (como pudiera ser el caso al usar un *CRS* basado en la proyección UTM en la que cada coordenada estuviera en una zona diferente). Si se especifica una instancia de la entidad *CRS* a nivel de la entidad *GeorefBox*, este tendrá precedencia sobre los definidos a nivel de las coordenadas que lo componen.

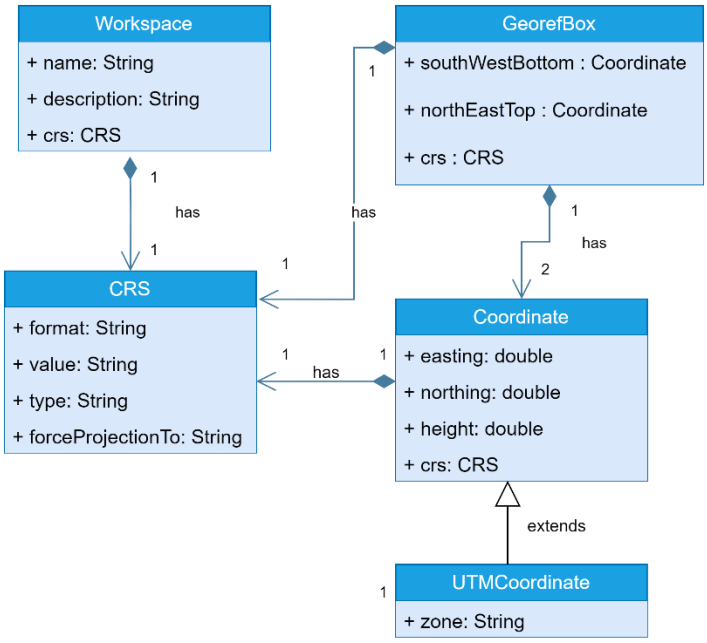


Figura2.9. Diagrama de clases con la nueva entidad CRS y las entidades que mantienen relación con ella

Soporte para múltiples estructuras de datos

En el modelo inicial de SPSLiDAR, cuando se añade un *dataset* o se genera una estructura de *datablocks*, la estructura de datos que modela cómo se organiza la información y se establecen vínculos entre las distintas entidades está prefijada a nivel de la implementación. El cliente tiene la posibilidad de especificar algunas variables que afectan a la parametrización de esta, pero no tiene control de decisión sobre la estructura utilizada. En muchos casos esto puede ser una solución suficiente: la interacción con la API es más sencilla para el cliente, la implementación es más concisa y se evita tener que responsabilizar a los usuarios sobre la toma de esta decisión, sobre la que quizás no tenga preferencias o un conocimiento exhaustivo de cómo puede afectar a sus datos.

Sin embargo, este enfoque puede no ser lo suficientemente potente para ciertos casos de uso. Una organización que quiera ofrecer servicios basados en SPSLiDAR puede requerir mantener dos formas diferentes de generar las estructuras de *datablocks* asociadas a un modelo de nube de puntos: por ejemplo, quadtrees para modelos aéreos de tipo 2.5D y octrees para aquellos con diferencias significativas en los tres ejes de coordenadas espaciales.

Con la especificación original, la toma de esta decisión no se puede llevar a cabo de forma nativa a través de la API. Es posible tener distintos backends, y delegar en algún otro componente de la arquitectura del sistema a cuál de estos redirigir una petición. Esto implicaría mantener dos bases de código independientes, diferenciadas únicamente por el detalle de la implementación de esta estructura. Esto presenta dificultades a la hora de mantener ambas implementaciones y la necesidad de aplicar cambios transversales en dos repositorios diferentes; la inclusión de nuevas estructuras de datos alternativas implica del mismo modo definir nuevas implementaciones, extendiendo aún más esta problemática.

También sería posible tener una única implementación y delegar en ella, en base a alguna lógica concreta o configuración externa sobre qué implementación utilizar: por ejemplo, si la diferencia entre los valores mínimo y máximo de la altura que abarca la nube no supera un umbral concreto, utiliza un quadtree, y en caso contrario un octree. Estas reglas pueden ser algo arbitrarias y siguen sin dar un control total al cliente a la hora de tomar la decisión.

Es por ello por lo que, en esta versión 1.1 del modelo, se define una capa de abstracción adicional, que permite un mayor grado de flexibilidad a la hora de decidir qué estructuras utilizar y cómo deben ser configuradas. Una implementación basada en este modelo puede exponer un catálogo de estructuras de datos disponibles, tanto a nivel de *Workspace-Dataset* como de *Dataset-Datablock*. Este catálogo contiene una descripción base de estas estructuras de datos, detalles de los parámetros requeridos que deberán proporcionar el cliente e información de los metadatos generados a nivel interno.

WorkspaceDataStructure/DatasetDataStructure

Para implementar esta extensión, hemos añadido dos entidades. Para las estructuras que modelan las relaciones entre *workspaces* y *datasets*, hemos definido la entidad *WorkspaceDataStructure*, que consta de los siguientes atributos:

- ***workspaceDataStructureId*** (Integer)**:** identificador numérico único que permite referenciar a esta estructura de datos de forma unívoca.
- ***workspaceDataStructureName*** (String): cadena de caracteres que contiene un identificador nominal de la estructura lo suficientemente representativo.
- ***workspaceDataStructureDescription*** (String): descripción de la estructura con información de utilidad sobre su funcionamiento y posibles casos de uso.
- ***parameters*** ([ ] *DataStructureParameter*): array de tipo *DataStructureParameter*. Indica los parámetros que caracterizan la estructura de datos, si los hubiera. Por ejemplo, en el caso de una estructura de jerárquica, el máximo número de niveles.
- ***cell*** (*WorkspaceDataStructureCell*): es un objeto que representa el tipo de nodo o celda definido por la estructura de datos, con sus correspondientes campos.

La clase *DataStructureParameter* representa un parámetro de configuración de esta estructura y define los siguientes campos:

- ***name*** (String): sirve como identificador nominal del parámetro.
- ***description*** (String): describe el propósito del parámetro.
- ***type*** (String): indica el tipo del parámetro, utilizando los tipos asociados a JSON como referencia.
- ***value***: define el valor que el cliente asigna al parámetro. Solo estará presente en los métodos en los que representa una instancia previamente creada. Su tipo dependerá del campo type.
- ***defaultValue***: indica el valor que le asignará la implementación por defecto a este parámetro en caso de que el cliente no lo especifique.

La clase *WorkspaceDataStructureCell* define los siguientes atributos:

- ***cellId*** (String): define el formato utilizado para dar valores a los identificadores de cada una de las celdas que componen las estructuras. Es un campo meramente informativo y que actúa a modo de pista sobre cómo se almacenará la información a nivel interno en el modelo.
- ***parameters*** ([ ] DataStructureParameter): array de tipo *DataStructureParameter*. Indica los campos que componen el nodo, si los hubiera. Un ejemplo sería el máximo número de puntos por nodo.

Para poder referenciar desde la entidad workspace a la estructura de datos que utilizará, se ha modificado su esquema para incluir el siguiente campo:

- ***dataStructure*** (WorkspaceDataStructure): indica la estructura de datos que se va a utilizar para organizar los *datasets* que queden asociados a este workspace.

Para las relaciones entre *datasets* y *datablocks*, hemos definido la entidad *DatasetDataStructure*, prácticamente idéntica a *WorkspaceDataStructure* pero con una nomenclatura ligeramente distinta en sus campos más adecuada para esta parte del modelo. La filosofía seguida es la misma, definir cómo se construyen y organizan en este caso los *datablocks* dentro de un *dataset*.

La Figura 2.10 ilustra la inclusión en el modelo de *WorkspaceDataStructure* y *DatasetDataStructure*, así como de otros ligeros cambios respecto a SPSLiDAR 1.0. En la entidad *Workspace*, se ha eliminado el campo *cellSize*. A partir de ahora, en las estructuras en las que fuese necesario, figura como una entrada más en la lista de parámetros que tiene asociada. Hemos aplicado el mismo concepto con el campo *datablockSize* de la entidad *Dataset*. Si bien este cambio tiene un carácter genérico, es posible que algún método requiera utilizar un parámetro diferente para delimitar el tamaño del *datablock* en lugar del volumen de puntos de la nube asociada (por ejemplo, definiendo cada *datablock* de manera que siempre corresponda un área de tamaño determinado, independientemente del número de puntos que estén presentes). Aun así, se recomienda su uso de forma general para poder establecer de forma clara y concisa el tamaño máximo esperado de los ficheros. Otro cambio que destacar es el cambio del tipo *id* de la entidad *Datablock*, que pasa a ser un String para soportar valores más complejos que no puedan ser representados con un tipo numérico.

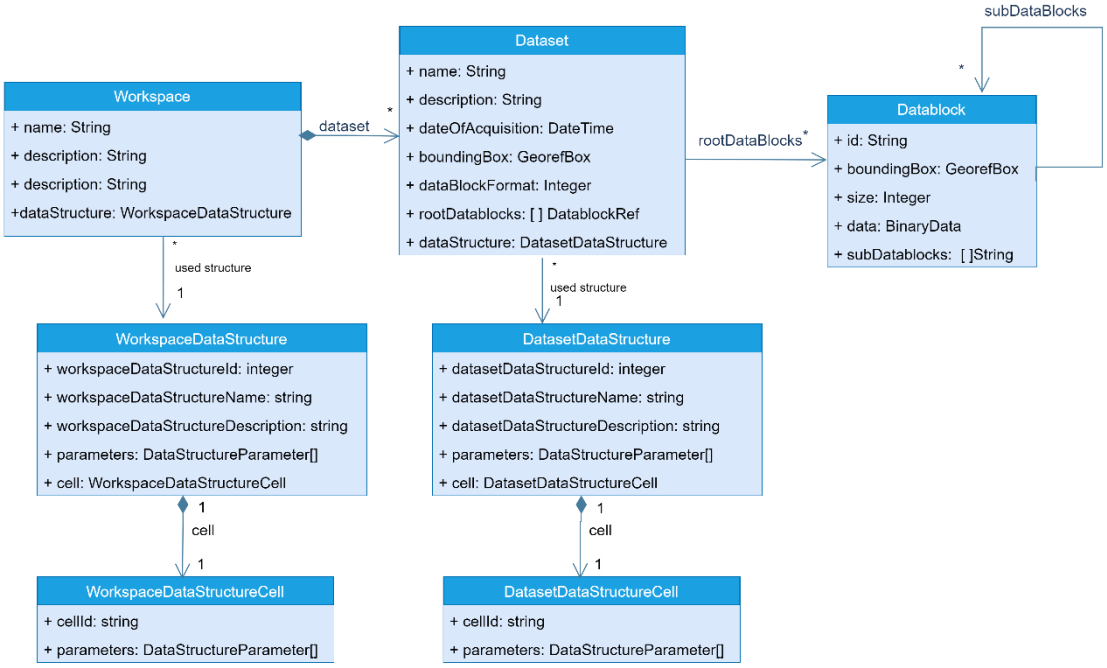


Figura2.10. Diagrama de clases con el soporte genérico de múltiples estructuras de datos.

API de SPSLiDAR 1.1

En la siguiente tabla figuran los nuevos servicios que se han incluido en la API. Los servicios **#12** y **#14** listan el catálogo de estructuras que ofrece la implementación para cada una de las relaciones. Para evitar cuerpos de respuesta de un tamaño excesivo, solo se muestran el identificador de la estructura y el nombre. Con los servicios **#13** y **#15** se puede recuperar el detalle completo.

ID	Verbo HTTP	URI	Descripción
#12	GET	**/metadata/workspaces/datastructures**	Devuelve una lista con las estructuras de datos disponibles a nivel de *Workspace-Dataset*. Respuesta de ejemplo: `[ { "workspaceDataStructureId" : 1, "workspaceDataStructureName" : "Grid disperso UTM de 2 niveles" } ]`

#13	GET	**/metadata/workspaces/datastructures**/{datastructure_id}	Devuelve la información completa de la estructura especificada. Respuesta de ejemplo**:**


```
{
    "workspaceDataStructureId" : 1,
    "workspaceDataStructureName" : "Grid disperso UTM de 2 niveles" ,
    "workspaceDataStructureDescription" : "Estructura de datos basada en un grid de dos niveles, utilizando el sistema de zonas UTM-MGRS en la primera capa y un grid disperso en la segunda." ,
    "parameters" : [
        {
            "name": "gridCellSize" ,
            "description" : "Define el tamaño de celda a utilizar en el segundo nivel de la estructura. Valor especificado en metros" ,
            "defaultValue" : 10000,
            "type" : "Integer"
        }
    ],
    "cell" : {
        "cellId" : "UTMZone - xCoordId - yCoordId" ,
        "parameters" : [
            {
                "name": "UTMZone",
                "description" : "ID de la zona UTM",
                "type" : "String"
            },
            {
                "name": "xCoordId" ,
                "description" : "ID de la celda en el eje X" ,
                "type" : "Integer"
            },
            {
                "name": "yCoordId" ,
                "description" : "ID de la celda en el eje Y" ,
                "type" : "Integer"
            }
        ]
    }
}
```


#14	GET	**/metadata/datasets/datastructures**	Devuelve una lista con las estructuras de datos disponibles a nivel de *Dataset-Datablock*. El esquema de la respuesta es similar al definido en el método **#12**.


```
[
    {
        "datasetDataStructureId" : 1,
        "datasetDataStructureName" : "Octree"
    }
```

```
]
```


#15	GET	**/metadata/datasets /datastructures** /{datastructure_id}	Devuelve los metadatos asociados a la estructura especificada. Respuesta de ejemplo:


```
{
      "datasetDataStructureId"       :  1,
      "datasetDataStructureName"     :  "Octree"  ,
      "datasetDataStructureDescription"          :
"Octree.    Cada nodo de la estructura
corresponde a un datablock."       ,
      "parameters"  :  [
            {
                  "name": "pointNumber"  ,
                  "description"     : "Define el
número de puntos máximo que ha de tener
cada nodo del octree. El criterio se
aplica en todos los niveles excepto el
último."   ,
                  "defaultValue"    : 10000,
                   "type"  :  "Integer"
            },
            {
                  "name":  "maxDepth" ,
                  "description"     : "Define el
nivel máximo de profundidad que puede
alcanzar la estructura."          ,
                  "defaultValue"    :  8,
                  "type"  :  "Integer"
            }
      ],
      "cell"   :  {
            "cellId"    :  "octreeId   -
workspaceCellId"   ,
            "parameters"  :  [
                  {
                        "name": "octreeId"    ,
                        "description"     :  "Define el
identificador numérico del nodo. "          ,
                        "type"  :  "Integer"
                  },
                  {
                        "name": "workspaceCellId"    ,
                        "description"     : "Correspond
e al cellId de la celda en la que está
encuadrado el octree."       ,
                        "type"  :  "Integer"
                  }
            ]
      }
}
```

#16	GET	**/crss**	Recupera una lista de los sistemas de coordenadas de referencia soportados por la implementación. Respuesta de ejemplo:
			
			[
			{
			"format" : "EPSG",
			"value" : "4326" ,
			"type" : "GCS"
			},
			{
			"format" : "EPSG",
			"value" : "4326" ,
			"type" : "GCS",
			"forceProjectionTo" : "UTM"
			}
			]

Tabla2.2. Descripción de los nuevos servicios definidos en SPSLiDAr 1.1.

A continuación se muestra un diagrama resumen con las distintas entidades del modelo y los servicios de la API asociados a ellas. Los servicios muestran únicamente el verbo HTTP que utilizan y su URI, omitiéndose detalles como los parámetros de consulta de la URL o el cuerpo de la petición. En la Figura 2.11 se muestran las entidades definidas en SPSLiDAR 1.0 y en la Figura 2.12 las nuevas entidades definidas en SPSLiDAR 1.1.

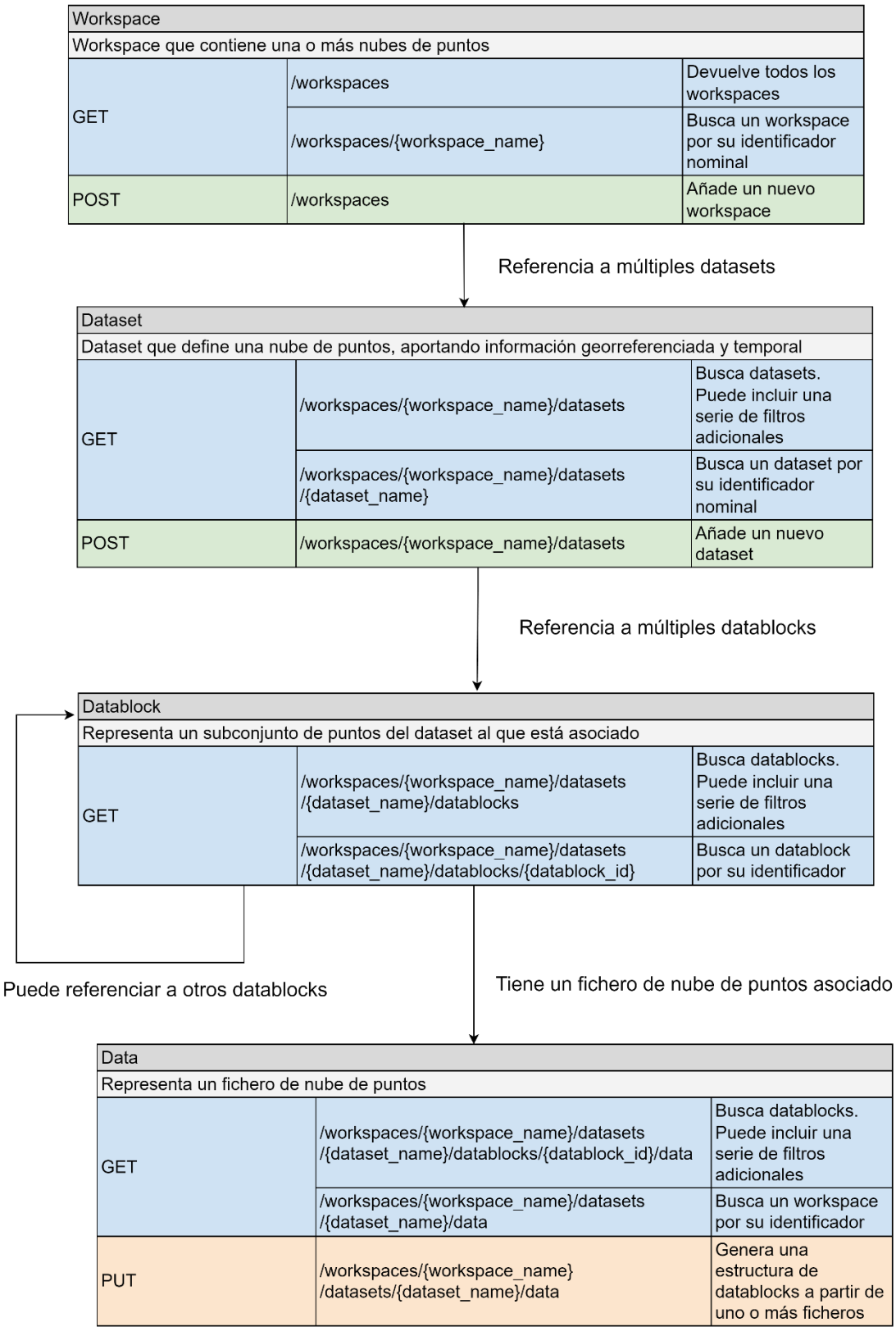


Figura2.11. Entidades y servicios asociados de SPSLiDAR 1.0.

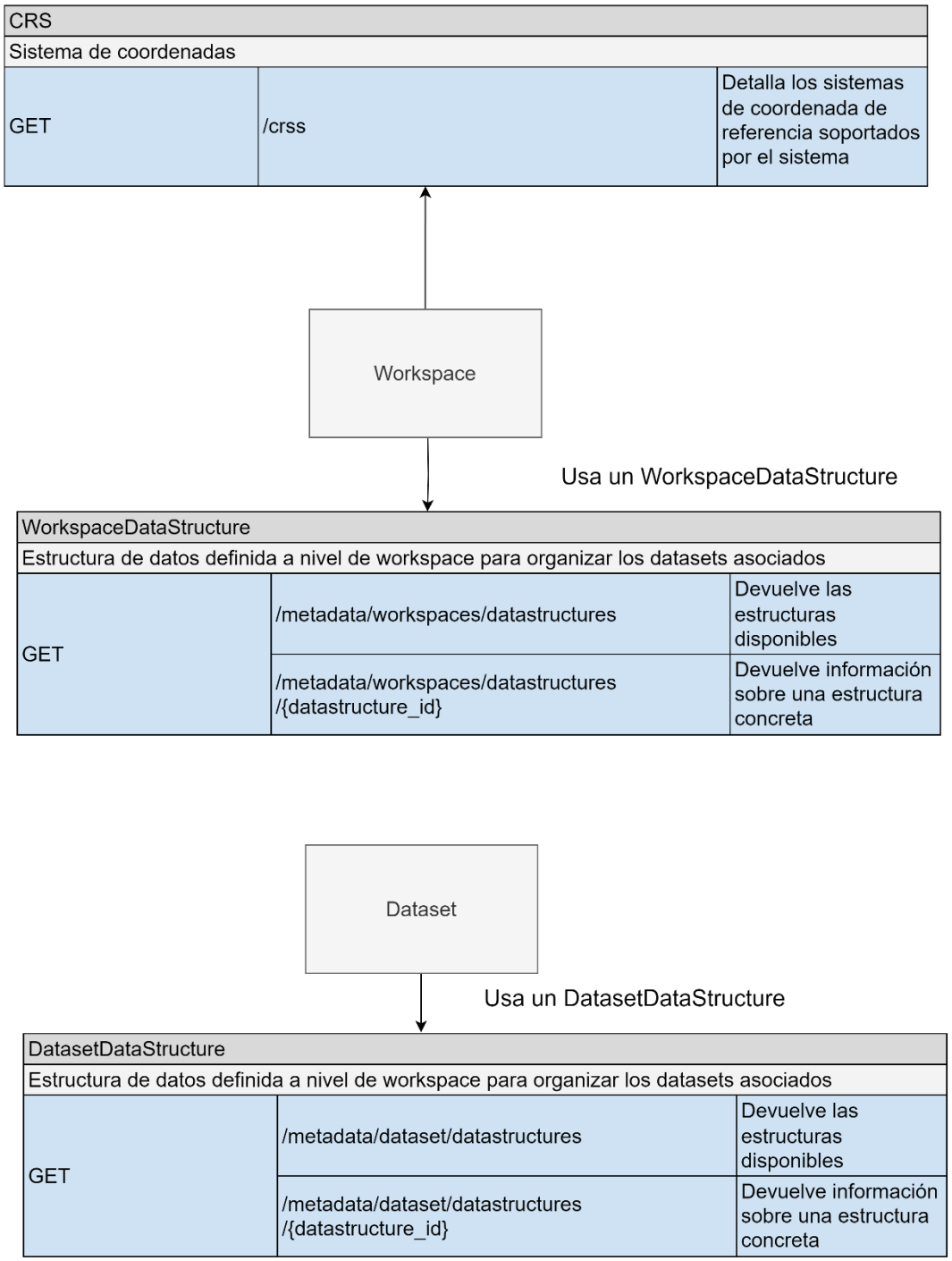


Figura2.12. Entidades y servicios asociados de SPSLiDAR 1.1.

2.5. SPSLiDAR: casos de estudio

SPSLiDAR pretende definir un modelo flexible y capaz de adaptarse a distintos contextos y requisitos. En esta sección se plantean dos casos de estudio con los que ilustrar estas características de SPSLiDAR. Ante distintos tipos de datos y patrones de acceso a la información, el modelo y API presentados en este capítulo proveen un mecanismo estandarizado y unificado para acceder a la información.

Ambos casos pueden ser integrados tanto bajo la especificación 1.0 como 1.1, aunque en el primer caso se trataría de dos implementaciones distintas del mismo modelo mientras que en el segundo existiría una única implementación con capacidad para adaptarse a distintos escenarios.

Preservación y restauración del patrimonio

Dentro de los muchos casos de uso de la tecnología LiDAR, uno de los más relevantes en el ámbito de las denominadas humanidades digitales es su aplicación en la preservación y restauración del patrimonio. En muchos casos, los sitios arqueológicos o históricos suelen ser frágiles y susceptibles a ser dañados por los efectos de los propios fenómenos naturales, siendo por lo tanto necesario llevar a cabo múltiples escaneos de forma regular que permitan detectar posibles daños. Durante una intervención de restauración que se prolongue en el tiempo, será necesario generar múltiples representaciones de la zona afectada, con el objetivo de controlar su evolución.

A la hora de decidir un tipo de organización para el *workspace*, es preciso tener en cuenta que va a existir un amplio número de *datasets* centrados en torno a ubicaciones muy específicas. Sobre estas localizaciones, se llevarán a cabo escaneos de forma regular, con el fin de documentar procesos de restauración y analizar posibles desperfectos. Un tipo de consulta que tendría especial relevancia en estos casos sería la llevada a cabo en base a criterios temporales, con vistas a recuperar modelos adquiridos en distintos momentos y llevar análisis comparativos sobre la evolución de estos conjuntos históricos.

En función del número de casos que se quieran almacenar y su distribución en el espacio, existen varias alternativas. Por ejemplo, es posible que un *workspace* almacene distintas localizaciones de la provincia de Jaén. Se puede utilizar en este caso una estructura de tipo grid que permita indexar en base a tres dimensiones: dos para las componentes espaciales con el fin de optimizar las consultas de ventana sobre la superficie terrestre y una tercera que refleje la línea temporal. En el caso de un *workspace* destinado a almacenar únicamente patrimonio de la ciudad de Jaén, sería posible descartar la componente espacial de este grid y definir una estructura unidimensional basada únicamente en la línea temporal; debido a la ubicación geográfica de estos *datasets*, es muy probable que queden indexados en una misma celda si se sigue utilizando una estructura de carácter espacial. Por lo tanto, las ventajas de esta solución serían mínimas y añadirían una complejidad adicional a las consultas; se puede eliminar directamente el indexado en dichas dimensiones, descartando el concepto de grid espacial y empleando únicamente en la dimensión temporal para agrupar *datasets*. También cabe la posibilidad de que el número de modelos obtenidos de estos conjuntos históricos a lo largo del tiempo sea reducido y, por lo tanto,

el uso de una simple lista para registrar los *datasets* de un *workspace*, implementando las consultas mediante búsqueda exhaustiva, podría ser suficiente. Esto se corresponde con la mayoría de escenarios, en los que lo normal es contar con poco más de una decena de versiones de un mismo modelo.

Elaborando más el segundo escenario descrito anteriormente, partiendo de la hipótesis de que se están llevando a cabo adquisiciones de nubes de puntos de una serie de monumentos de la localidad de Jaén con una frecuencia elevada, vamos a plantear el uso de una estructura de grid unidimensional indexado únicamente en el ámbito temporal, dividiendo en intervalos regulares y utilizando como origen un instante temporal predeterminado. De este modo, los *datasets* próximos en el tiempo, y por tanto dentro de la misma ventana temporal, quedarán asociados al mismo nodo, o a nodos vecinos en el peor de los casos. Las consultas de ventana temporal se realizarían usando esta estructura, lo que aseguraría que el número de nodos a leer es mucho menor que el de *datasets*, mejorando el rendimiento de una consulta de fuerza bruta sobre todos los *datasets* del *workspace*. Se utilizaría el campo *dateOfAcquisition* para definir a qué celda estaría asociado cada *dataset*.

A un nivel interno, los *datablocks* de un *dataset* podrían ser organizados en torno a una estructura jerárquica. El uso de un octree puede ser una opción interesante al tratar con edificios que se expanden de forma consistente en las tres dimensiones, con diferencias significativas en todas ellas. En cambio, en el caso de un *workspace* centrado en la gestión de yacimientos arqueológicos, un quadtree podría ser suficiente.

En la Figura 2.13 vemos como un *workspace* denominado #wsA indexa varios *datasets* que están asociados a una celda que representa una ventana temporal de un mes. Cada uno de estos *datasets* tienen un octree asociado que contiene un modelo de puntos del edificio ilustrado en la imagen. Puesto que a la hora de buscar un *dataset*, la ubicación de este edificio siempre será la misma, la consulta por ventana temporal será la herramienta principal de filtrado, quedando optimizada por la estructura *Workspace-Dataset* que indexa a nivel temporal. Una vez seleccionado el *dataset*, el octree nos proporcionará una estructura de niveles de detalle óptima para la descarga progresiva y/o visualización, ya que la exploración de la nube sí que se realizará en base a criterios espaciales.

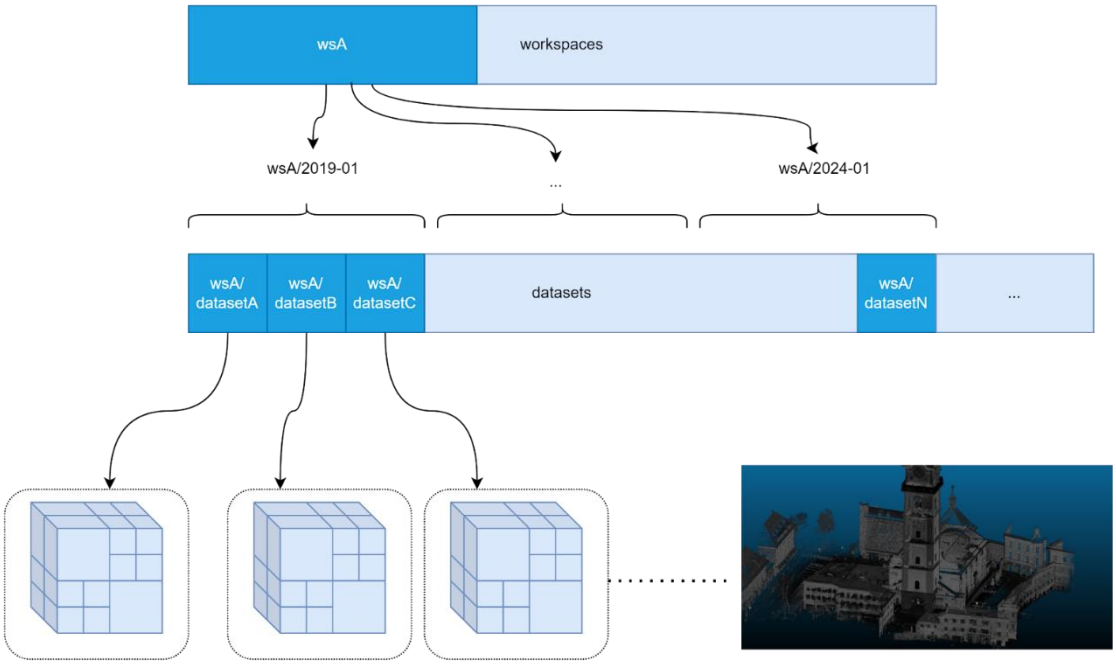


Figura2.13. Ejemplo decombinación de EEDDs en el modelo SPSLiDAR para construir un repositorio para almacenar modelos de nube de puntos asociadas a sitios históricos

Infraestructuras civiles

Las nubes de puntos pueden ser un recurso muy útil de cara al mantenimiento y construcción de infraestructuras civiles, como por ejemplo carreteras o redes ferroviarias. Las representaciones del terreno obtenidas mediante escaneos aéreos permiten generar documentación topográfica con la que analizar y planear la construcción de nuevas infraestructuras. También, permiten llevar análisis sobre tramos ya existentes de cara al mantenimiento de estos y la detección temprana de posibles daños.

En el caso de carreteras y redes ferroviarias hablamos de trazados complejos compuestos por miles de kilómetros. Dependiendo del alcance del repositorio, podrían registrarse modelos distribuidos, no solamente por la geografía nacional, sino incluso a nivel continental o global. Al mismo tiempo, su distribución suele ser muy irregular y dispersa, siguiendo el recorrido lineal de la vía y las zonas adyacentes. La adquisición de modelos puede venir guiada por dos aspectos: el desarrollo y construcción de nuevos tramos, que implicará probablemente una mayor frecuencia de escaneos en la zona, y la monitorización de las zonas ya existentes con el fin de vigilar su evolución y detectar posibles daños, lo cual tendrá una frecuencia en el tiempo menor.

Partiendo de estas premisas, una estructura en forma de grid disperso puede ser una opción adecuada para modelar la relación entre *workspaces* y *datasets*. Planteamos el uso de un grid compuesto por dos capas, una primera basada en zonas UTM y una segunda dentro de cada zona con información que contendrá celdas de un tamaño menor generadas de forma dinámica en base a la información registrada. Debido al carácter global de estos *datasets*, las zonas UTM definen un primer nivel de filtrado, mientras que el grid disperso del segundo nivel permite indexar con mayor precisión los *datasets* y resolver consultas espaciales de forma más eficiente.

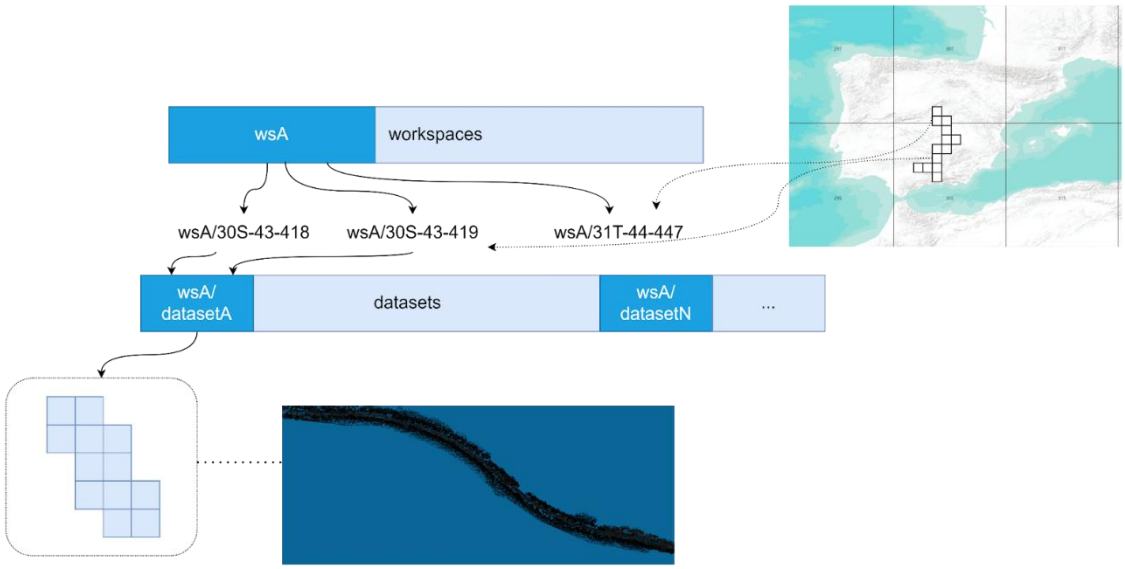


Figura2.14. Ejemplo de combinación de EEDDs en el modelo SPSLiDAR para construir un repositorio para almacenar modelos de nube de puntos asociadas a modelos de la red ferroviaria.

Las características mencionadas anteriormente también influirán a nivel de *datablock*. A diferencia de en el escenario anterior, los escaneos adquiridos poseerán una perspectiva más cercana al 2.5D, por lo que el uso de octrees aporta menos ventajas con respecto al caso de uso anterior. Hay varias opciones que podrían ser interesantes: quadtrees (aunque su adaptabilidad a modelos lineales no sea la más óptima como los vías ferroviarias o carreteras), grids dispersos, R-trees, etc. Para llevar a cabo tareas de renderizado progresivo, el uso de alguna de las estructuras jerárquicas puede ser interesante, mientras que si se quiere priorizar las consultas de ventana extensas con las que recuperar múltiples *datablocks*, el grid podría ser una opción más interesante por su sencillez. La Figura 2.14 muestra un ejemplo concreto de estos casos utilizando grids como estructura de datos en ambos niveles del modelo.

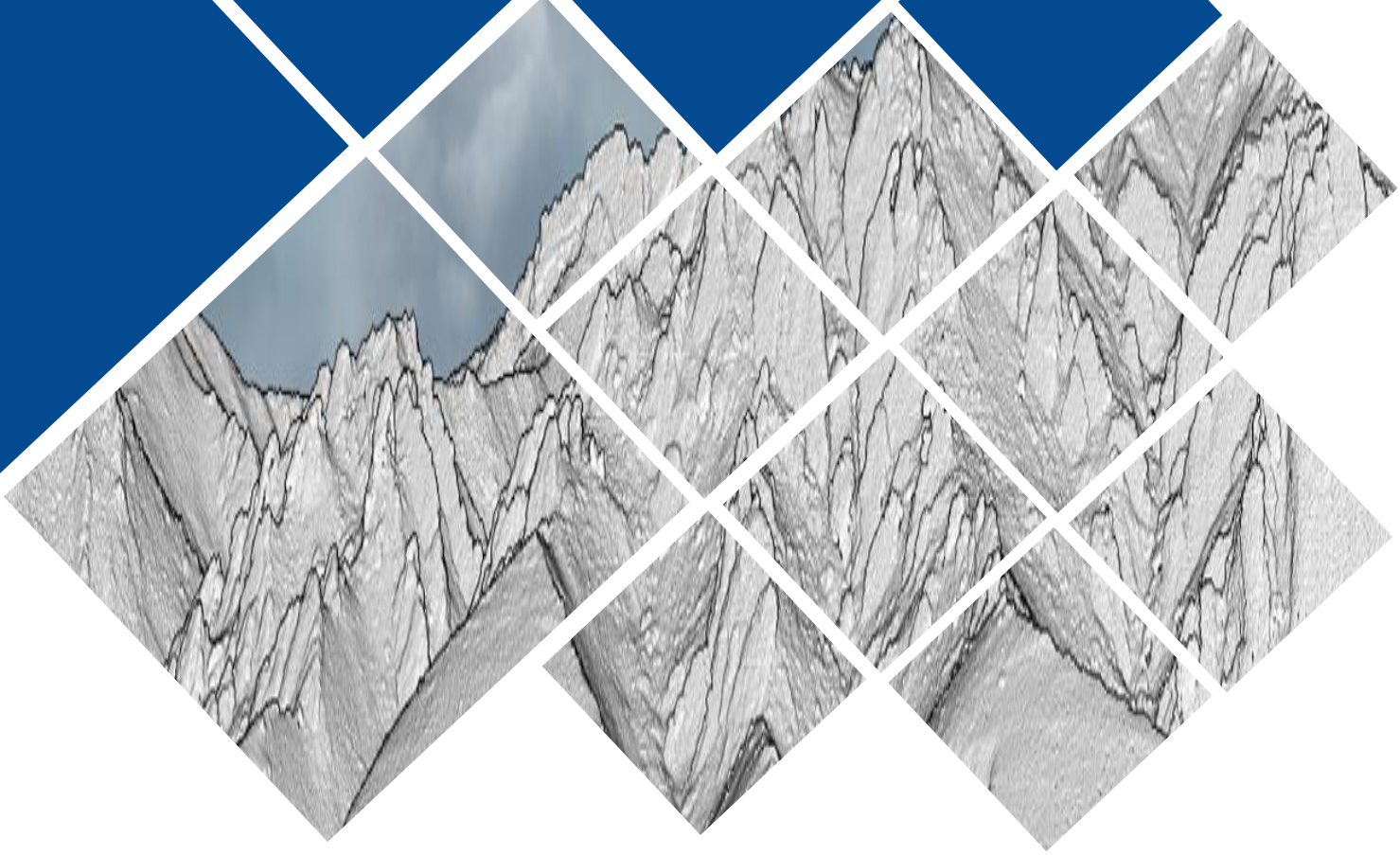
2.6. Conclusiones

La necesidad de definir soluciones para la gestión integral de grandes colecciones de modelos de nubes de puntos es cada vez más apremiante ante la enorme utilidad que han demostrado estos modelos en multitud de disciplinas como la ingeniería civil, arquitectura, arqueología o gestión forestal y medioambiental, entre otras. El crecimiento exponencial del número y tamaño de los modelos manejados requiere soluciones innovadoras para su almacenamiento, catalogación, organización y acceso desde las aplicaciones encargadas de su análisis y procesamiento.

SPSLiDAR define un modelo conceptual con este fin, capaz de organizar en múltiples niveles este tipo de información. Es un modelo simple y extensible, que puede ser adaptado por las implementaciones para adecuarse a sus necesidades, ya sea mediante la inclusión de nuevos atributos o utilizando diferentes estructuras de datos para modelar las relaciones entre las entidades del modelo.

Junto al modelo, hemos presentado una API REST para a la implementación de SPSLiDAR como repositorio o servicio de provisión de nubes de puntos. En base a las necesidades identificadas en la industria, el uso de estas soluciones permite la centralización de grandes volúmenes de información en un único sistema, de manera que las aplicaciones cliente puedan acceder de forma ubicua a los datos que requieran. Esta API define distintos servicios con los que interactuar con el modelo, facilitando principalmente operaciones para consultar, recuperar e insertar información. Los servicios expuestos permiten descubrir y localizar la información requerida, ya sea accediendo mediante identificadores propios para cada entidad o a través de consultas espacio-temporales. También define métodos con los cuáles es posible definir nuevos *datasets* y organizar sus puntos en distintos tipos de estructuras.

Si bien la solución definida define un estándar básico con el que dar respuesta a los problemas planteados, también puede ser mejorado de múltiples formas. En SPSLiDAR 1.1 hemos presentado algunas mejoras con las que permitir dar cabida a distintos sistemas de coordenadas dentro de una misma implementación. También define un sistema con el que ofrecer un catálogo de distintas estructuras de datos elegibles en el momento de la importación de un *dataset*, de forma que pueda escogerse la organización espacial más adecuada en función de sus características. Aun así existen múltiples líneas de mejora, que abarcan desde la definición de nuevas geometrías para las consultas espaciales, nuevos criterios de consulta, exposición de representaciones alternativas de los datos, así como nuevos servicios para otros casos concretos como la edición de nubes de puntos. Si bien las posibilidades son amplias, consideramos que SPSLiDAR define una base robusta y flexible sobre la que trabajar.



Capítulo 3 - IMPLEMENTACIÓN DE SPSLIDAR EN UN REPOSITORIO PARA NUBES DE PUNTOS

3.1. Introducción

En el **Capítulo 2** se ha descrito el modelo SPSLiDAR 1.0 y su extensión SPSLiDAR 1.1 desde un punto de vista conceptual y general, describiendo sus entidades, relaciones y los servicios para la navegación por el modelo. En este capítulo se abordan aspectos concretos de implementación del modelo SPSLiDAR, describiendo un repositorio que implementa este modelo y expone la API propuesta, planteando distintos tipos de estructuras de datos posibles y recogiendo detalles a un nivel más técnico que el que hemos considerado hasta ahora.

Debe quedar claro que SPSLiDAR es un marco amplio que admite muchas implementaciones posibles. En la implementación de referencia de SPSLiDAR (Rueda-Ruiz et al., 2022), se utilizó una organización espacial clásica para modelos de nube de puntos basada en octrees que incorpora niveles de detalle. En este capítulo complementaremos esta estructura con otras alternativas. Todas ellas pueden ser implementadas bajo las dos versiones del modelo propuestas.

3.2. Tecnologías de la implementación

Existen múltiples opciones a la hora de implementar un sistema de estas características. La implementación de referencia se ha llevado a cabo utilizando Java, principalmente por su popularidad en el ámbito del desarrollo a nivel backend. Hemos utilizado el framework Spring debido a su madurez, calidad contrastada y extenso catálogo de funcionalidades que incorporan aspectos como la configuración del servidor, la conexión con la mayoría de sistemas de gestión de bases de datos relacionales y NoSQL, y la implementación sencilla de APIs REST.

Además, se ha utilizado la variante reactiva de este framework, Spring Webflux, basado en la librería Reactor. La justificación del uso de un stack no bloqueante viene dada por varios motivos. Ante el almacenamiento masivo de datos, es lógico pensar que el acceso por parte de los clientes será frecuente y concurrente. En un modelo tradicional de un hilo por petición, la escalabilidad solo es posible incrementando el número de hilos expuestos por el servidor web, y en última instancia, incrementando el número de nuevas instancias de la aplicación en distintos nodos. Cuando se analiza la ejecución de cualquier servicio destaca la existencia de un número alto de operaciones de tipo E/S: por un lado, accesos constantes a base de datos para recuperar la información y por otro, lecturas y escrituras en disco para procesar ficheros de nubes de puntos, ya sea de forma directa o mediante una aplicación externa, que como se verá más adelante es nuestro caso. Bajo un paradigma bloqueante, el tiempo transcurrido en estas operaciones es tiempo perdido para el hilo, que queda bloqueado hasta que concluye la operación. Sin embargo, bajo el paradigma reactivo, el hilo queda liberado y puede atender mientras tanto otras peticiones, haciendo un uso más eficiente de los recursos del sistema. Cuando una operación E/S concluye, se notifica a la aplicación y un hilo retoma el flujo de ejecución.

Este elevado número de operaciones de E/S hace que la elección de la tecnología de almacenamiento sea un factor crítico en la implementación. Muchas estrategias son posibles: desde simples ficheros organizados mediante algún tipo de índice en una base de datos hasta la plena integración de toda la información en bases de datos relacionales o NoSQL. Para ayudar en la toma de decisiones realizamos un estudio comparativo (Béjar-Martos et al., 2022) que se detalla

en el **Capítulo 4** de esta memoria. En esta implementación de referencia original elegimos el sistema de gestión de base de datos MongoDB por las siguientes razones:

- Ofrece un mejor rendimiento en comparación con las bases de datos relacionales a la hora de llevar a cabo consultas por clave-valor o consultas geoespaciales (Agarwal & Rajan, 2017; Rachit Pandey, 2020). Otras bases de datos NoSQL similares como Cassandra ofrecen un rendimiento incluso superior (Deibe et al., 2018) pero su integración con Spring Webflux tiene un menor grado de madurez con respecto a MongoDB. Además, no tiene soporte directo para almacenamiento de grandes ficheros, lo que sí nos permite la tecnología GridFS (aunque se podría implementar una solución propia siguiendo una estrategia similar).

- Soporta escalado horizontal mediante sharding. Utilizando una clave y estrategia adecuada podemos distribuir de forma eficiente el volumen de datos entre distintos nodos manteniendo tiempos de respuesta satisfactorios. Del mismo modo podemos incluir nuevos nodos para incrementar la capacidad de almacenamiento del sistema.

- Permite almacenar ficheros de gran tamaño utilizando la especificación GridFS. Gracias a esto podemos almacenar los ficheros de puntos de forma sencilla en la propia base de datos, aprovechando las ventajas de MongoDB a la hora de garantizar la correcta distribución de los datos, prevención de inconsistencias entre metadatos y ficheros y transaccionalidad.

- Excelente documentación y evolución continua. MongoDB es uno de los sistemas de gestión de bases de datos más populares, dispone de una documentación y soporte excelentes, y sigue en continua evolución incorporando nuevas características.

Integración de ficheros de nubes de puntos

La implementación de referencia desarrollada se centra en el uso de los estándares LAS y su versión comprimida, LAZ. Es necesario por lo tanto tratar el procesamiento de estos formatos en el sistema, de manera que se puedan hacer transformaciones sobre estos ficheros para adecuarlos a las estructuras definidas y extraer de ellos la información para construir las distintas instancias de la entidad *Datablock*.

Dentro del ecosistema de Java, no existen actualmente librerías maduras para trabajar sobre ficheros de nubes de puntos basados en LAS/LAZ. La más popular es laszip4j (Reutegger, 2023), un port en Java de la popular librería Laszip (Isenburg, 2013). Recientemente ha introducido soporte para escribir ficheros, uno de los principales inconvenientes detectados durante el momento en el que se desarrolló esta implementación de SPSLiDAR, lo que la hace una solución a estudiar por parte de implementaciones futuras basadas en Java, aunque las funcionalidades que ofrece a la hora de manipular y transformar estos ficheros siguen siendo bastante limitadas. También existe la posibilidad de llevar a cabo una integración mediante JNI con algunas de las herramientas más populares en materia de procesamiento de nubes de puntos, como PDAL y LasTools.

En esta implementación se ha utilizado una integración con LasTools, aunque utilizando la invocación de procesos externos desde la JVM. LasTools es un conjunto de herramientas muy completo, robusto y eficiente que soporta múltiples operaciones y transformaciones (muestreo, filtrado por distintos atributos, transformaciones entre formatos y sistemas de coordenadas, etc.). De esta forma se pueden realizar las operaciones de bajo nivel, como la lectura, particionado y creación de los ficheros asociados a cada bloque mediante invocación de las distintas utilidades de LasTools (las2las, laszip, etc.). Ciertamente, una implementación de calidad integrada en una biblioteca Java proporcionaría un rendimiento mejor, puesto que la sucesiva invocación de procesos externos y las lecturas y escrituras repetidas de ficheros con resultados intermedios, necesarias al no poder guardar los resultados temporales en memoria, tienen un coste que puede igualar al del propio procesamiento. Aun así, hay que relativizar la importancia del tiempo de importación puesto que como es evidente, se trata de una operación que se realiza una única vez por *dataset*. Sólo en aplicaciones muy específicas, que tengan un flujo continuo de entrada de *datasets* al repositorio sería relevante y justificaría abordar esta operación con un desarrollo propio.

3.3. Arquitectura de la aplicación

La aplicación desarrollada sigue una filosofía basada en tres capas, un enfoque habitual dentro del ecosistema Java que facilita la separación de responsabilidades dentro del sistema y el desacoplamiento entre los distintos componentes de este. Los clientes pueden comunicarse con la aplicación utilizando el protocolo HTTP/HTTPS, utilizando los endpoints definidos en la API propuesta en el **Capítulo 2**. La primera de las capas normalmente se denomina de presentación, y se encarga de enrutar las peticiones recibidas a los servicios correspondientes, procesar la información recibida (transformándola a objetos definidos dentro del modelo de la aplicación) y construir la respuesta final que se devolverá al cliente. Esta capa encapsula varios controladores, que agrupan los servicios necesarios para interactuar con las distintas entidades del modelo.

Desde la capa de presentación las peticiones se redirigen a la capa de servicios, en la que se aplica la lógica necesaria para su resolución. La implementación de la mayoría de servicios es relativamente sencilla, requiriendo principalmente la interacción con la capa de persistencia para satisfacer una o varias consultas con las que construir la información final que deberá devolver al cliente. La complejidad se incrementa en los servicios encargados de construir las estructuras de datos. Este es el principal aspecto que se aborda en este capítulo, analizando la integración de varias estructuras de datos en el sistema y los algoritmos utilizados para su construcción. En el caso de aquellas estructuras a nivel de *Dataset-Datablock* la complejidad se ve acentuada por la necesidad de orquestar un proceso de composición de la estructura que requiere el encadenamiento de múltiples tareas de procesamiento sobre ficheros a través de LasTools. La creación de estas estructuras también requiere el almacenamiento de ficheros temporales, ya sea para almacenar la información recibida desde el cliente como para construir ficheros intermedios necesarios en el flujo de las operaciones realizadas con LasTools.

Finalmente, la capa de infraestructura se encarga de interactuar con los componentes externos que utiliza el sistema. Se define la integración por un lado con el sistema de base de datos, MongoDB en esta implementación de referencia, utilizando el protocolo MongoDB Wire para la comunicación. Incluye la lógica necesaria para implementar el almacenamiento de

información y define el esquema de los documentos asociados a cada una de las entidades del modelo. En esta capa se incluyen también las clases para la interacción con LasTools y el sistema de archivos. Es fundamental que la comunicación entre la capa de servicios y esta capa de infraestructura se haga a través de interfaces, con el fin de evitar el acoplamiento con estas tecnologías. Esto nos permitirá, como veremos en el **Capítulo 4**, llevar a cabo una integración con otros sistemas de bases de datos alternativos de manera relativamente sencilla.

La Figura 3.1 muestra las distintas capas y la comunicación con los distintos elementos externos a la aplicación. Los clientes se comunican a través de servicios RESTful usando el protocolo HTTP con la capa de presentación que delega en la capa de servicios la aplicación de la lógica necesaria; esta a su vez interactúa con tres servicios de infraestructura: las herramientas LasTools para llevar a cabo las operativas sobre ficheros necesarias utilizando procesos a nivel de sistema operativo, el sistema de archivos para almacenar ficheros con carácter temporal y la capa de persistencia para recuperar o almacenar la información final.

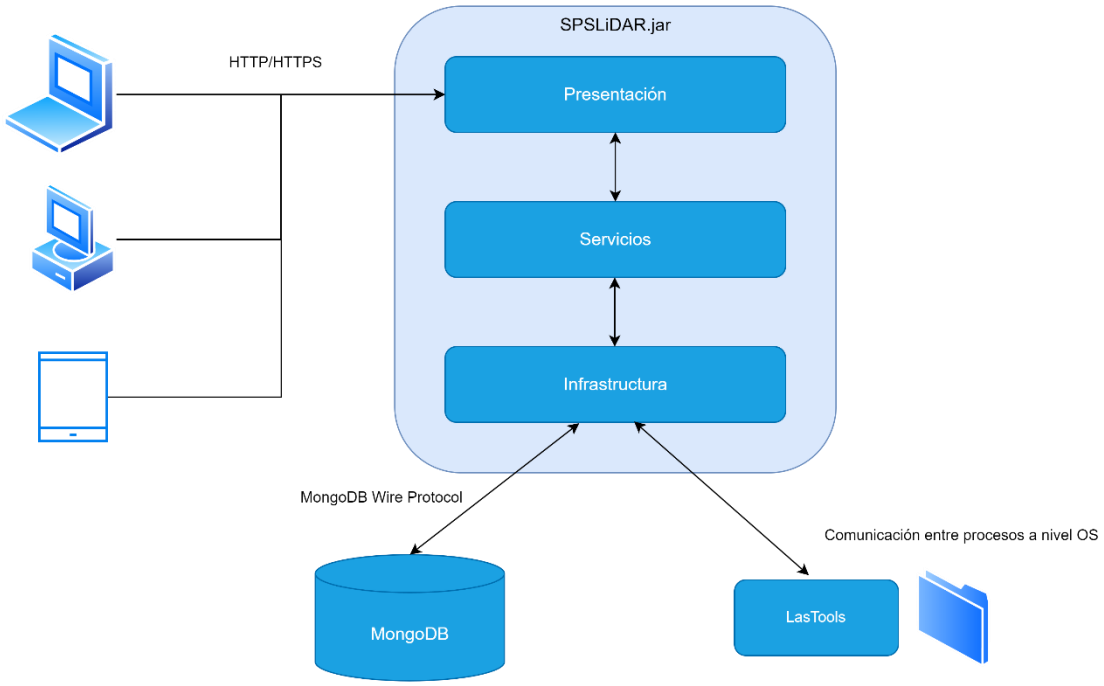


Figura3.1. Diagrama de la aplicación SPSLiDAR implementada.

3.4. Estructuras de datos

Uno de los aspectos críticos a resolver dentro de la implementación desarrollada es el soporte para distintas estructuras de datos. La implementación de los algoritmos utilizados dentro del marco de trabajo propuesto por las tecnologías seleccionadas y la adecuación de los datos generados a la especificación SPSLiDAR serán los puntos principales que tratar en las próximas secciones de este capítulo. Abordaremos el desarrollo de distintas estructuras, adecuadas a diferentes escenarios, tanto a nivel de la indexación de *datasets* como de *datablocks*. En este último caso se detalla, para las estructuras definidas, el proceso llevado a cabo para transformar

los ficheros de nubes de puntos recibidos en colecciones de *datablocks* y ficheros adecuados al modelo establecido. Las estructuras abordadas pueden ser implementadas tanto en la versión original del modelo como en la versión 1.1. Al final del capítulo se han incluido los esquemas JSON correspondientes a las distintas estructuras presentadas, utilizando las nuevas entidades presentadas en esta versión 1.1 de la especificación.

Estructuras de datos a nivel dataset

Existen múltiples formas de implementar la relación entre *workspaces* y *datasets*, en la que factores como los patrones de búsqueda o el número de *datasets* tendrán un impacto clave a la hora de decidir la opción más adecuada. La elección de esta estructura tendrá repercusiones en cómo se organizan y dividen los puntos del propio modelo; cada nodo de esta estructura que indexe al *dataset* servirá también para realizar un particionamiento preliminar sobre la nube de puntos que se le asocie, generando una instancia de la estructura *Dataset-Datablock* por cada nodo.

Grid disperso UTM de 2 niveles

Una rejilla o grid es una estructura de datos en la que cada nodo representa una región del espacio. Es una estructura habitual a la hora de organizar información espacial, debido a su sencillez y efectividad a la hora de indexar información en base a estos criterios. Esta estructura forma parte de la implementación de referencia con la que se presentó SPSLiDAR (Rueda-Ruiz et al., 2022).

La estructura abordada en este apartado es un grid compuesto por dos niveles. El primero, utiliza las zonas UTM para definir un conjunto de celdas sobre la superficie terrestre. Utiliza el sistema extendido de celdas definido por el MGRS: las celdas utilizan el espectro de valores recogido por este en el ámbito latitudinal (C a X), no limitándose exclusivamente a N o S. A efectos prácticos, el territorio consta de 60 divisiones a nivel longitudinal y 20 a nivel latitudinal, dando un total de aproximadamente 1200 zonas (existiendo algunas excepciones en la zona del archipiélago de Svalbard y en los círculos polares). Dentro de cada zona, se construye un segundo nivel, un grid disperso 2D. El tamaño de celda viene definido por el atributo *cellSize* de la entidad *Workspace*. La celda asociada a un punto se obtiene a través de la división entera de cada una de sus coordenadas por el tamaño de celda.

Existen dos operaciones principales a abordar en esta estructura: la inserción e indexación de *datasets* y la búsqueda de estos en base a criterios espaciales. Para el primero de estos dos procesos usamos la caja envolvente del *dataset*, disponible a través del atributo *boundingBox*. Para cada uno de los puntos de caja envolvente se calculan las celdas asociadas. En el caso de que ambos valores sean los mismos, se interpreta que la caja envolvente está definida de forma íntegra bajo la misma celda. En caso de que sean diferentes, la caja envolvente solapa con más de una celda: estas se obtienen iterando sobre las celdas comprendidas en el intervalo X e Y definido por las obtenidas para los dos puntos de la caja envolvente. El *dataset* se indexa a cada una de las celdas obtenidas. Nótese que al tratarse de un grid disperso la celda puede no existir todavía, por lo que puede ser necesaria su creación antes de indexar el *dataset*.

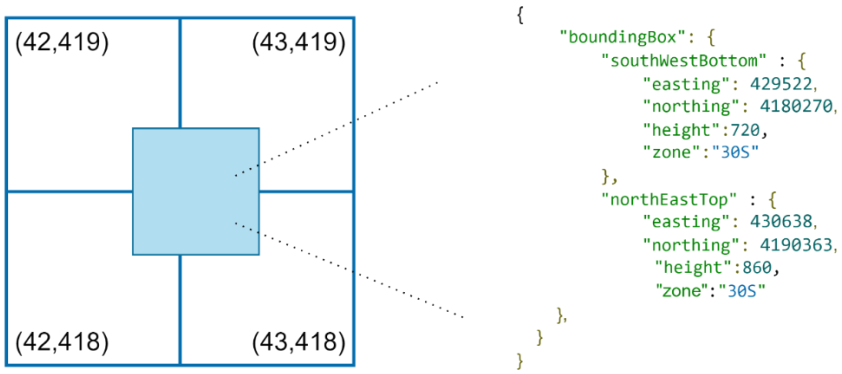


Figura3.2. Cálculo de celdas soportadas por una caja envolvente en un grid 2D.

Para ilustrar el proceso, en la Figura 3.2 se muestra un *dataset* de referencia. Con la coordenada *southWestBottom*, obtenemos la celda (42, 418). Con *northEastTop*, la (43, 418). En este caso todas las celdas ilustradas solapan con el *dataset*.

Sin embargo, existe la posibilidad de que las celdas inferidas mediante este proceso realmente no solapen con la nube de puntos que acabe localizando el sistema. Esto es un proceso normal y consecuencia de tratar de acotar una forma irregular, como es una nube de puntos, con un rectángulo. En la Figura 3.3 se muestra este caso, en el que dentro de la caja envolvente se dibuja el área real de la nube que representa el *dataset*. En esta situación, las celdas que no solapen serán finalmente descartadas, aunque este proceso se llevará a cabo durante la creación de la estructura de *datablocks*, momento en el que sí se tiene acceso a los ficheros de nubes de puntos. En este ejemplo, se actualiza la celda (42, 419), lo que puede implicar simplemente la eliminación de la referencia al *dataset* de esta celda, y eventualmente su completo borrado del sistema en caso de que esta sea la última referencia.

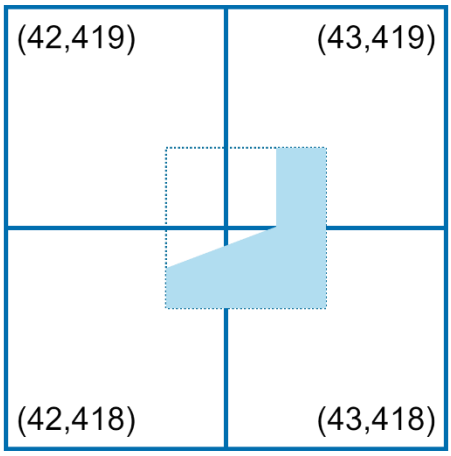


Figura3.3. Representación del área real del *dataset* presentado en la Figura 3.2. Es necesario descartar la celda (42,419) al no solapar con dicha área.

La Figura 3.4 muestra cómo este proceso se lleva a cabo durante la llamada al servicio de ingesta de *datasets*. Cuando los ficheros son procesados, se agrupan en base a las celdas en las que se ha dividido el *dataset*. Si para alguna de las celdas no existe un fichero asociado, esto implica que no existe ningún punto en ella y por lo tanto se detecta como un falso positivo. Es en este momento cuando se revierte la indexación previamente realizada durante el proceso de creación del *dataset*.

Las zonas UTM, si bien definen unos límites teóricos, estos no son estrictos, y por tanto es posible representar un punto en coordenadas de una zona distinta a la que le corresponde, siempre y cuando se asuma la consiguiente pérdida de precisión, que se acrecienta conforme nos alejamos del origen de coordenadas de la zona. En estos casos, lo que se hace es reproyectar los puntos de las esquinas de la caja envolvente al sistema de coordenadas de la otra zona, generando dos cajas envolventes. Para cada una se lleva a cabo el proceso descrito con anterioridad: se calcula la celda asociada a cada punto de la caja envolvente y se determina un conjunto de celdas potencial con las que solapa el *dataset*. En estos casos con múltiples zonas UTM, es muy probable que sea necesario el descarte de celdas a posteriori, una vez se tengan los ficheros de nubes de puntos, siguiendo el procedimiento descrito anteriormente. Aunque el *dataset* quede registrado en dos zonas UTM para facilitar su localización, internamente los puntos mantendrán la zona UTM especificada en los ficheros originales desde los que se importaron para evitar errores de precisión. En el caso del estándar LAS, esta información puede obtenerse de las cabeceras de los ficheros. En la Figura 3.5 se ilustra esta idea de generar celdas en varias zonas UTM para este tipo de *datasets*.

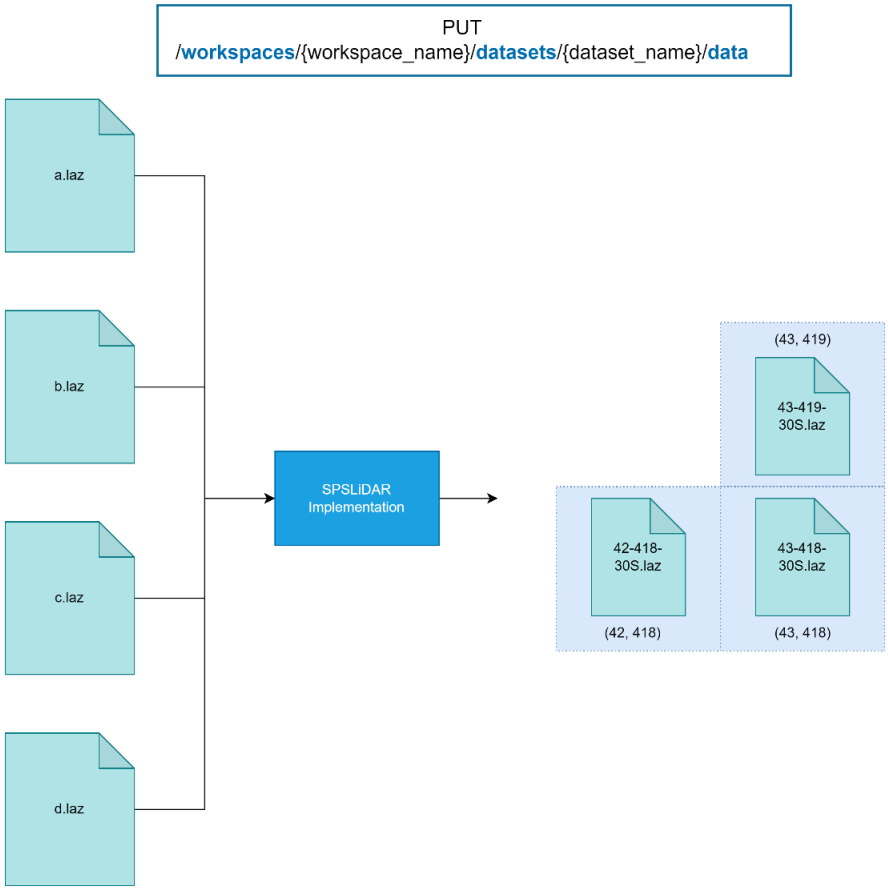


Figura3.4. Proceso de descarte de celdas consideradas como falsos positivos durante la etapa de construcción de la estructura de*datablocks*

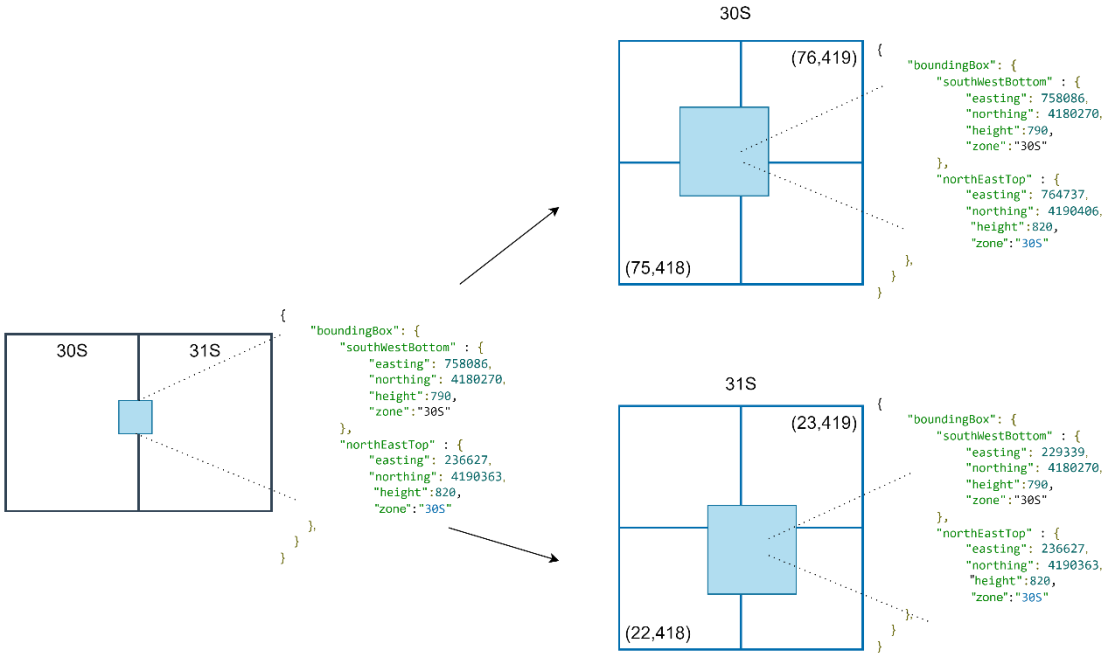


Figura 3.5. Indexación de celdas en *datasets* con coordenadas descritas en distintas zonas UTM.

Por último, debemos considerar los casos en los que el área del *dataset* esté representada por vértices adscritos a zonas UTMs no adyacentes. Primero, queremos recalcar que este tipo de escenarios no deberían producirse. Un *dataset* debe tener coherencia espacial, representando una porción continua del territorio o un elemento concreto de interés de origen natural o artificial, procedentes de una o más sesiones de captura en la misma zona. Un *dataset* formado por varias nubes de puntos no conexas es una construcción artificial que en muchos casos puede ser provocada por un error involuntario al seleccionar los archivos con los datos a importar durante su creación. Aun así, en el caso de que su inclusión fuese necesaria, el *dataset* no se asociaría inicialmente a ninguna celda, y sería tras la subida de todas las nubes de puntos que lo componen cuando se analizarían y calcularían las celdas a las que debe quedar indexado. De esta forma se evita crear un número excesivo de celdas en el sistema para un área que es difícil de delimitar en el caso de las zonas intermedias.

Una vez establecido cómo se realiza el proceso de inserción, debe definirse cómo se pueden resolver consultas de ventana sobre esta estructura. El proceso consta de dos fases principales ilustradas en la Figura 3.6. En memoria, se deben calcular los índices de las celdas asociados a la ventana definida. Esto es similar a la primera parte del proceso realizado en la inserción: para cada uno de los puntos que definen la ventana se calculan las celdas que tiene asociadas. Utilizando estos valores calculados (los denominaremos *WindowMinXId* y *WindowMaxXId* para el eje este-oeste y *WindowMinYId* y *WindowMaxYId* para el eje norte-sur), se define una consulta a ejecutar sobre el sistema de base de datos basada en estos filtros:

$$(CellUTMZone == WindowUTMZone)$$
$$\&\& (WindowMinXId \leq CellXId \leq WindowMaxXId)$$
$$\&\& (WindowMinYId \leq CellYId \leq WindowMaxYId)$$

Los parámetros con el prefijo *Cell* representan los atributos asociados al esquema de celda definido en base de datos. Se buscan por lo tanto aquellas celdas para las que la zona UTM sea igual y su identificador de celda esté en los rangos de los identificadores obtenidos para la ventana. Dentro del subconjunto de celdas candidatas, se aplica un segundo filtrado para obtener de forma precisa aquellos *datasets* dentro de las celdas que cumplen la condición. Para ello, utiliza los valores completos en los ejes X e Y de la ventana.

$$(Window.west \leq Dataset.west < Dataset.east \leq Window.east)$$
$$\&\& (Window.south \leq Dataset.south < Dataset.north \leq Window.north)$$

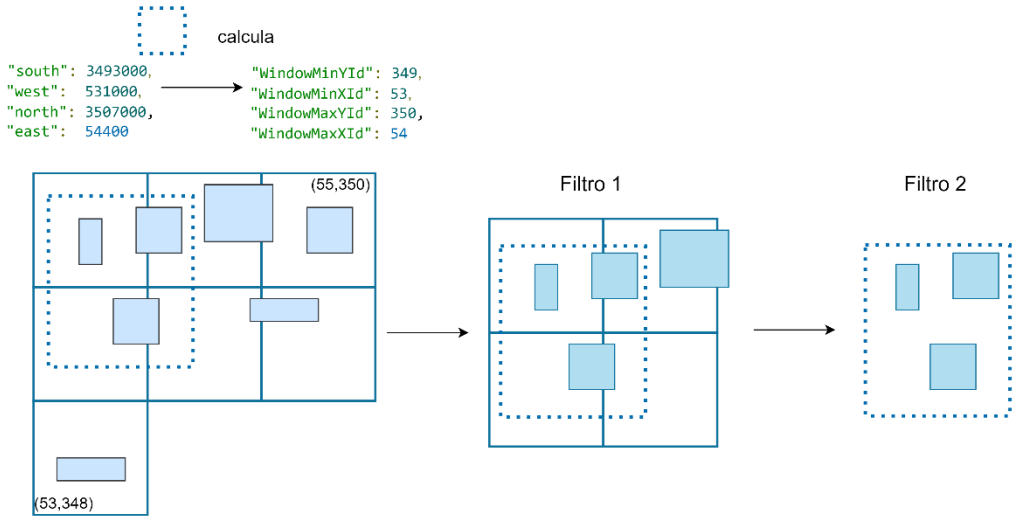


Figura3.6. Operación de búsqueda por ventana. Un primer filtro obtiene las celdas que solapan con la ventana y el segundo obtiene con un mayor grado de granularidad los *datasets* que coinciden de manera exacta con la ventana

Grid disperso UTM multicapa

Una de las desventajas de la estructura presentada anteriormente es la ausencia de adaptabilidad a distintos tamaños de *datasets*. El uso de un tamaño de celda pequeño hace que los *datasets* que abarquen grandes regiones sean indexados por un elevado número de celdas, incrementando el número de elementos a evaluar en las consultas. Por el contrario, un tamaño de celda excesivo reduce la eficiencia de la primera parte de la consulta, causando un mayor número de *datasets* a evaluar en la segunda etapa de filtrado.

Con esta estructura proponemos aplicar una estrategia similar a la vista en la sección anterior, aunque ahora, dentro de cada zona UTM, convivirán varios grids de distinto tamaño. Al insertar un *dataset*, se evaluará en función del tamaño de su caja envolvente cuál de los grids es el mejor candidato para su indexación, siguiendo un enfoque *bottom-to-top*. Para ello, iteramos sobre los distintos tamaños de grids definidos en la estructura de datos hasta que se cumpla la condición en la que el tamaño de la celda del grid sea mayor o igual que el lado de mayor tamaño del *dataset*. Si no se satisface la condición, se acaba seleccionando el grid con el mayor tamaño de celda disponible. De este modo conseguimos que cada *dataset* pueda quedar indexado con cuatro celdas a lo sumo, salvo para el caso de *datasets* que excedan el mayor de los tamaños definidos.

A la hora de llevar a cabo las consultas de área, la ventana definida deberá ser evaluada en los N niveles de las que conste la estructura. Esto implica que en la Figura 3.7, el primer proceso de filtrado incluye un número de condiciones adicionales proporcional al número de capas definidas. La ventana definida por el cliente será adaptada internamente a las tres capas del grid y se llevará a cabo una consulta que aplique los siguientes filtros para cada una de ellas, siendo *LevelN* el identificador de cada una de estas.

(CellUTMZone == WindowUTMZone)
&& (WindowMinXIdLevelN <= CellXIdLevelN <= WindowMaxXIdLevelN)
&& (WindowMinYIdLevelN <= CellYIdLevelN <= WindowMaxYIdLevelN)

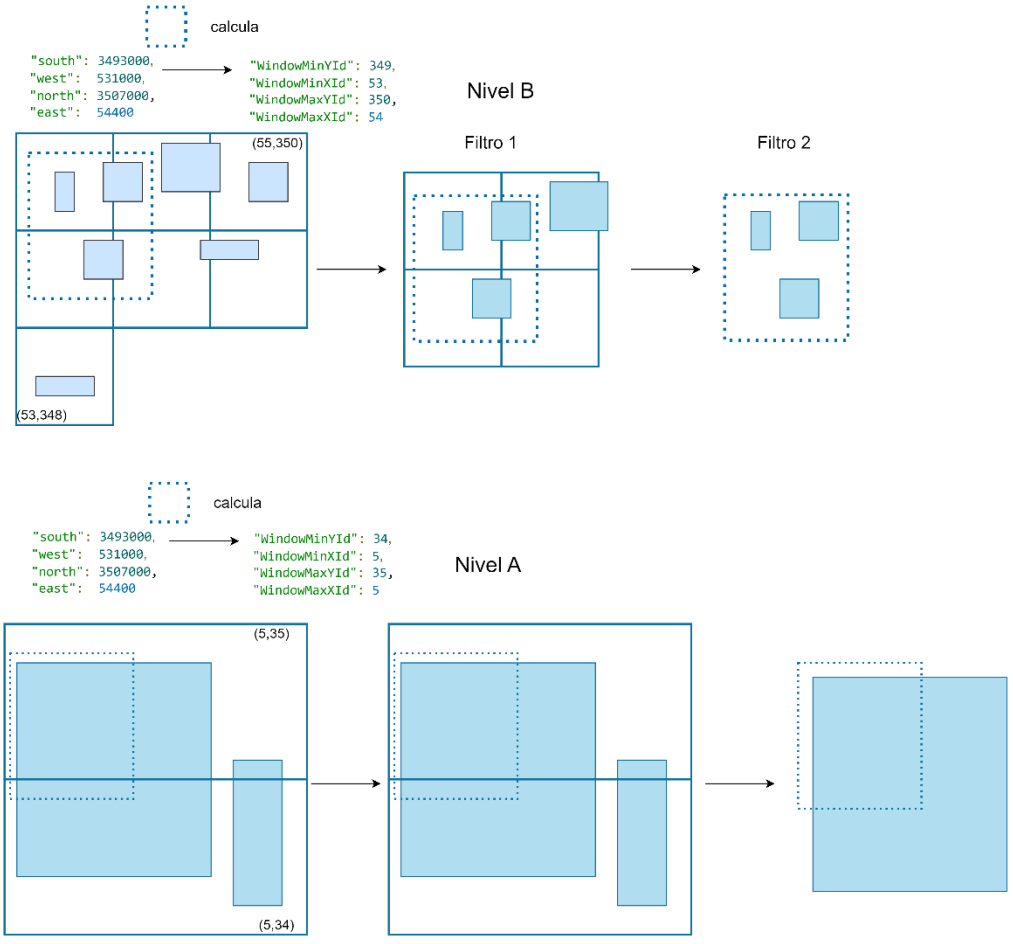


Figura 3.7. Consulta sobre un grid basado en dos niveles. El nivel B aloja *datasets* de menor tamaño mientras que el nivel A los de mayor. Los resultados de las consultas en cada nivel son agregados en el resultado fina

Este tipo de consulta puede ser resuelto a nivel de implementación siguiendo dos estrategias diferentes:

- Realizando una única consulta. En este enfoque directo se inspeccionan secuencialmente todos los niveles de la estructura; la complejidad de la consulta se ve incrementada al incluir un mayor número de condiciones.

- Distribuyendo el filtrado en múltiples consultas. Este enfoque puede beneficiarse de una implementación distribuida de la capa de persistencia, siempre que las celdas de cada

nivel estén almacenadas en nodos distintos. Con este enfoque, se pueden lanzar en paralelo las consultas, permitiendo que cada nodo resuelva una parte de la consulta por separado y agregando finalmente los resultados. Esto puede agilizar los tiempos de respuesta en sistemas que gestionen un número muy elevado de *datasets*, aunque añade un nivel de complejidad adicional debido a la necesidad de definir estrategias de distribución de los datos para que sea llevado de forma satisfactoria. También hay que considerar que el acceso a base de datos se suele realizar a través de un pool de conexiones con un tamaño predefinido. Establecer múltiples conexiones en paralelo puede limitar el número de conexiones disponibles para otras peticiones.

Independientemente de la estrategia utilizada, la respuesta final proporcionada por el servicio contendrá todos los *datasets* encontrados en todos los niveles. Mientras que con el primer enfoque esta combinación de los resultados se lleva a cabo en la base de datos, en la segunda se resuelve en memoria. Aprovechando la implementación asíncrona del stack tecnológico seleccionado, se podría llevar a cabo un streaming de los *datasets* recuperados al propio cliente que ha realizado a petición, de manera que se envíen datos de forma progresiva conforme las diferentes consultas se vayan resolviendo a nivel de base de datos.

Métodos de hashing alternativos

En los grids regulares descritos en las secciones anteriores los identificadores se obtienen de manera directa dividiendo las coordenadas por el tamaño de celda establecido. Sin embargo, existen otros mecanismos para indexar la información espacial. Estos se basan en el uso de curvas que abarcan el total del espacio, generando una codificación en base al orden en el que se recorren las distintas regiones (Deiotte & La Valley, 2017). Algunos de los más populares son la curva Z (o de Morton) y la curva de Hilbert (Figura 3.8). Su ventaja es asociar identificadores cercanos numéricamente a *datasets* cercanos espacialmente, lo que puede aprovecharse para optimizar el almacenamiento. Si trabajamos con coordenadas en forma de latitud/longitud, la codificación basada en GeoHash (Niemeyer, 2008) es otra alternativa que sigue la misma filosofía.

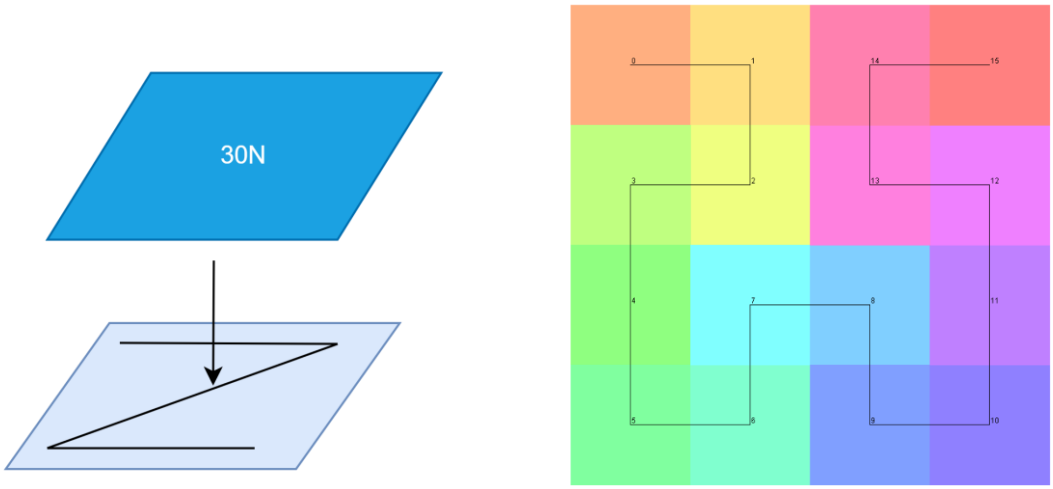


Figura3.8. Curva de Morton (izq.) y curva de Hilbert (dcha.)

Para poder definir una curva de estas características, es necesario conocer con antelación las dimensiones del espacio sobre el que se va a situar. Para la proyección UTM utilizada en los casos anteriores (complementada con la división en bandas latitudinales de MGRS de C a X), las dimensiones esperadas son de 6º de ancho y 8º de alto. En su conversión a metros estos valores varían en función de la ubicación de la zona debido a la naturaleza de este sistema de coordenadas, pero se pueden esperar valores máximos cerca de 700.000 metros de ancho y 900.000 metros de alto. Para simplificar, se genera un cuadrado de 1.000.000 de metros de lado, el cual da margen suficiente para aquellos casos puntuales en los que una nube de puntos obtenida en el borde entre zonas se extienda por encima de este límite. Será sobre esta región sobre la que se construya la curva.

Una coordenada puede no solamente trascender los límites teóricos de la zona sino también los de esta región de mayor tamaño que hemos definido. Estos casos podemos tratarlos como un error, omitiendo la indexación y devolviendo un código de respuesta para indicar el problema. En la Figura 3.9 se muestra un ejemplo de esto, en el que el Dataset D3 sería descartado.

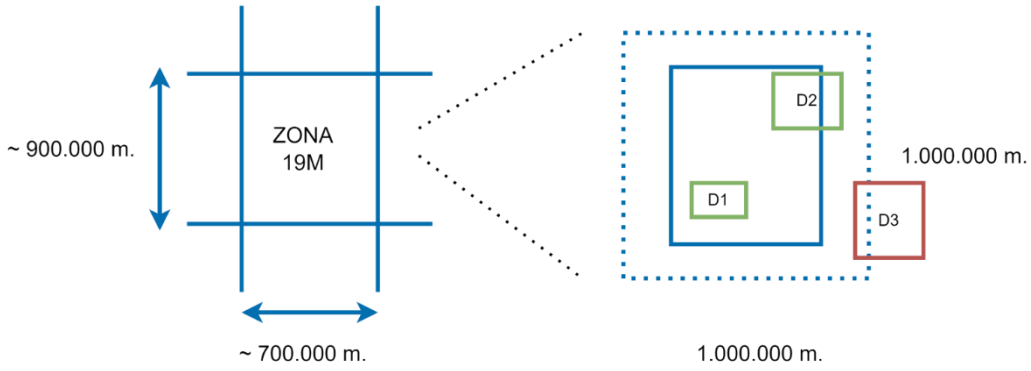


Figura 3.9. Indexación de *datasets* sobre una zona UTM utilizando una codificación basada en curva. El Dataset D1 queda encuadrado dentro de la zona UTM al completo, el Dataset D2 supera los límites esperados de la zona pero sigue estando en el rango del área sobre el que se dibuja la curva y el Dataset D3 es descartado por contar con parte de su área fuera de la zona.

Otra solución implicaría utilizar un grid, similar a los ya vistos anteriormente: una zona UTM tendría asociado internamente un grid disperso utilizando un tamaño de celda elevado, y dentro de cada celda se construiría la curva. En la Figura 3.9, el Dataset D3 sería indexado en dos de estas celdas, dividiendo su área de forma similar a la presentada en la Figura 3.5. Esta solución añade un nivel de complejidad mayor cuyo único fin es tratar estos casos límite, por lo cual consideramos preferible utilizar el primer enfoque.

Uso de árboles para la indexación de datasets

Pese a la popularidad de estructuras basadas en árboles para indexar información espacial, consideramos que no resultan igual de adecuadas para la organización de *datasets* a nivel de *workspace*. El bajo volumen de datos implicados en este caso, incluso si hablamos de *workspaces* con miles de *datasets*, no justifica la complejidad extra de esta estructura de datos. Además, la inserción de *datasets* es un proceso progresivo a lo largo del tiempo. Durante la inserción de un

nuevo *dataset* en el sistema, eventualmente sería necesaria la creación de un nodo y el rebalanceo del árbol. Estas operaciones pueden ser costosas ya que requieren múltiples lecturas y escrituras.

Otras ventajas de estas estructuras jerárquicas, como puede ser la posibilidad de ofrecer niveles de detalle, no tienen utilidad en este contexto, ya que no es necesario generar una descarga progresiva de los *datasets* ni tareas similares. Tampoco son útiles las relaciones entre nodos que proveen este tipo de estructuras, ya que los *datasets* son elementos disjuntos sin relación entre sí salvo la de pertenencia al mismo *workspace*. No hay una relación padre-hijo como se plantea para la entidad *Datablock*. Por ello la opción más efectiva es el uso de estructuras de datos sencillas como el grid regular, en las que la inserción de un nuevo *dataset* es directa, y en las que podemos identificar cada nodo mediante un conjunto reducido de atributos de fácil indexación.

Indexación usando criterios temporales

Por lo general, en las búsquedas de *datasets* prima el criterio espacial frente a otros. Sin embargo, cuando los *datasets* se encuentran en una misma zona geográfica, la utilidad de este tipo de búsquedas disminuye. Esta situación se da en el primero de los casos de estudio presentados en el **Capítulo 2**: tareas de construcción o remodelación sobre patrimonio histórico ubicado en una región concreta y donde la principal diferenciación entre *datasets* vendrá dada por su fecha de adquisición.

Para estos casos proponemos una solución basada en torno a la definición de un sistema de coordenadas temporal. El proceso es idéntico al de los casos anteriores, con la particularidad de que ahora trabajamos sobre una estructura unidimensional: se define un tamaño de celda en base a una unidad temporal, se definen celdas que agrupan *datasets* próximos en el tiempo, y se resuelven las consultas traduciendo las fechas inicial y final a este sistema para encontrar los *datasets* adecuados.

También puede combinarse el particionamiento temporal con algunas de las estructuras basadas en particionamiento espacial que hemos descrito anteriormente. De este modo, se extiende una estructura como un grid 2D para dar lugar a una estructura 3D que puede indexar información en los ejes este-oeste, norte-sur y temporal. Esto permite optimizar consultas de tipo espacio-temporal, a cambio de incrementar la complejidad de la estructura y la necesidad de mantener un índice adicional en base de datos.

Estructuras de datos a nivel datablock

La relación *Dataset-Datablock* define la estructura utilizada para organizar los *datablocks* en los que se distribuye la nube de puntos asociada a un *dataset*. En el **Capítulo 2** se mencionan las ventajas de este modelo, entre las que destaca la posibilidad de descomponer la nube en paquetes de tamaño reducido que contienen puntos relacionados entre sí, lo que permite a los clientes obtener de forma rápida el subconjunto de información de la nube que necesitan.

La creación de esta estructura y los *datablocks* asociados solo se lleva a cabo una vez, durante la importación del *dataset* al repositorio. En función de la estructura, los *datablocks*

mantienen relaciones entre sí, permitiendo la exploración del modelo y la generación de niveles de detalle que completan la información de manera progresiva y acorde a la región de interés establecida por el cliente. Como veremos, este proceso de inserción es más complejo, y requiere de un mayor tiempo de ejecución en comparación con la inserción de nuevos elementos llevada a cabo en la estructura *Workspace-Dataset*.

En esta sección nos centraremos principalmente en el proceso de creación de estas estructuras y la definición de los *datablocks* a partir de ellas. También se muestra cómo se han integrado las distintas herramientas de LasTools para poder llevar a cabo las transformaciones sobre los ficheros de nubes de puntos para dar lugar a los archivos finales indexados por cada *datablock*.

Octree

El uso de estructuras jerárquicas en aplicaciones de visualización aporta claras ventajas, permitiendo incrementar la resolución de las zonas de nube de forma progresiva en función de las necesidades del usuario. Navegando a través de la estructura se puede acceder a descendientes que aporten distintos niveles de detalle. Es por esto que los octrees son una de las estructuras de datos más habituales a la hora de almacenar modelos de puntos (Elseberg et al., 2013; L. Huang et al., 2021; Scheiblauer & Wimmer, 2011; Schütz et al., 2020), y consecuentemente, en nuestra implementación de referencia presentada en SPSLiDAR (Béjar-Martos et al., 2022) utilizamos el octree como estructura. El proceso para la construcción se distribuye a lo largo de varios pasos descritos a continuación:

1. Generación de los ficheros base de cada raíz. Este proceso está íntimamente relacionado con la estructura seleccionada a nivel *Workspace-Dataset*. En función del número de celdas sobre las que se distribuye el *dataset* en esta estructura, se realiza una agrupación y particionamiento preliminar de los ficheros. El número de octrees construidos dependerá del número de celdas que tengan puntos, generándose uno por cada una de ellas. En este momento es además cuando se verifica si alguna de las celdas sobre las que se indexó el *dataset* debe ser descartada, validando el contenido real de los ficheros frente a estas y actualizando la información en la capa de persistencia. También se marca un campo definido a nivel de base de datos en el que se guarda el estado del *dataset*, indicando que se ha iniciado el proceso de construcción de la estructura de *datablocks*. Esto permite que, en caso de que se repita la petición durante el proceso, se pueda informar al cliente de que este ya se está ejecutando.

2. Utilizando los ficheros obtenidos tras el particionamiento anterior, se generará un octree a partir de cada uno de ellos. Con el objetivo de liberar el hilo encargado de gestionar esta petición y evitar posibles bloqueos debido a la carga de este proceso, a partir de este momento la petición es delegada a un planificador propio. La clase abstracta *Scheduler* viene definida en la API de Reactor y nos ofrece un modelo para gestionar la definición y delegación de tareas en hilos. En nuestro caso, utilizamos la implementación *boundedElastic*, que es la más adecuada para procesar tareas de larga duración. Cada uno de los octrees que se generen será procesado en un hilo independiente y de forma paralela, con el objetivo de agilizar la construcción del octree. Este proceso se ilustra en la Figura 3.10.

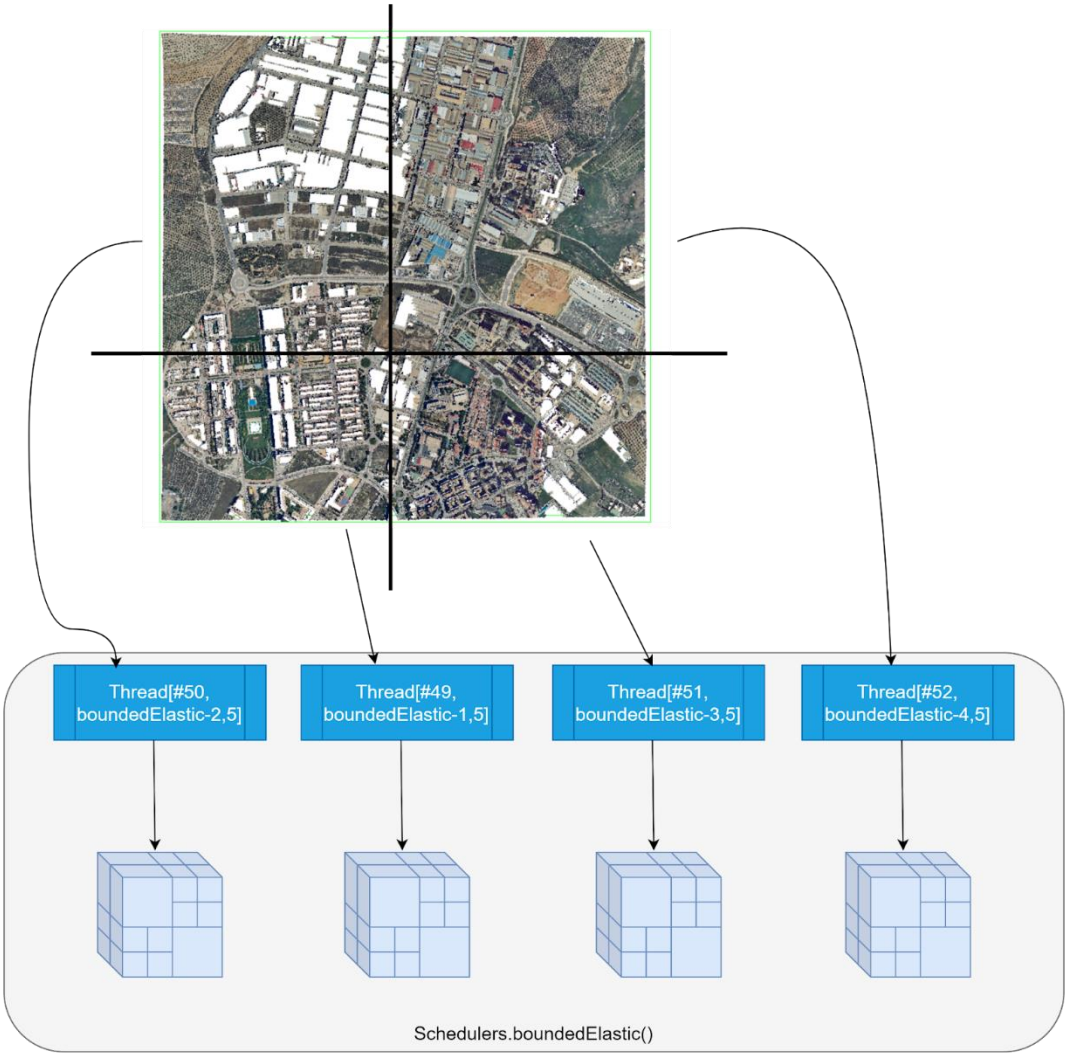


Figura3.10. Distribución de una nube de puntos en múltiples octrees procesados de forma concurrente dentro de un *Scheduler* propio.

3. Los octrees se construyen utilizando una función recursiva, que acepta como parámetros un *datablock* y un valor numérico que marca el número máximo de puntos que deberá contener el archivo que se le asocie. El *datablock* inicialmente guardará una referencia a un fichero que contendrá todos los puntos de la nube asociados a su región y que no han sido previamente asociados a otro *datablock*, al cual llamamos *fichero base*. En el caso del nodo raíz, este fichero será el que se ha obtenido en el paso 1 para su celda correspondiente. Se realiza un muestreo de este fichero (en este caso aleatorio, pero otras estrategias también pueden ser válidas) para obtener una versión con un número de puntos menor. Estos puntos se eliminan del fichero base y se particionan en 8 subregiones. Si en alguna de estas regiones no hubiera puntos, no se crearía un fichero para ella, ni un *datablock* hijo. A partir de cada partición se genera un *datablock* nuevo y se le asigna el fichero obtenido, que será utilizado para invocaciones recursivas posteriores. El *datablock* actual actualiza la referencia en su array de nodos hijos para guardar el identificador de cada uno de estos *datablocks* creados, y finalmente, se invoca la función recursiva para cada uno de ellos. Esta función devuelve un flujo de *datablocks* de Reactor. Cada vez que en la función se realiza la llamada recursiva, el resultado de esta llamada

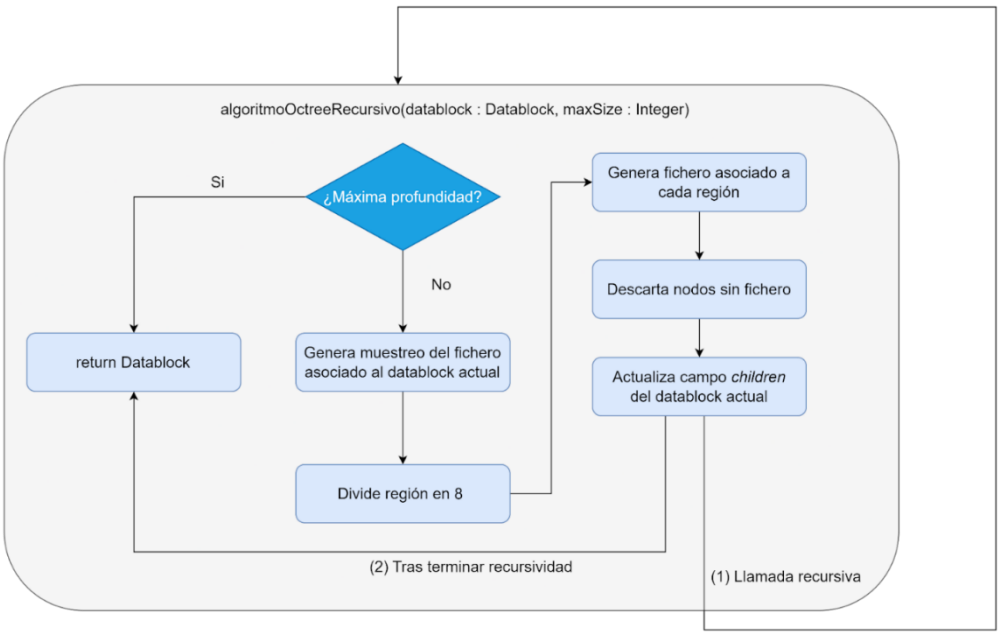


Figura 3.11. Proceso para la construcción del octree

se concatena a este flujo. Mientras que el árbol se construye partiendo del nodo raíz, el flujo de *datablock* se va completando en el orden inverso. Primero, se añadirán los *datablocks* asociados a los nodos hoja, progresando en orden inverso hasta llegar al nodo raíz. En la Figura 3.11 se sintetiza este algoritmo. El muestreo de un fichero base para obtener una versión reducida se realiza con las operaciones de LASTools *las2las -keep_every_nth N* y *las2las -drop_every_nth N*, siendo N obtenido a partir de la división del número total de puntos del fichero y el tamaño máximo soportado. Los ficheros base generados para cada uno de los hijos se obtienen a través de la operación *las2las -keep_xyz minX minY minZ maxX maxY maxZ*. La secuencia de estas operaciones se muestra en la Figura 3.12.

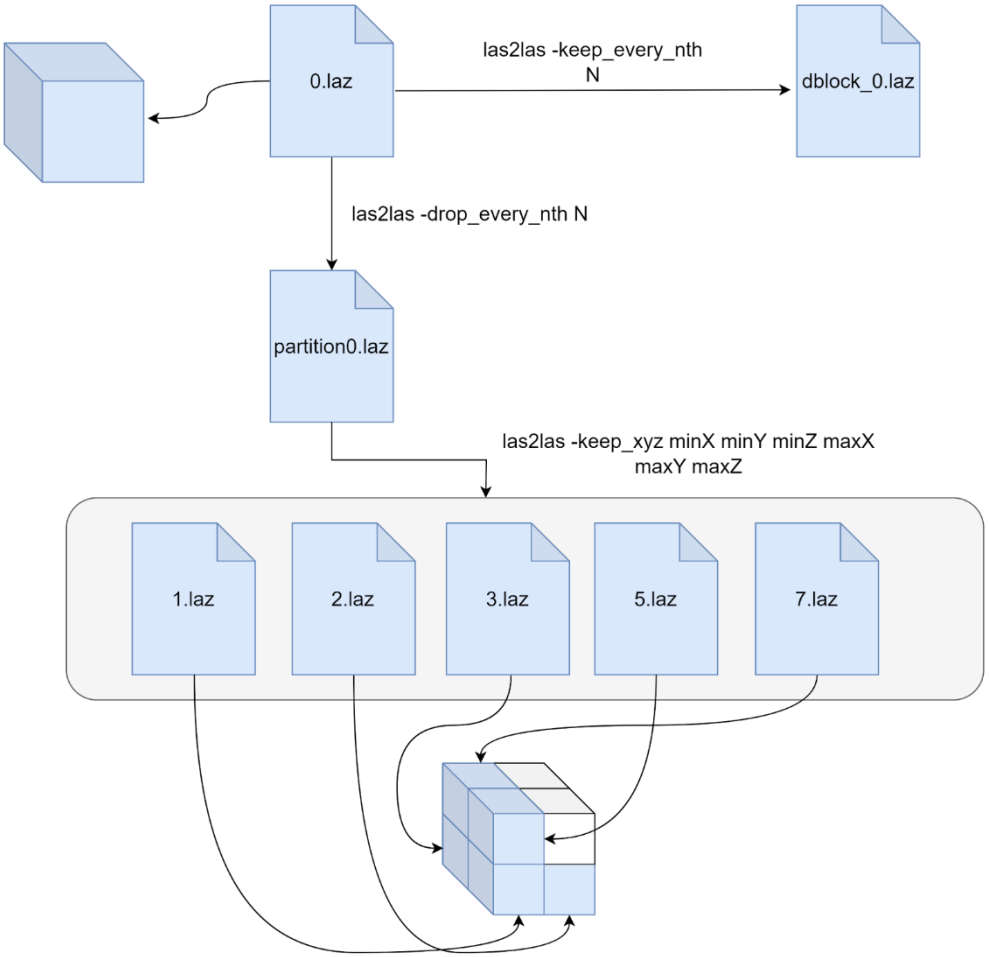


Figura3.12. Operaciones de LasTools utilizadas para generar un fichero de nube de puntos. A partir del nodo raíz se generan 4 ficheros correspondientes a distintas zonas

4. Almacenamiento de *datablocks* y ficheros. Finalmente, el flujo obtenido es procesado para realizar la inserción de los *datasets* y sus ficheros asociados en la base de datos. Los ficheros temporales generados durante el proceso son eliminados. También actualizamos el campo de la entidad *dataset* que almacena su estado para indicar que el proceso se ha completado.

Grid disperso

También sería posible usar una estructura no jerárquica basada en un grid disperso, con una filosofía similar a las propuestas en la sección de la relación entre *workspace* y *dataset*. En el **Capítulo 2** se presentaron dos formas de acceder a los *datablocks*: la primera, mediante un proceso progresivo por el cuál, utilizando las referencias en la raíz definidas en el *dataset* de interés, se puede acceder a la estructura y acceder a distintos niveles de profundidad usando los enlaces entre *datablocks*. Y la segunda, mediante un servicio que permite establecer consultas a través de filtros en forma de ventanas de tipo espacial. En este caso, el servicio devuelve un listado compuesto por uno o más *datablocks* que se adecúan a estos filtros. Cuando la forma de interactuar con el modelo se lleva a cabo fundamentalmente a través de este segundo método y

en aplicaciones en las que el concepto de nivel de detalle no es necesario, las estructuras jerárquicas no presentan tantas ventajas. Por el contrario, podemos utilizar un modelo de división espacial más sencillo en forma de grid disperso.

Los pasos seguidos para la construcción de la estructura son:

1. Generación de los ficheros base de cada raíz. Este paso es idéntico al paso 1 establecido en el octree y viene fundamentado por las mismas razones.

2. Ordenación de los puntos. Para cada uno de los ficheros obtenidos como resultado del paso 1 se realiza una ordenación de sus puntos siguiendo un criterio de proximidad espacial. Esta ordenación va a facilitar en el siguiente paso la división de la nube en conjuntos de puntos contiguos espacialmente que darán lugar a los *datablocks*. La opción más precisa pasa por realizar una indexación espacial, utilizando una curva Z como las que se han presentado anteriormente en este capítulo. Otra solución más rápida, pero menos exacta, consiste en utilizar el atributo *gps_time* presente en el estándar LAS, partiendo de la idea de que dos puntos adquiridos en momentos similares han de estar próximos, siguiendo el recorrido del escáner. Sin embargo, este criterio puede presentar problemas si existen múltiples barridos de una misma zona realizados en distintos momentos. En cualquier caso, ambas soluciones están soportadas de forma nativa por la herramienta *lassort*, que forma parte de LasTools.

3. Generación de los ficheros asociados a los *datablocks*. Se recorre el fichero y usando la ordenación establecida se agrupan los puntos espacialmente, generando los ficheros finales sobre los que se define cada *datablock*. El número de ficheros será igual a la volumetría de puntos del fichero original dividido entre el valor del atributo *datablockSize* definido para el *dataset*. En caso de que la división no sea exacta, el último fichero tendrá un tamaño menor. Es importante que el solape entre las cajas envolventes asociadas a los puntos de cada *datablock* sea mínimo. En caso contrario, ante una consulta de ventana el número de *datablocks* devuelto tenderá a ser mayor, y por tanto el volumen de información enviado al cliente se incrementará, deteriorando el rendimiento y enviando un mayor número de puntos definidos como falsos positivos. La Figura 3.13 muestra este concepto, ilustrando como la ordenación de puntos realizada en el paso anterior puede ayudar a la hora de conseguir *datablocks* más concisos y reduciendo el número de falsos positivos en las consultas de ventana.

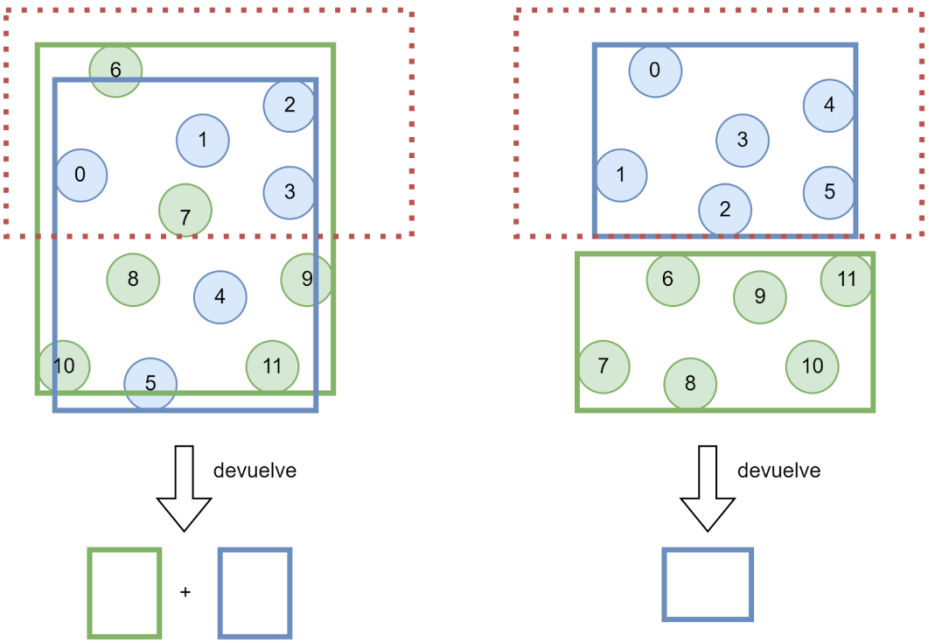


Figura 3.13. Un fichero con puntos no ordenados en base a un criterio de proximidad (izq.) y un fichero con puntos ordenados (dcha.). El número en cada punto indica su posición en el fichero. Se generan en ambos casos dos *datablocks*, marcados en verde y azul. Sin embargo, el área abarcada por estos es mayor en la izquierda, de manera que, ante la consulta de ventana marcada en rojo, se devuelve el doble de información, con puntos en exceso que no se adhieren a la consulta original.

4. Generación de *datablocks* e indexación mediante grid. Se genera una instancia de *datablock* por fichero y se indexan las cajas envolventes de estos *datablocks* en un grid, de manera que cada una de las celdas pueda referenciar uno o más *datablocks*. Este grid se usa internamente con el fin de optimizar las consultas y no es expuesto a través de la API. En la Figura 3.14 se muestra cómo se genera el grid. El tamaño de celda puede ser añadido como un parámetro adicional a nivel de *dataset* en el modelo SPSLiDAR 1.0. En SPSLiDAR 1.1, la instancia de la entidad *DatasetDataStructure* incluirá en su lista de parámetros este valor.

5. Almacenamiento de *datablocks* y ficheros. Al igual que con el octree, los *datablocks* generados y los ficheros finales asociados se almacenan en el sistema de base de datos.

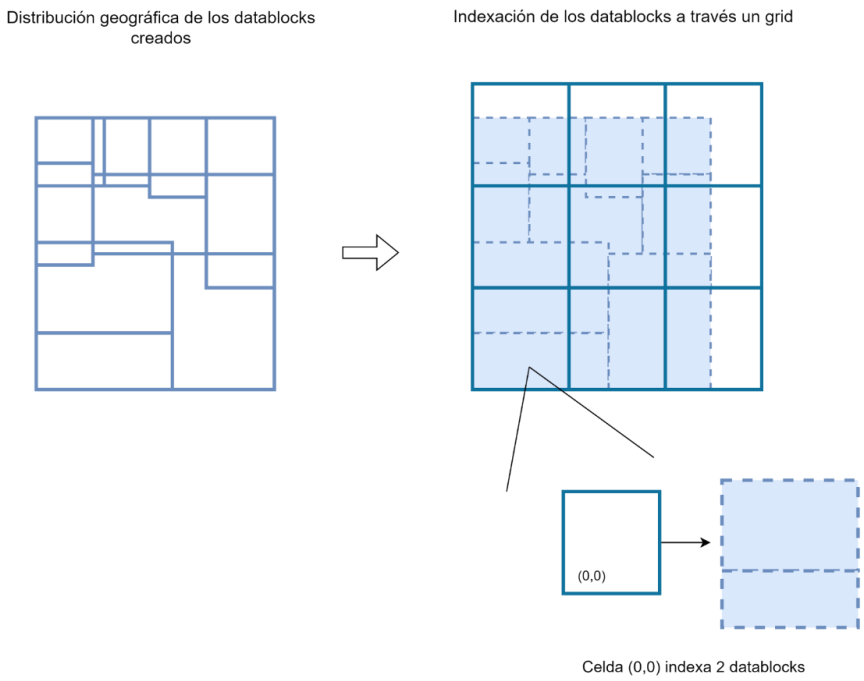


Figura 3.14. Una nube de puntos es distribuida en *datablocks* que son indexados a través de un grid, de manera que un *datablock* pueda quedar indexado por una o más celdas.

La estructura final consistirá en una serie de *datablocks*, indexados a nivel de *dataset* sin una estructura jerárquica. No obstante, es posible extender esta implementación para ofrecer una jerarquización con la que establecer niveles de detalles. Se introduce un mayor nivel de complejidad lógicamente, por lo que es necesario evaluar su idoneidad en cada caso. El enfoque que proponemos en esta versión extendida pasa por definir múltiples niveles de grids. Los pasos establecidos anteriormente seguirán siendo necesarios y se realizará un proceso de abajo hacia arriba para generar varios niveles de detalle. Para cada celda del grid construido, se define un único *datablock* que tendrá una región asociada a la extensión geográfica de la celda. El fichero asociado se genera de la siguiente manera: primero, se accede a los *datablocks* indexados por esta celda y se recuperan sus ficheros. Estos se combinan, se aplica un filtrado espacial para limitar la región del fichero combinado a la determinada por la celda (con el fin de evitar puntos no pertenecientes a esta región en caso de que alguno de los *datablocks* indexados solape con más de una celda) y se realiza un proceso de muestreo para limitar el tamaño del fichero al definido por el atributo *datablockSize*. Este proceso se puede repetir hasta satisfacer un número de iteraciones preestablecido, o hasta que el área total quede cubierta por una única celda. En la Figura 3.15 se muestra cómo se pueden establecer niveles de detalles y relaciones bajo esta premisa.

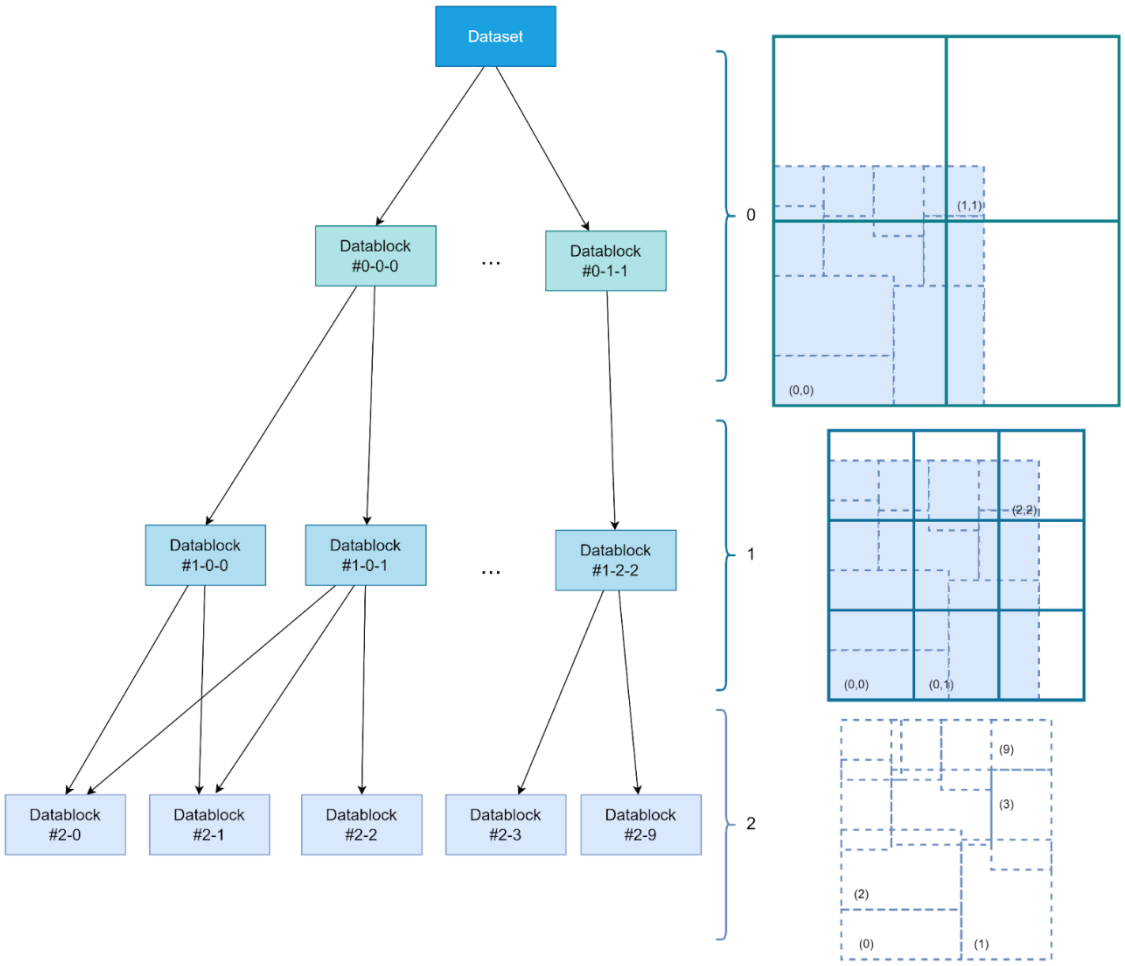


Figura 3.15. Generación de niveles de detalle usando un grid multinivel con distintos tamaños de celda. Cada círculo representa un *datablock*. El grid con la celda de mayor tamaño marca el nivel 0 de la jerarquía y es el referenciado por el *dataset*. El segundo nivel muestra los *datablocks* generados en el grid con la celda de menor tamaño. Por último, los *datablocks* hoja son aquellos que fueron construidos en el paso 3 del proceso descrito previamente. Cada identificador de *datablock* incluye el nivel de la estructura (0,1,2) y un identificador correspondiente a su posición dentro del nivel. En los niveles 0 y 1, cada *datablock* está representado por un valor asociado al eje X e Y de la celda del grid que lo constituye. En el nivel 2 los *datablocks* cuentan con un solo valor, en este caso asociado al orden en qué fue construído el fichero en el paso 3 del proceso.

Un aspecto a destacar de esta versión multinivel de la estructura es la presencia de puntos redundantes. Cuando se genera un *datablock* asociado a una celda del grid se realiza un muestreo de los puntos de los ficheros asociados a los *datablocks* que indexa en el nivel inmediatamente inferior. Los puntos seleccionados se copian al fichero del *datablock* permaneciendo también en los ficheros originales. Por ello en las consultas de ventana, los *datablocks* de los niveles intermedios generados se ignoran y solo se procesan los datablocks del último nivel de la estructura. A nivel interno, la implementación debe tener en cuenta que al resolver las consultas de ventana, las celdas del grid en las que se apoya pueden indexar a más de un *datablock*, debido aplicar un posprocesamiento para descartar duplicados antes de enviar la respuesta al cliente.

Implementación mediante estructuras de datos anidadas

En el seno del proyecto de investigación donde se desarrolló el trabajo descrito en esta tesis se propuso la definición de un marco flexible para crear estructuras de datos espaciales para gestionar nubes de puntos LiDAR en el contexto de Big Data Geoespacial (Ogayar-Anguita et al., 2023a). Yendo más allá de las típicas soluciones híbridas de dos niveles para optimizar operaciones específicas, como el almacenamiento o el rendering, en dicho artículo se presenta una metaestructura que puede tener una profundidad ilimitada y una combinación personalizada de estructuras anidadas, como grids, quadtrees, octrees o kd-trees a criterio del diseñador de la arquitectura del sistema. Con este enfoque, la indexación externa de nubes de puntos se puede adaptar a diferentes tipos de *datasets*, teniendo en cuenta la distribución espacial de los datos. Por consiguiente, se puede lograr la indexación espacial óptima para cualquier tipo de *dataset*, desde pequeñas escenas basadas en TLS hasta escenas basadas en ALS a escala planetaria. Este enfoque nos permite trabajar con conjuntos de datos superpuestos de diferentes resoluciones capturados con distintas tecnologías de adquisición en la misma estructura.

Aquí, el aspecto clave es diseñar lo que denotamos como estructura global, es decir, una combinación de estructuras anidadas. Cada una de estas estructuras se considera una estructura interna, que puede ser cualquier estructura espacial definida por el usuario. En las experimentaciones se probaron versiones adaptadas de grids, kd-trees, quadtrees y octrees, pero también se pueden integrar otras estructuras espaciales en este framework. Cada combinación posible de estructuras internas (una estructura global) debe ser definida por el diseñador del sistema siguiendo ciertos criterios que deben depender del conjunto de datos específico y los requisitos de la aplicación. La Figura 3.16 muestra todos los conceptos aquí presentados a través de un ejemplo. Cada estructura interna tiene sus propios niveles de profundidad, que se denominan niveles locales, que también ocupan un lugar en el esquema global, donde a su vez son considerados como niveles globales. El anidamiento de estructuras espaciales implica que los datos se propagan hacia abajo por la estructura hasta llegar a los nodos hoja. Después, para cada nodo hoja se debe calcular si se necesita crear un nuevo nivel inferior para distribuir los datos en subconjuntos más pequeños. Cada vez que se crea un nuevo nivel durante la subdivisión espacial, la estructura interna propietaria de un nodo determinado crea un nuevo nivel en ese nodo, solo si es una estructura jerárquica y no se alcanza la profundidad máxima. Si es el caso, se consulta el esquema de estructura global para ver cuál es la siguiente estructura interna para crearla como nueva instancia y adjuntarla a ese nodo.

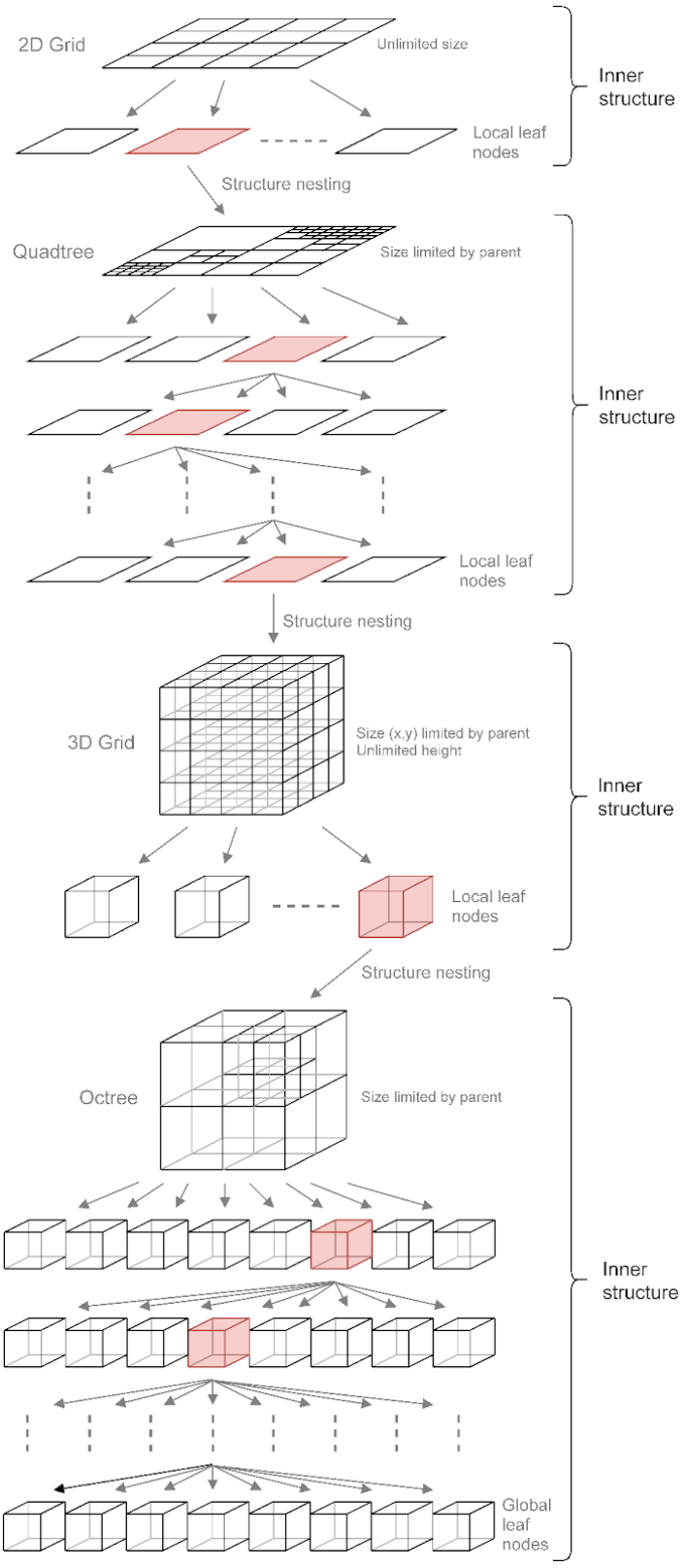


Figura3.16. Ejemplo de una estructura global compuesta por un grid 2D de quadtrees de grids 3D de octrees. La cuadrícula 2D se utiliza como estructura de nivel superior. Los octrees se utilizan para las regiones más pequeñas (Ogayar-Anguita et al., 2023a)

Para combinar y conectar todas las estructuras internas, se deben aplicar algunos principios de diseño en su implementación. En los prototipos realizados se han implementado

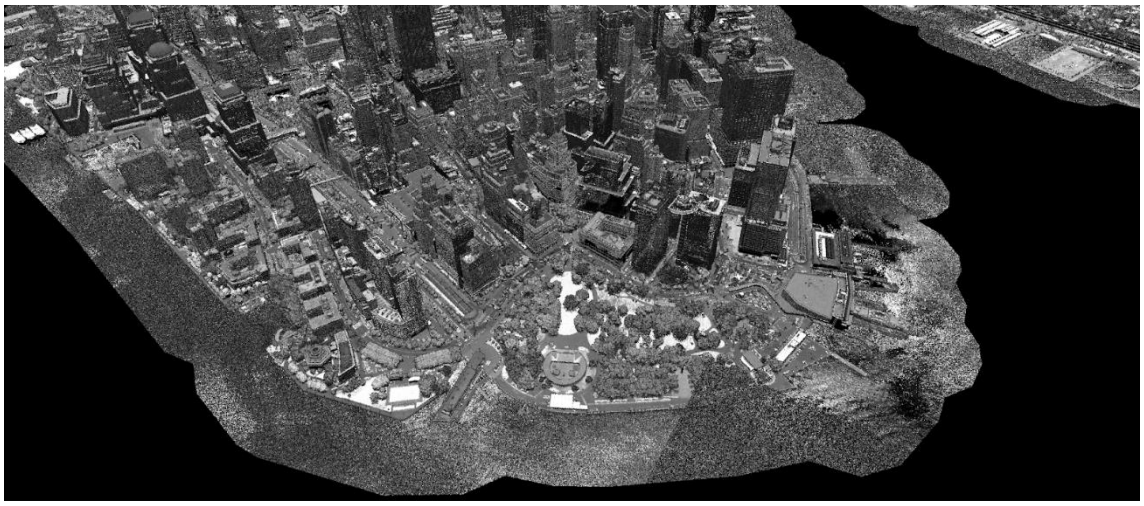


Figura 3.17. - Uno de los datasets LiDAR utilizados en los experimentos del trabajo de Ogayar-Anguita et al., 2023a) Isla de Manhattan. © City of New York, NYC Open Data. Se utilizó una anidación de grid 2D, quadtrees, grids 3D y octrees, en ese orden.

adaptaciones específicas de las estructuras, por lo que grid, quadtrees, octrees, etc. deben ser versiones adaptadas cuyas particularidades son las siguientes. En primer lugar, asumimos que todos los nodos en cada estructura espacial son del mismo tamaño en todas las dimensiones, es decir, cuadrados para estructuras planas y cubos para estructuras volumétricas. Esto simplifica enormemente el resto de decisiones a tomar, y casi todas las demás soluciones publicadas siguen esta misma convención. En segundo lugar, el esquema de estructura jerárquica global, definido por el usuario, debe ser el mismo para todo el modelo, y durante todo su ciclo de vida. Eso significa que durante su construcción, cuando una determinada estructura espacial alcanza su límite de crecimiento (máximo nivel de profundidad, máximo número de puntos por nodo, etc.), se crea una nueva instancia de estructura espacial y se le adjunta (si no se alcanza la profundidad global máxima), y el tipo de esa nueva estructura siempre es el mismo, dependiendo solo de su nivel en el esquema global (Figura 3.17).

Lo más interesante de este trabajo es que se puede integrar con SPSLiDAR, consiguiendo aumentar notablemente la flexibilidad del sistema. La principal ventaja radica en la mejor adaptabilidad de las estructuras espaciales utilizadas a la distribución espacial de los puntos de cada *dataset*. De esta manera, para escaneados TLS lo habitual será mezclar quadtrees con otras estructuras internas como otro nivel de quadtrees, u octrees para edificaciones. Desde el punto de vista del cliente, la navegación es transparente, al quedar abstraída por una instancia de *datablock*. El cliente podrá navegar sobre las jerarquías definidas y la transición entre estructuras será transparente para él. Aun así, podría ser interesante indicar de forma explícita a cuál de las estructuras pertenece un *datablock*. Para ello, se puede añadir un campo adicional en la entidad *datablock* que refleje esto.

En la Figura 3.18 se muestra un ejemplo concreto de este tipo de estructuras anidadas para la organización de los *datablocks* de un *dataset*, basada en la concatenación de un quadtree y un octree. La idea es realizar una codificación inteligente de los identificadores de los *datablocks* para establecer la correspondencia entre estos y los nodos dentro de cada una de las estructuras anidadas. En este ejemplo cada *datablock* tiene asociado un identificador compuesto por la celda a nivel de *Workspace-Dataset* en la que se encuentra (#42-418-30S) y el identificador dentro de la estructura de *datablocks*. Puesto que el octree ilustrado está asociado al *datablock*

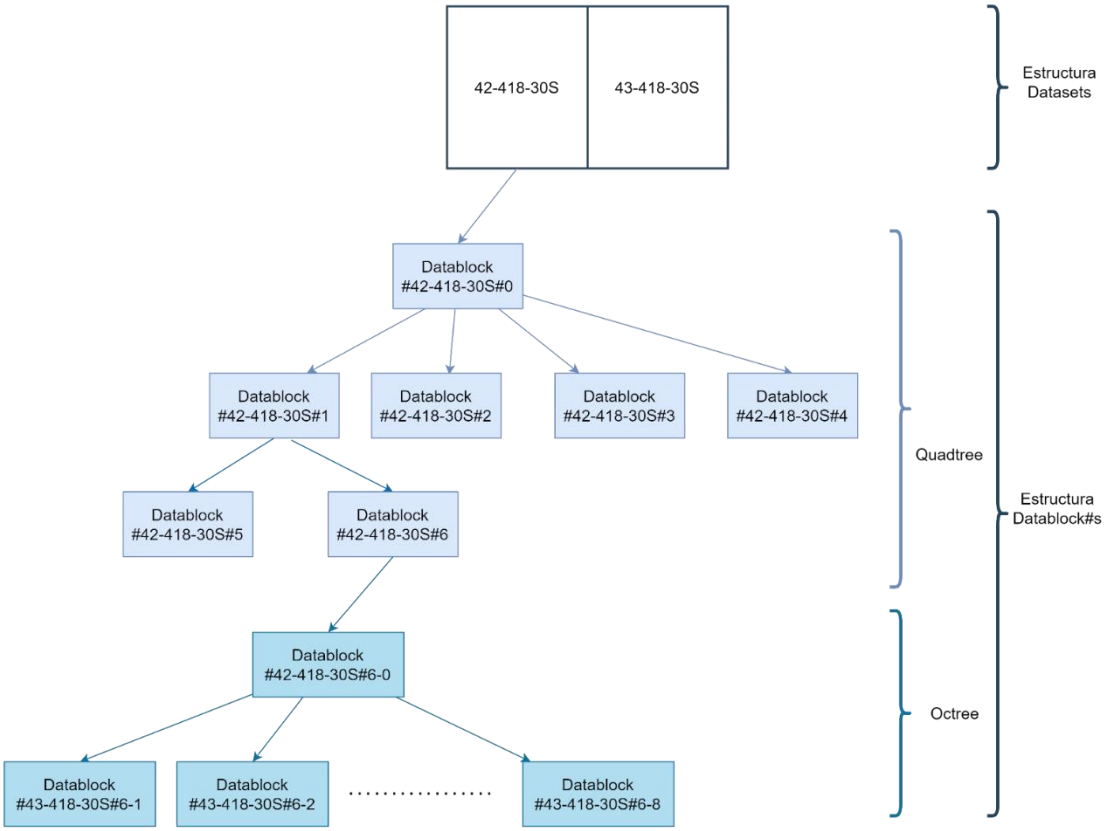


Figura3.18. Representación del uso de un quadtree y un octree como estructuras anidadas. El quadtree representa el nivelsuperior dentro de la jerarquía y los nodos hoja asociados a regiones con un subconjunto puntos mayor al máximo establecido generan una estructura anidada en forma de octree bajo la cual se distribuirán los puntos restantes.

con identificador #6 dentro del quadtree, todos los nodos tienen presente este valor en su propio identificador; siguen una estructura de tipo {quadtreeId}-{octreeId}: #6-0, #6-1, etc. A nivel de implementación esto implica que el campo *id* de la entidad *Datablock* deba ser definido como un String para soportar estos valores.

En la Figura 3.19 se ilustra la representación de los metadatos asociados a los *datablocks* en la frontera entre el quadtree y el octree. El *datablock* que actúa como nodo raíz del octree presenta un nuevo tipo de identificador, resultado de la concatenación del identificador del nodo hoja del quadtree dentro del cual está definido y el identificador propio dentro del octree. En esta implementación, el *datablock* raíz del octree no tendrá un fichero de nubes de puntos asociado, de ahí que su valor *size* sea 0, actuando meramente como una puerta de acceso a la estructura de *datablocks*, puesto que la región que representa es idéntica a la del nodo hoja del quadtree y por tanto es redundante. Otra alternativa posible es la eliminación de este nodo raíz, de manera que el nodo hoja presente en su array de *subDatablocks* los identificadores del primer nivel del octree con puntos asociados.

Este tipo de estructuras pueden ser construidas tanto en SPSLiDAR 1.0 como en SPSLiDAR 1.1. En la sección final de este capítulo se muestra la representación bajo el modelo SPSLiDAR 1.1 de la misma estructura ilustrada en la Figura 3.18.

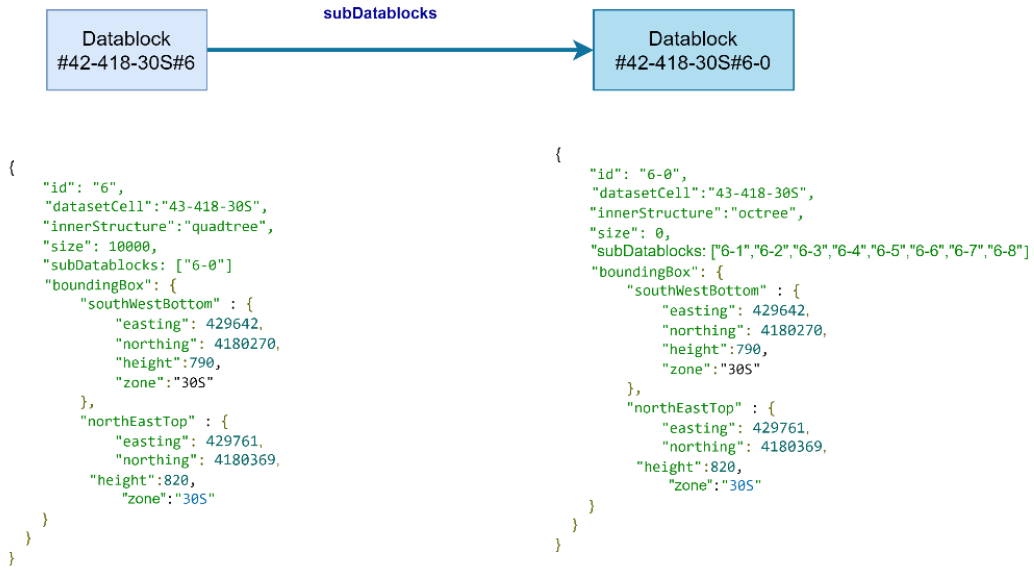


Figura 3.19. Representación de los *datablocks* presentes en la frontera entre dos estructuras anidadas (quadtree y octree). El *datablock* asociado al nodo raíz de la estructura indexada no tiene puntos asociados y actúa como un mecanismo de enlace.

Esquemas SPSLiDAR 1.1

Una de las características de SPSLiDAR 1.1 es la posibilidad de exponer la información de las estructuras de datos que implementa un repositorio para permitir a los clientes seleccionar la que más se adecúa a las características de los modelos a importar. Para poder definir estas estructuras a través de SPSLiDAR 1.1 es necesario generar una instancia de las clases definidas en el modelo. En esta sección se muestran los esquemas utilizados para las distintas estructuras que hemos presentado. Se ha utilizado una representación en formato JSON para documentarlas, similar a la respuesta esperada en caso de llamar al servicio del repositorio que expone esta información. Estos documentos contienen información sobre los identificadores lógicos de la estructura dentro del sistema y sus características fundamentales. En el campo *parameters* se guardan los parámetros de la estructura, que podrá sobrescribir el cliente con valores personalizados; en caso de no definirse, utiliza el valor por defecto indicado. El campo *cell* expone un objeto que contiene el detalle de los nodos que construirá la estructura, indicando los parámetros que se utilizan para identificar unívocamente cada nodo dentro del modelo. Esta información puede ser utilizada para realizar un análisis preliminar que ayude a interpretar cómo se distribuirá la información en la estructura y ayudar a la toma de decisiones sobre cuál es la más adecuada en cada contexto.

Workspace-Dataset

Grid disperso UTM 2 niveles

Esta estructura cuenta con un único parámetro para determinar el tamaño de celda del segundo nivel del grid. Cada celda del grid viene identificada por su zona UTM y los identificadores en los ejes X e Y.

```
{
    "workspaceDataStructureId" : 1,
    "workspaceDataStructureName" : "Grid disperso UTM de 2 niveles" ,
    "workspaceDataStructureDescription" : "Estructura de datos basada en un grid de dos niveles, utilizando el sistema de zonas UTM - MGRS en la primera capa y un grid disperso en la segunda." ,
    "parameters" : [
        {
            "name": "gridCellSize" ,
            "description" : "Define el tamaño de celda a utilizar en el segundo nivel de la estructura. Valor especificado en metros" ,
            "defaultValue" : 10000,
            "type" : "integer"
        }
    ],
    "cell" : {
        "cellId" : "UTMZone xCoordId - yCoordId" ,
        "parameters" : [
            {
                "name": "UTMZone",
                "description" : "ID de la zona UTM" ,
                "type" : "string"
            },
            {
                "name": "xCoordId" ,
                "description" : "ID de la celda en el eje X" ,
                "type" : "integer"
            },
            {
                "name": "yCoordId" ,
                "description" : "ID de la celda en el eje Y" ,
                "type" : "integer"
            }
        ]
    }
}
```

Grid disperso UTM de N niveles

Sigue una filosofía similar al anterior, pero utiliza el atributo *gridCellSizes*, un array que se ordena a nivel interno para definir los tamaños de celda para cada nivel del grid.

```
{
   "workspaceDataStructureId"      :   2,
   "workspaceDataStructureName"    :   "Grid disperso UTM de N niveles"          ,
   "workspaceDataStructureDescription"         :   "Estructura de       datos basada en un grid de dos niveles,
utilizando el sistema de zonas UTM           - MGRS en la primera capa y un grid disperso en la segunda."                ,
   "parameters"   :   [
      {
         "name":   "gridCellSizes"      ,
         "description"      :   "Define el array de tamaños de celda a utili                zar en el segundo nivel de
la estructura. Valores especificados en metros"                ,
         "defaultValue"     :   [ 1000,   10000,   100000],
         "type"   :   "[] : integer"
      }
   ],
   "cell"    :   {
      "cellId"     :   "UTMZone xCoordId - yCoordId - cellLevel"     ,
      "parameters"   :   [
         {
            "name":   "UTMZone",
            "description"      :   "ID de la zona UTM"     ,
            "type"   :   "string"
         },
         {
            "name":   "xCoordId"   ,
            "description"      :   "ID de la celda en el eje X"           ,
            "type"   :   "integer"
         },
         {
            "name":   "yCoordId"   ,
            "description"      :   "ID de la celda en el eje Y"           ,
            "type"   :   "integer"
         },
         {
            "name":   "cellLevel"     ,
             "description"     :   "Nivel de la celda. Los niveles utilizan el identificador numérico
del tamaño con el que fueron          construidos."     ,
            "type"   :   "integer"
         }
      ]
   }
}
```

Grid disperso temporal

Estructura utilizada para llevar a cabo indexación de *datasets* en base a la dimensión temporal. Indexa los *datasets* en una única dimensión, utilizando un tamaño de ventana temporal especificado en horas para definir nodos.


```
{
   "workspaceDataStructureId"      :   3,
   "workspaceDataStructureName"    :   "Grid disperso temporal"        ,
   "workspaceDataStructureDescription"         :   "Estructura de       datos basada en un grid unidimensional
basado en la línea temporal con origen de datos definido en tiempo UNIX"                        ,
   "parameters"   :   [
      {
         "name":   "temporalCellSize"      ,
        "description"      :   "Define el tamaño de la ventana temporal de las celdas del grid.                    Definido
en horas."    ,
         "defaultValue"     :   720,
         "type"   :   "integer"
      }
   ],
   "cell"    :   {
      "cellId"     :   "temporalCell"     ,
      "parameters"   :   [
         {
            "name":   "temporalCell"     ,
            "description"      :   "ID generado para la celda."          ,
            "type"   :   "integer"
         }
      ]
   }
}
```

Dataset-Datablock

Octree

El octree define dos atributos para su parametrización. El primero es *datablockSize*, que es equivalente al campo homónimo que estaba presente en la entidad *dataset* en la versión 1.0 y que ha sido eliminado en la versión 1.1 de la especificación para ser definido como un parámetro más de la estructura. También se define un parámetro *maxDepth* que determina el tamaño máximo del octree y el parámetro *redundancy* que indica si se utiliza una estrategia con redundancia o no (un punto presente en el nodo padre puede aparecer también en sus nodos hijos o no). También se podrían añadir parámetros adicionales para poder elegir entre distintas estrategias de muestreo.

```
{
   "datasetDataStructureId"       :  1,
   "datasetDataStructureName"     :  "Octree"  ,
    "datasetDataStructureDescription"        :   "Estructura  los  datablocks  en  base  a  un  octree
regular."    ,
   "parameters"  :  [
      {
         "name":  "maxDepth" ,
         "description"     :  "Define el nivel de profundidad máximo              permitido en el octree"        ,
         "type"  :  "integer"    ,
         "defaultValue"     :  8
      },
      {
         "name":  "redundancy"  ,
         "description"     :  "Define si los puntos presentes en un nodo pueden también aparecer en
sus hijos. "      ,
         "type"  :  "boolean"  ,
         "defaultValue"     :  true
      },
      {
         "name":  "datablockSize"     ,
         "description"     :  "Define el número máximo de puntos por nodo."           ,
         "type"  :  "integer"    ,
         "defaultValue"     :  10000
      }
   ],
   "cell"   :  {
      "cellId"    :  "#octreeId   - #datasetCell"    ,
      "parameters"  :  [
         {
            "name":  "octreeeId"    ,
            "description"     :  "Identificador numérico asociado al nodo del octree"                ,
            "type"  :  "integer"
         },
         {
            "name":  "datasetCell"     ,
            "description"     :  "Indica la celda del dataset             a la que pertenece el octree."          ,
            "type"  :  "integer"
         }
      ]
   }
}
```

Grid disperso

Al igual que en el caso del octree, esta estructura cuenta con un parámetro *pointNumber*. En la descripción de esta estructura indicamos que puede ser implementada tanto bajo un modelo no jerárquico como uno jerárquico compuesto por varios niveles. Esta estructura pretende recoger una definición válida para ambos casos. El parámetro denominado *gridCellSizes* permite definir un array con los distintos tamaños de celda: el valor por defecto define un único tamaño, lo cual resulta en una estructura no jerárquica. Si se define más de un tamaño, se crearán niveles de detalle y relaciones entre *datablocks* siguiendo lo especificado en el detalle de la implementación.

```
{
   "datasetDataStructureId"       :  2,
   "datasetDataStructureName"     :  "Grid disperso"      ,
    "datasetDataStructureDescription"          :  "Estructura basada en un grid disperso, opcionalmente
multinivel."      ,
   "parameters"   :  [
      {
         "name":  "gridCellSizes"      ,
        "description"     :  "Define los tamaños de celda a utilizar en el grid. El número de niveles
será equivalente tamaño del array."            ,
         "type"  :  "[] integer"       ,
         "defaultValue"     :  [ 100]
      },
      {
         "name":  "datablockSize"     ,
         "description"     :  "Define el número máximo de puntos por nodo."           ,
         "type"  :  "integer"    ,
         "defaultValue"     :  10000
      }
   ],
   "cell"    :  {
      "cellId"     :  " #gridCellId     - gridLevel    - #datasetCell"     ,
      "parameters"   :  [
         {
            "name":  "gridCellId"      ,
            "description"     :  "Identificador numérico asociado al nodo del octree"                ,
            "type"  :  "integer"
         },
         {
            "name":  "gridLevel"     ,
              "description"     :  "Identificador numérico asociado al nivel del grid                . Se rá nulo si
cuenta   con un único nivel.       ",
            "type"  :  "integer"
         },
         {
            "name":  "datasetCell"     ,
            "description"     :  "Indica la celda del dataset a la que pertenece el octree."                    ,
            "type"  :  "integer"
         }
      ]
   }
}
```

Quadtree-octree anidado

La última de las estructuras expuestas se trata de un ejemplo de estructura anidada, utilizando un quadtree y un octree. El parámetro *quadtreeDepth* define la profundidad máxima de la estructura y el nivel a partir del cual se procederá a distribuir los puntos bajo un octree. Los identificadores de las celdas son fruto de la combinación de los identificadores parciales de cada una de las estructuras definidas, tanto la actual como sus superiores en la jerarquía. Nótese que frente a la representación de ejemplo presentada en la Figura 3.19, el campo *datasetCell* de la estructura *Datablock* se omite y se incluye dentro del propio del *id*, correspondiendo al parámetro homónimo *datasetCell*.

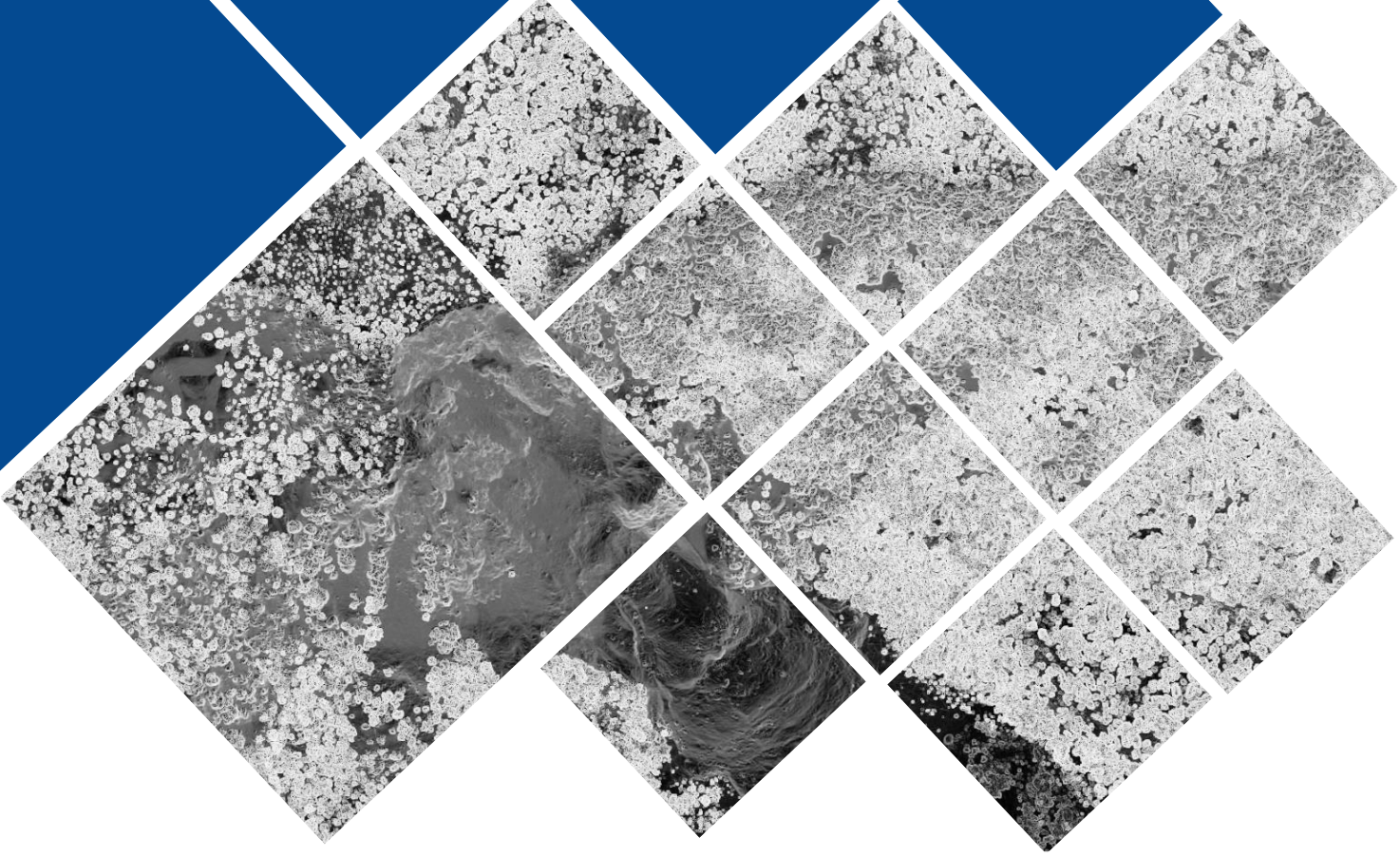
```
{
  "datasetDataStructureId" : 3,
  "datasetDataStructureName" : "Quadtree - Octree anidado" ,
   "datasetDataStructureDescription" : "Estructura anidada compuesta por un primer nivel
modelado por un quadtree a partir del cual se podrán desarrollar octrees." ,
  "parameters" : [
    {
      "name": "quadtreeMaxDepth" ,
       "description" : "Define el nivel de profundidad máximo permitido en el quadtree.
Establece la frontera a partir de la cual se procederá a construir un octree." ,
      "type" : "integer" ,
      "defaultValue" : 4
    },
    {
      "name": "octreeMaxDepth" ,
       "description" : "Define el nivel de profundidad máximo permitido en el quadtree.
Establece la frontera a partir de la cual se procederá a construir un octree." ,
      "type" : "integer" ,
      "defaultValue" : 4
    },
    {
      "name": "datablockSize" ,
      "description" : "Define el número máximo de puntos por nodo." ,
      "type" : "integer" ,
      "defaultValue" : 10000
    }
  ],
  "cell" : {
    "cellId" : "#quadtreeId - octreeId - #datasetCell" ,
    "parameters" : [
      {
        "name": "quadtreeId" ,
        "description" : "Identificador numérico asociado al nodo del octree" ,
        "type" : "integer"
      },
      {
        "name": "octreeeId" ,
        "description" : "Identificador numérico asociado al nodo del octree" ,
        "type" : "integer"
      },
      {
        "name": "datasetCell" ,
        "description" : "Indica la celda del dataset a la que pertenece el octree." ,
        "type" : "integer"
      }
    ]
  }
}
```

3.5. Conclusiones

Con el fin de demostrar la viabilidad de SPSLiDAR como solución para implementar repositorios orientados al procesado y almacenamiento de nubes de puntos, en este capítulo se han presentado las bases para el desarrollo de una aplicación adecuada a dicha finalidad. Se ha propuesto una solución basada en un stack tecnológico conformado por Java y Spring Webflux y centrado principalmente en el tratamiento de información en los formatos LAS/LAZ, utilizando LasTools para el procesado de estos datos.

A lo largo del capítulo se han definido distintas estructuras de datos, especificando de forma más concreta cómo se lleva a cabo la integración de estas con el estándar SPSLiDAR y las particularidades técnicas a tener en cuenta dentro de las tecnologías seleccionadas. El resultado de este proceso es un amplio abanico de opciones que permiten organizar la información del modelo de manera que se optimicen consultas a nivel espacial o temporal y se estructure la información en base a modelos jerárquicos y no jerárquicos.

Si cada una de estas estructuras puede ser implementadas en la versión base de SPSLiDAR, también se describe en este capítulo su definición JSON dentro de SPSLiDAR 1.1, ilustrando las ventajas de ofrecer todas las opciones presentadas en una única implementación.



Capítulo 4 - SISTEMAS DE ALMACENAMIENTO Y ESCALABILIDAD EN EL CONTEXTO DE SPSLIDAR

4.1. Introducción

A lo largo de los capítulos anteriores hemos presentado el modelo de datos de SPSLiDAR y distintas estrategias y estructuras de datos para su implementación. Una parte importante de esta es la definición de una capa de almacenamiento escalable, resiliente y con alta disponibilidad, que permita mantener la información de manera segura y resolver consultas de forma rápida y eficiente.

Son múltiples las cuestiones a abordar con respecto al almacenamiento de nubes de puntos. Una de las más importantes es la decisión del enfoque a utilizar para su almacenamiento. Por un lado, en la literatura encontramos métodos que almacenan esta información en forma de ficheros. Los puntos quedan registrados en archivos basados en un formato estándar para nubes de puntos, como LAS. El segundo enfoque que encontramos implica almacenar de forma individualizada cada punto de la nube, normalmente en un sistema de base de datos y utilizando un esquema propio o bien alguno ya definido de forma nativa en la base de datos elegida. La entidad utilizada para representar cada punto puede tener un número variable de atributos, pero deberá contar como mínimo con las coordenadas del punto en el sistema de coordenadas, cartesiano o esférico, en el que está definida la nube. Es posible registrar otros atributos: el color, la intensidad, el número de retorno, la clasificación o cualquier otro que puedan obtenerse en un escaneado LiDAR, o eventualmente tras cierto posprocesamiento (p. ej. una estimación de la normal). El tener todos estos atributos accesibles dentro del modelo nos ofrece la capacidad de realizar operaciones de filtrado muy detalladas y precisas. También facilita la descarga selectiva de estos atributos, que pueden ser especificados en la propia consulta. Por ejemplo, un punto dentro de la versión 1.4 de la especificación LAS puede definir hasta 26 atributos distintos y llegar a alcanzar un tamaño total de 67 bytes (ASPRS, 2019).

La especificación de SPSLiDAR propuesta en el **Capítulo 2** utiliza un enfoque basado en ficheros, utilizando la entidad *Datablock* para representar sus características fundamentales. En este capítulo se desarrollará una solución basada en la misma filosofía, almacenando estos archivos construidos durante las etapas de procesamiento vistas en el **Capítulo 3** a través de las cuáles se generan los *datablocks* que organizan la nube de puntos asociada a un *dataset*. Si bien es posible desarrollar una solución con SPSLiDAR en la que el almacenamiento se haga a nivel de punto, extrayendo de cada fichero obtenido sus puntos y volcándolos en una base de datos, la decisión de utilizar ficheros implica ventajas significativas en nuestro contexto. A nivel de almacenamiento, los datos se registran utilizando un formato específico y altamente optimizado reduciendo el espacio necesario; para un repositorio destinado a albergar una gran volumetría de modelos, esto es una ventaja significativa a nivel de recursos necesarios. Esto también se aplica a la transmisión de la información a través de la red, con un tamaño más reducido de las respuestas. También garantiza que la información recibida por el cliente esté definida en un estándar popular y ampliamente soportado, sin necesidad de tener que adaptarse a un nuevo esquema específico para su integración con SPSLiDAR. A nivel de eficiencia de las consultas, estas son más sencillas y rápidas de resolver, puesto que las operaciones de filtrado se realizan sobre un conjunto de datos definido por la volumetría de la entidad *datablock*, que será mucho menor al conjunto de puntos total.

El principal inconveniente de este modelo es el coste en tiempo de computación necesario para llevar a cabo operaciones de filtrado sobre los puntos, ya sea con criterios espaciales o usando atributos adicionales. Su implementación requiere abrir el fichero e iterar sobre todos los puntos,

descartando aquellos que no cumplan los criterios, para finalmente generar un nuevo fichero temporal con el subconjunto de puntos resultante, que se enviaría a la aplicación cliente. Debido al alto coste y la complejidad del procesamiento necesario tomamos la decisión de no desarrollar el soporte de este tipo de filtrados, a nivel de puntos, en nuestra implementación de referencia.

Cualquier operación de consulta sobre la nube de puntos utiliza únicamente la información aportada a nivel de *datablocks*, (p. ej. la caja envolvente del mismo) para decidir si los puntos del fichero asociado se adecúan a ella. Es evidente que no todos los puntos del fichero asociado al *datablock* tienen por qué ajustarse a la consulta: una consulta de ventana puede intersectar parcialmente la caja envolvente del *datablock*, lo que va a generar un conjunto más o menos grande de falsos positivos. Esta idea de falsos positivos ya se mencionó en el **Capítulo 3**. Sin embargo, utilizando algoritmos que generen *datablocks* con alta cohesión, teniendo en cuenta las altas velocidades de red disponibles hoy en día, y la eficiencia de formatos como LAZ para codificar los modelos de nube de puntos, se ha tomado la decisión de implementar el envío de estos ficheros de forma completa tal y cómo están almacenados en el sistema, sin aplicar ningún tipo de posprocesamiento que implique un incremento en los tiempos de respuesta. La aplicación cliente podrá decidir qué hacer con los puntos extra recibidos: descartarlos en su lógica de procesamiento o cachearlos en caso de que puedan ser necesarios a continuación. Este segundo caso es especialmente interesante ya que permite evitar invocaciones adicionales al repositorio.

El tamaño de las nubes que se generen (ligado al atributo *datablockSize* de la entidad *Dataset*) es un parámetro crítico en este proceso. Valores demasiado grandes implican ficheros de mayor tamaño, incrementando los tiempos de transmisión a través de la red con puntos que pueden no ajustarse a la consulta realizada. Valores demasiado pequeños repercutirán en un mayor número de *datablocks* necesarios para satisfacer una consulta, aunque el ajuste del resultado a esta consulta puede ser mejor, es decir, el número de falsos positivos será menor. Sin embargo, la necesidad de procesar más *datablocks* va a ralentizar los tiempos de respuesta y obligar a un número más elevado de peticiones concurrentes que pueden suponer problemas de rendimiento en el servidor. En este capítulo se evaluará el rendimiento del sistema en base a diferentes tamaños de puntos.

Una vez definido el tipo de información que almacena el sistema, queda establecer las tecnologías utilizadas para ello. Ante la volumetría de datos esperada y la necesidad de escalar fácilmente, los modelos de almacenamiento distribuidos son los más interesantes. Dentro de este paradigma distribuido existen múltiples sistemas de almacenamientos de ficheros, siendo quizás el más popular Hadoop Distributed File System (HDFS) y sobre el cual ya se han llevado a cabo implementaciones para almacenar modelos de nube de puntos (C. Wang et al., 2017). Otra alternativa son los sistemas de base de datos. Muchos de ellos cuentan con mecanismos nativos para ser desplegados en entornos distribuidos, permitiendo la escalabilidad del sistema a través de la incorporación de nodos. Algunos casos como MongoDB definen una especificación propia, diseñada especialmente para el almacenamiento de ficheros en entornos distribuidos y sobre la cual ya hay propuestas basadas en el uso de ficheros LAS (Boehm & Liu, 2015). En este capítulo nos centraremos particularmente en el uso de bases de datos, tanto de tipo relacional como NoSQL, ya que son más versátiles y permiten definir un mismo estándar para la recuperación, tanto de los metadatos que genera el esquema SPSLiDAR, como para el almacenamiento de los propios ficheros de nubes de puntos. En este capítulo se presenta una evaluación en materia de rendimiento siguiendo las pautas definidas en los capítulos anteriores.

Los resultados del estudio que se presenta a continuación se resumen en una publicación del autor de esta memoria (Béjar-Martos et al., 2022), donde se evalúa el rendimiento de cuatro populares sistemas de gestión de bases de datos con grandes conjuntos de datos LiDAR: Cassandra, MongoDB, MySQL y PostgreSQL. Para realizar una evaluación realista, integramos estos motores de bases de datos en una implementación de SPSLiDAR para poder realizar consultas espaciales y de metadatos, así como la recuperación progresiva/parcial de datos. Nuestra experimentación concluye que, como se esperaba, las bases de datos NoSQL muestran una diferencia de rendimiento modesta pero significativa a su favor, y que Cassandra proporciona la mejor solución de base de datos general para datos LiDAR.

4.2. Planteamiento de los experimentos

Para estudiar el rendimiento de los diferentes motores de bases de datos integrados en SPSLiDAR se utilizaron diversas nubes de puntos pertenecientes a la ciudad de Pamplona (España). Elegimos esta área en particular debido a la disponibilidad pública de datos adquiridos en diferentes momentos y densidades (0,5-10 puntos/m²). Las áreas cubiertas por los conjuntos de datos LiDAR están ubicadas en la zona UTM 30N y han sido capturadas durante 6 años, incluyendo una variedad de terreno (urbano, rural, caminos, etc.). La Tabla 4.1 resume las características de los cuatro conjuntos de datos, y la Figura 4.1 muestra una representación parcial del Dataset 1.

	Dataset 1	**Dataset 2**	**Dataset 3**	**Dataset 4**
Name	Pamplona 2011	Pamplona2017b (reduced version of dataset 4)	Pamplona 2017c (reduced version of dataset 4)	Pamplona 2017
Number of points	30 401 539	244 894 563	534 073 172	1 068 146 345
Density (points/m^2)	0.5	2.5	5	10
Grid size (meters)	10 000	10 000	10 000	10 000
Size (MBs)	138.29	1 850.21	4 085.94	8 054.36
Point Data Record Length	34	38	38	38
LAS Point Data Record Format	3	8	8	8

Tabla4.1. Características de los *datasets* usados en los experimentos.

Figura4.1. Una vista parcial del Dataset(Pamplona2011)

Utilizamos la implementación de referencia de SPSLiDAR que utiliza una cuadrícula regular para organizar los *datasets* y un octree para los datablocks dentro de cada *dataset*. El detalle de estas estructuras quedó presentado en el **Capítulo 3** de esta tesis. Las características del sistema de gestión de bases de datos tienen una influencia significativa en el rendimiento de SPSLIDAR, ya que cada vez que se envía un *dataset* al sistema se generan uno o más octrees, y cada uno puede comprender cientos de miles de nodos y archivos LAZ que deben ser almacenados eficientemente en la base de datos, tanto en términos de tiempo como de espacio ocupado. Además, la API SPSLiDAR permite la navegación por los octrees del *dataset* y la recuperación de las nubes de puntos en los archivos LAZ asociados a los nodos de interés. La capacidad de la base de datos para recuperar los *datablocks* deseados y transmitir el contenido de los archivos de nube de puntos asociados tiene un impacto directo en los tiempos de latencia y el rendimiento del repositorio, considerando también que normalmente varios clientes pueden interactuar con el sistema al mismo tiempo.

Bases de datos

En nuestra experimentación utilizamos la implementación original del SPSLiDAR basada en MongoDB y tres adaptaciones diferentes, correspondientes al resto de sistemas de gestión de bases de datos evaluados. Aunque el almacenamiento de grandes *datasets* LiDAR se ajusta a la definición de Big Data Geoespacial, no queríamos limitarnos únicamente a bases de datos no relacionales, generalmente consideradas la solución de almacenamiento estándar para aplicaciones en este dominio. Las bases de datos relacionales también se han utilizado para el almacenamiento de cantidades masivas de datos (Pääkkönen & Pakkala, 2015), y por tanto se ajustan a las necesidades de nuestro caso de uso.

MongoDB, Cassandra, PostgreSQL y MySQL han sido las cuatro tecnologías adoptadas. Todos ellos son productos consolidados y maduros. Entre las bases de datos no relacionales, MongoDB proporciona una solución orientada a documentos, mientras que Cassandra sigue un enfoque de columnas anchas. En la categoría de bases de datos relacionales, PostgreSQL y MySQL se encuentran entre las más populares. Para realizar la comparación más justa, el modelo conceptual SPSLiDAR original se ha implementado en cada base de datos de la forma más sencilla.

MongoDB

Es una base de datos no relacional orientada a documentos que utiliza JSON para codificar información. Los documentos en MongoDB presentan un esquema flexible, de modo que dos documentos que representan un concepto similar puedan tener campos diferentes y se puedan agregar o eliminar nuevos atributos en tiempo de ejecución. MongoDB introduce el concepto de colección como un medio para agrupar documentos que representan el mismo concepto o tipo de información. Proporciona además la especificación GridFS como una forma de almacenar archivos binarios de un tamaño arbitrario más allá del límite de 16 MB por documento impuesto por MongoDB. GridFS usa dos colecciones para almacenar archivos. Una colección almacena los fragmentos de archivos (*fs.chunks*) y la otra almacena metadatos de archivos (*fs.files*). Cada archivo se descompone en múltiples fragmentos almacenados en la colección *fs.chunks*, cada uno de los cuales contiene los datos binarios de la sección correspondiente del archivo, un atributo de orden que especifica su posición en la secuencia de fragmentos y una referencia al documento de archivo en la colección *fs.files*. De forma predeterminada, la cantidad máxima de datos por fragmento está establecida en 255 KB. Las colecciones de GridFS se muestran en la Figura 4.2.

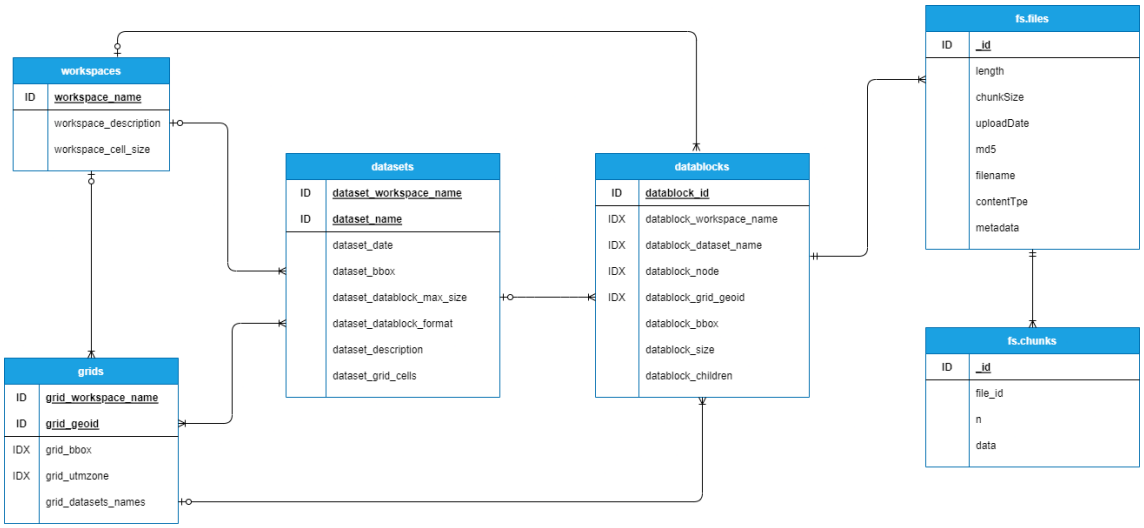


Figura4.2. Diseño de base de datos para MongoDB. PK se refiere a clave primaria y FK a clave externa.

El almacenamiento de archivos LAZ en la base de datos es uno de los puntos clave de la implementación, ya que tiene un efecto directo sobre el rendimiento. Desde un punto de vista técnico, podríamos guardar estos archivos a través de GridFS o incrustar el contenido en un documento de forma binaria. Decidimos utilizar GridFS como forma estándar para almacenar

archivos, ya que esto permite la construcción de octrees con un tamaño de nodo arbitrario. También se podría seguir un enfoque híbrido, almacenando el archivo LAZ a través de un campo incrustado si no supera el límite de tamaño de 16 MB para documentos MongoDB, o mediante GridFS en caso contrario. Este enfoque híbrido se puede implementar fácilmente en MongoDB gracias a su esquema de documento flexible, que permite codificar los datos del archivo en un campo de tipo *binData* o, alternativamente, incluir un campo *DBRef* con el identificador del documento guardado en la colección *fs.files* de GridFS.

La estructura final de documentos y colecciones definidas en MongoDB se muestra en la Figura 4.2, que especifica las relaciones entre las diferentes entidades. Se siguió un enfoque principalmente desnormalizado: por ejemplo, un documento de *datablock* contiene información redundante que hace referencia al espacio de trabajo, al *dataset* y a la celda de la cuadrícula a la que pertenece. Estos campos se han indexado para acelerar las consultas, ya que la recuperación de bloques de datos es una operación crucial para acceder a los archivos de datos LAZ.

Cassandra

Es una base de datos no relacional con un modelo de filas particionadas. Las filas se almacenan en diferentes particiones, identificadas por la clave de partición (PK) que puede constar de una o más columnas. Una partición puede contener más de una fila; en estos casos también se necesita una clave de clúster (CK), que también puede constar de más de una columna e identifica de forma única cada fila dentro de una partición. Por lo tanto, una única fila se define mediante una clave compuesta formada por una clave de partición y, opcionalmente, una clave de clúster.

No existe ninguna especificación análoga a GridFS en Cassandra, por lo tanto, implementamos un sistema personalizado con la misma estructura. Se ha definido una nueva entidad llamada *chunk*, que contiene una columna con datos binarios y se identifica de forma única mediante una clave compuesta formada por la clave de partición del *datablock* al que pertenece y un atributo adicional *chunk_order*, que identifica la posición en la secuencia de fragmentos en los que está dividido el archivo. Como resultado, todos los fragmentos que pertenecen al mismo archivo se almacenan en la misma partición y el atributo *chunk_order* facilita una recuperación completa y ordenada de los fragmentos del archivo. El tamaño máximo de contenido para cada fragmento se ha establecido en 255 KB, el mismo valor predeterminado utilizado por GridFS.

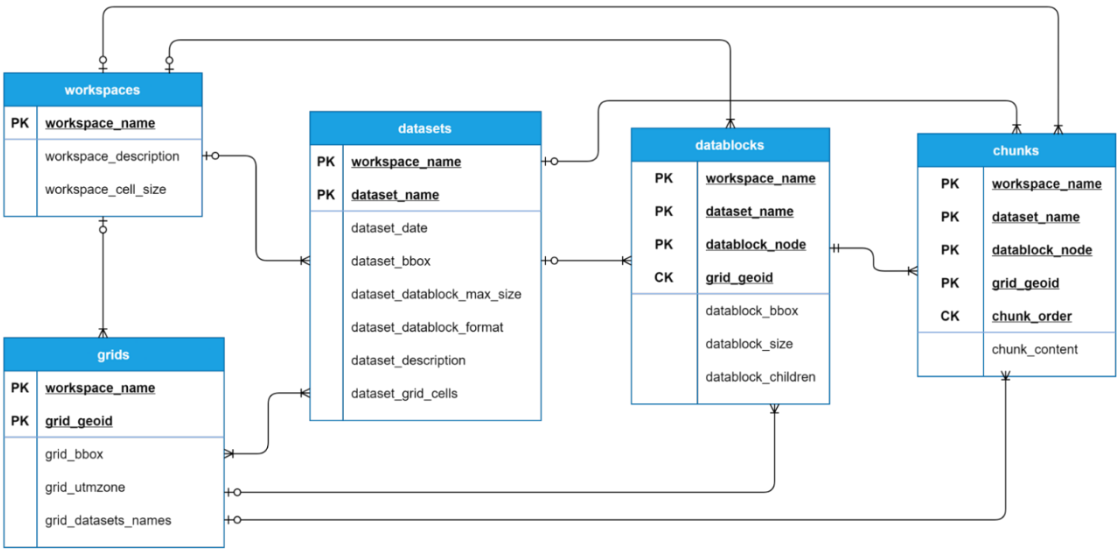


Figura4.3. Diseño de base de datos para implementación de Cassandra. PK se refiere a clave primaria y FK se refiere a claveæxterna.

La Figura 4.3 muestra las tablas y las columnas definidas para la persistencia del modelo de datos en Cassandra. Como en MongoDB, se siguió un enfoque desnormalizado. Cassandra recomienda un diseño de modelo basado en las consultas a realizar, fomentando la duplicación de datos para aumentar la eficiencia de la lectura de datos.

Bases de datos relacionales

En cuanto a bases de datos relacionales, se han implementado dos alternativas diferentes basadas en PostgreSQL y MySQL. Debido a sus similitudes, utilizamos casi la misma implementación en la capa de persistencia, modificando solo los tipos para que se ajusten a la especificación de cada base de datos.

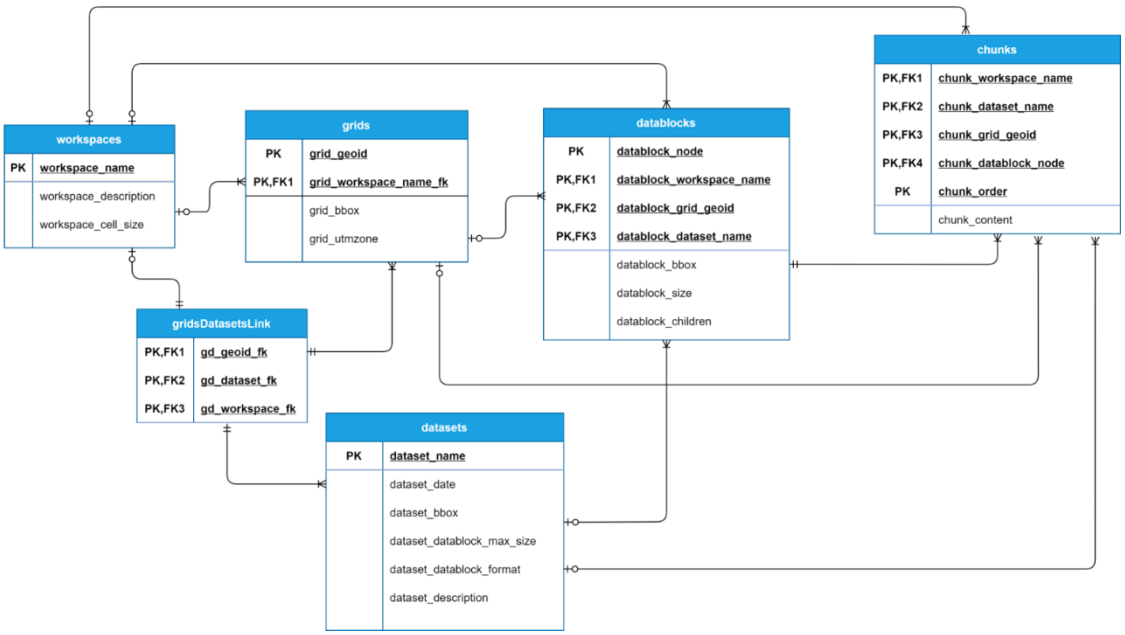


Figura4.4. Diseño de bases de datos para implementaciones MySQL y PostgreSQL. PK se refiere a clave primaria y FK se refiere a clave externa.

Al igual que con Cassandra, implementamos una solución a medida para el almacenamiento de archivos. La solución propuesta define una nueva entidad llamada *chunk*, que representa un bloque del archivo asociado al *datablock*, identificado por una clave primaria compuesta formada por la clave primaria del *datablock*, junto a un atributo adicional que representa su posición en la secuencia de bloques binarios que forma el archivo. La normalización suele ser un requisito en los esquemas de bases de datos relacionales, pero para evitar uniones de tablas que puedan afectar negativamente al rendimiento, se desnormalizan tanto los *datablocks* como los *chunks*. Consideramos que la mejora del rendimiento compensa las desventajas de un esquema desnormalizado para estas tablas en particular, ya que se espera que contengan una gran cantidad de filas y soporten una gran cantidad de consultas. En la Figura 4.4 se muestra un diagrama entidad-relación del modelo para ambas bases de datos relacionales.

4.3. Experimentación y resultados

Se realizaron tres experimentos con objeto de evaluar el rendimiento de cada una de las bases de datos con grandes *datasets* LiDAR. Se basan en los propuestos en (Van Oosterom et al., 2015). La primera prueba (carga) evalúa el tiempo de carga y el espacio de almacenamiento total requerido por los cuatro conjuntos de datos de prueba. La segunda prueba (solicitudes simples) mide el tiempo promedio de descarga de 1.000.000 de puntos desde un solo cliente. Para ello se solicitan múltiples archivos LAZ aleatorios hasta completar ~20.000.000 puntos, calculando el promedio del tiempo total de descarga. La tercera prueba (solicitudes concurrentes) evalúa la respuesta del sistema a cargas de trabajo elevadas, midiendo el tiempo de descarga promedio para múltiples solicitudes concurrentes de 1.000.000 de puntos desde varios clientes.

La prueba de carga y la prueba de solicitudes simples se implementaron a través de un script para Python 3.9 que utiliza la biblioteca Requests (Reitz, 2023) para realizar las peticiones al servidor. La prueba de solicitudes concurrentes se desarrolló utilizando el paquete JavaScript LoadTest para Node.js (Fernandez, 2021). Todas las operaciones se realizaron a través de la API SPSLiDAR. Los datos probados corresponden a los cuatro *datasets* presentados y descritos en la Tabla 4.1. Para cada uno de los *datasets* se probaron tres tamaños máximos de *datablock* diferentes: 10.000 , 100.000 y 1.000.000 de puntos. Estos valores definen el número máximo de puntos del archivo LAZ almacenados en cada nodo del octree, y en consecuencia determinan su complejidad, ya que se requeriría un número mayor o menor de niveles para almacenar el dataset.

Todos los experimentos se llevaron a cabo ejecutando las diferentes implementaciones del repositorio SPSLiDAR en un servidor con Intel i7-10700 Octa-Core a 2,9 GHz, 16 GB de RAM y unidad SSD de 1TB, ejecutando Windows 10 Enterprise 64. Todas las bases de datos se implementaron a través de Docker usando imágenes con MongoDB 4.2.10, Cassandra 3.11.0, MySQL 8.0.24 y PostgreSQL 13.2. El cliente era un portátil Macbook Pro Intel Core i7-4770HQ Quad-Core a 2,2 GHz, 16 GB de RAM, SSD de 256 GB y macOS Big Sur. La conexión entre el servidor y el cliente se realizó a través de una red de cable Ethernet de 1 GB/s.

Test de subida

La carga de un *dataset* consta de dos etapas. Primero, el *dataset* original se preprocesa y se subdivide en un conjunto de pequeños archivos LAZ cuyo tamaño está limitado por el tamaño máximo del *datablock*. Estos archivos se indexan mediante octrees que representan la nube de puntos original. En segundo lugar, los archivos y los metadatos del octree se almacenan en la base de datos. La primera etapa es muy exigente desde el punto de vista computacional y consume la mayor parte del tiempo. Por lo tanto, la influencia del rendimiento de la base de datos en el tiempo total es limitada. La etapa de preprocesamiento es externa a la capa de persistencia, y de carácter determinista, lo que resulta en la inserción de una misma cantidad de archivos en cada base de datos. Además, hay que tener en cuenta que en la práctica esta operación de carga solo debe realizarse una vez por *dataset*. Por este motivo, consideramos que los resultados de esta prueba no son los más determinantes para la elección de la base de datos.

Dataset	Maximum datablock size	MongoDB	Cassandra	PostgreSQL	MySQL
Dataset 1	1.000.000	67	68	69	70
	100.000	218	211	224	213
	10.000	1444	1373	1553	1411
Dataset 2	1.000.000	746	793	824	727
	100.000	2089	2027	2016	1993
	10.000	13962	12791	12650	12942
Dataset 3	1.000.000	1945	1806	1899	2191
	100.000	3906	3843	3897	3962
	10.000	25176	24528	26823	23307
Dataset 4	1.000.000	4131	4220	4150	4816
	100.000	8460	8258	8887	8575
	10.000	45035	41290	49714	42018

Tabla4.2. Tiempos de carga de *datasets* (en segundos) para las implementaciones basadas en MongoDB, Cassandra, PostgreSQL y MySQL.

La Tabla 4.2 muestra los tiempos promedio de carga para cada combinación de *dataset* y tamaño máximo de *datablock* tras tres ejecuciones. Las mayores diferencias entre las bases de datos se producen en el Dataset 4, ya que la estructura de octree generada es la más compleja y con la mayor cantidad de archivos LAZ para almacenar. La Figura 4.5 muestra una comparación relativa de los tiempos de carga de Cassandra, PostgreSQL y MySQL con respecto a MongoDB. Usamos MongoDB como referencia por formar parte de la implementación estándar de SPSLiDAR descrita en el capítulo anterior.

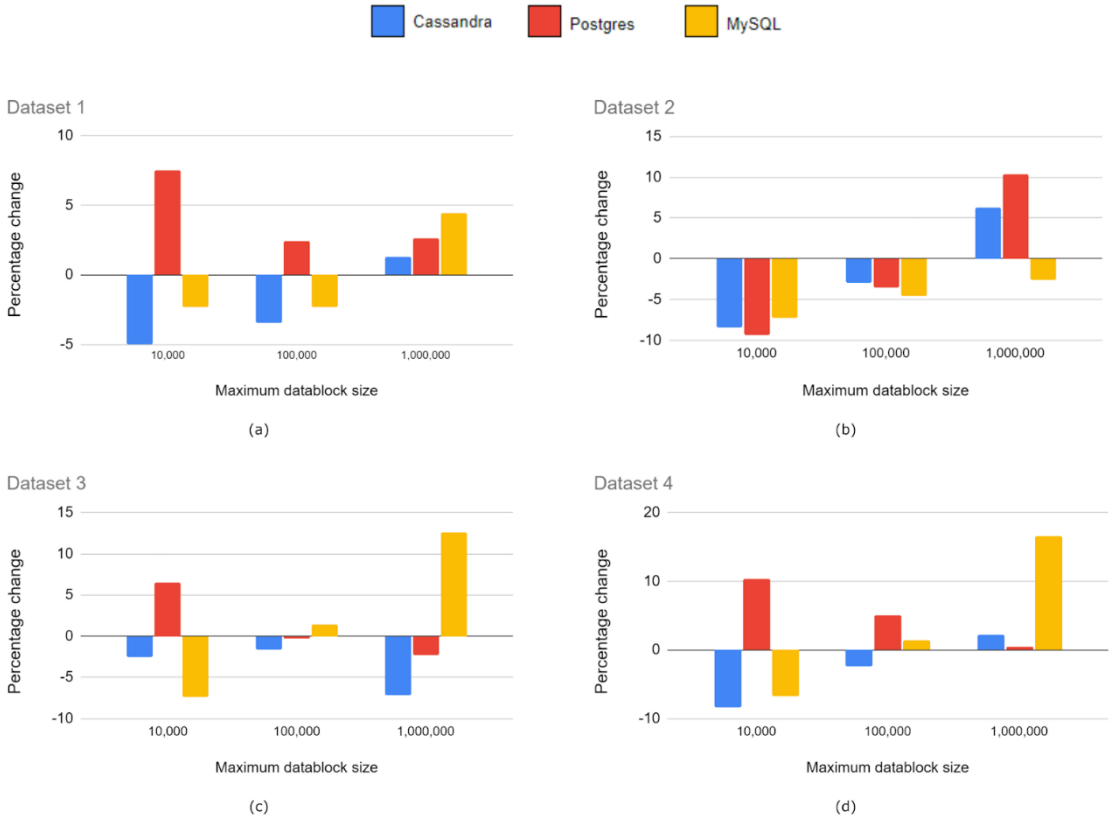


Figura 4.5. Comparación relativa de tiempos de carga. Resultados para (a) Dataset 1; (b) Dataset 2; (c) Dataset 3; y (d) Dataset 4.

La Tabla 4.3 muestra el espacio de almacenamiento final requerido a nivel de base de datos por los conjuntos de datos después de las pruebas de carga. Esto incluye los archivos LAZ generados en el proceso para que todas las entidades necesarias para que el sistema funcione correctamente. Los resultados mostrados son el promedio de tres operaciones de carga, aunque observamos una baja variación entre los resultados. La Figura 4.6 muestra la información de la tabla como una comparación relativa en el tamaño de almacenamiento con respecto a MongoDB.

Dataset	Maximum datablock size	MongoDB	Cassandra	PostgreSQL	MySQL
Dataset 1	1.000.000	198	197	214	224
	100.000	202	200	217	220
	10.000	235	224	247	262
Dataset 2	1.000.000	1875	1882	1960	2118
	100.000	1893	1884	1970	2109
	10.000	2162	2056	2262	2603
Dataset 3	1.000.000	4233	4243	4413	4337
	100.000	4313	4308	4486	4365
	10.000	5062	4753	5377	6130
Dataset 4	1.000.000	8265	8291	8609	9116
	100.000	8388	8478	8726	9397
	10.000	9893	9160	10270	12406

Tabla4.3. Almacenamientorequerido (en MB) por las implementaciones basadas en MongoDB, Cassandra PostgreSQL y MySQL.

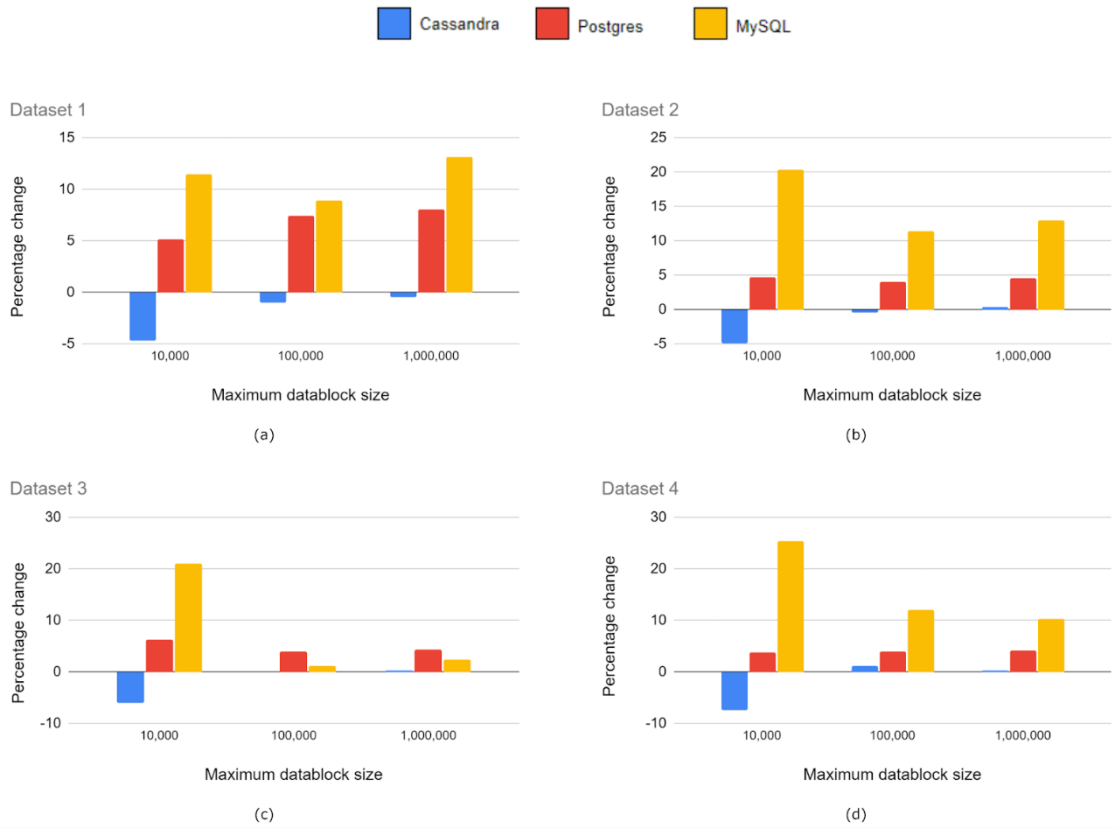


Figura4.6. Comparación relativa del almacenamiento requerido. Resultados para (a) Dataset 1; (b) Dataset 2; (c) Dataset 3; y (d) Dataset 4.

Test de peticiones simples

La Tabla 4.4 muestra una comparación relativa de los tiempos promedio necesarios para leer 1.000.000 de puntos desde un solo cliente en las diferentes implementaciones probadas. Los resultados también se resumen gráficamente en la Figura 4.7.

Dataset	Maximum datablock size	MongoDB	Cassandra	Postgres	MySQL
Dataset 1	1.000.000	0,107	0,11	0,264	0,268
	100.000	0,333	0,303	0,965	0,425
	10.000	2,547	1,824	5,997	1,72
Dataset 2	1.000.000	0,149	0,222	0,308	0,324
	100.000	0,701	0,438	0,992	0,4
	10.000	3,718	3,036	9,48	2,792
Dataset 3	1.000.000	0,17	0,237	0,304	0,469
	100.000	0,429	0,332	1,094	0,551
	10.000	3,559	2,453	8,991	2,223
Dataset 4	1.000.000	0,348	0,188	0,304	0,572
	100.000	0,452	0,329	1,042	0,641
	10.000	3,663	2,701	6,859	2,894

Tabla4.4. Tiempos promedio (en segundos) de una operación de solicitud de 1000puntos en las implementaciones basadas en MongoDB, Cassandra, PostgreSQL y MySQL.

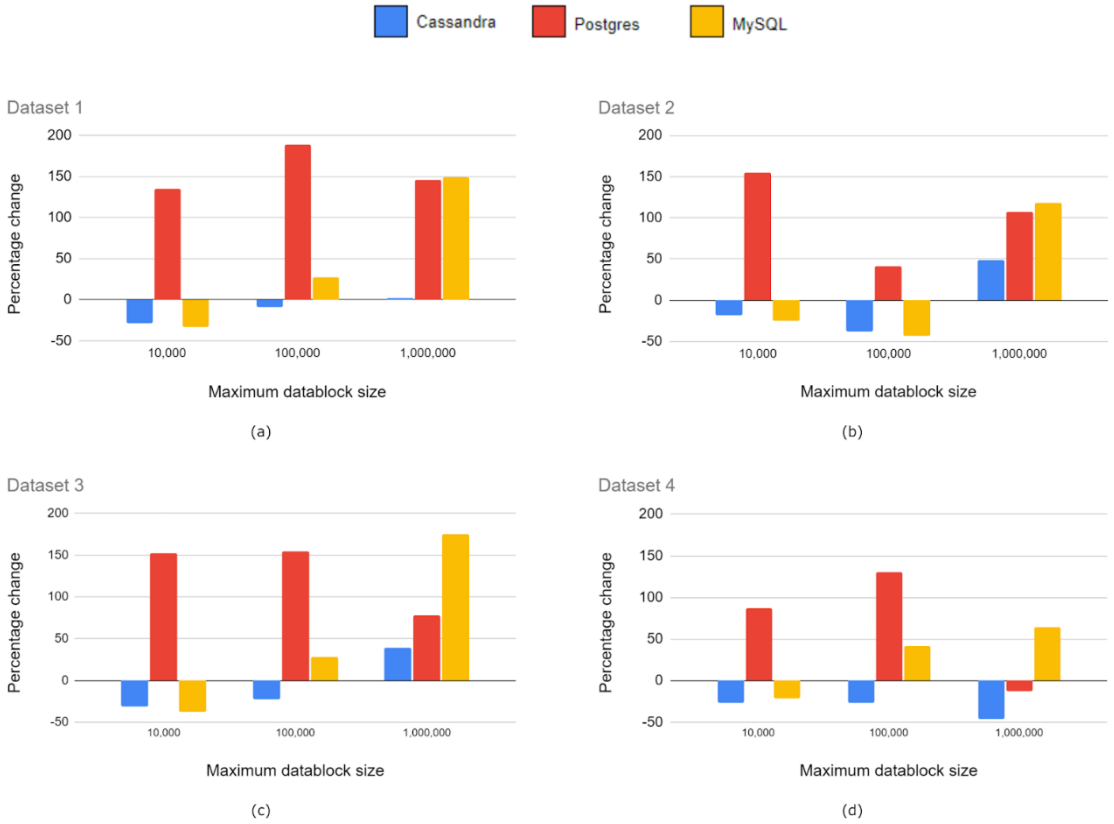


Figura4.7. Comparación relativa de los resultados de la prueba de solicitudes simples con (a) Dataset 1; (b) Dataset2; (c) Dataset 3; y (d) Dataset 4.

Test de peticiones concurrentes

El tercer experimento evalúa el tiempo de respuesta para varias solicitudes simultáneas de 1.000.000 de puntos desde diferentes clientes. Las tablas 4.5 a 4.8 muestran los resultados obtenidos para cada implementación con 10 y 100 clientes concurrentes, respectivamente. Por simplicidad, las solicitudes solo se realizan en el Dataset 1. La Figura 4.8 compara gráficamente el rendimiento de las diferentes implementaciones.

Concurrent users	Maximum datablock size	Average Request Time (s)	Maximum Request Time (s)	Throughput (requests per second)	Total Time (s)
10	1.000.000	1,085	1,512	7	1,532
	100.000	0,174	0,23	55	1,831
	10.000	0,025	0,074	386	2,59
100	1.000.000	13,36	18,572	5	18,61
	100.000	1,715	2,589	56	17,73
	10.000	0,237	0,45	419	23,86

Tabla4.5. Resultados de la prueba de solicitudes concurrentes en Cassandra.

Concurrent users	Maximum datablock size	Average Request Time (s)	Maximum Request Time (s)	Throughput (requests per second)	Total Time (s)
10	1.000.000	1,653	1,68	6	1,691
	100.000	0,252	0,42	39	2,5705
	10.000	0,038	0,076	259	3,8628
100	1.000.000	1,614	1,657	6	16,164
	100.000	2,079	3,823	47	21,4625
	10.000	0,269	0,527	370	27,011

Tabla4.6. Resultados de la prueba de solicitudes concurrentes en MariaDB.

Concurrent users	Maximum datablock size	Average Request Time (s)	Maximum Request Time (s)	Throughput (requests per second)	Total Time (s)
10	1.000.000	2,386	3,002	3	3,024
	100.000	0,224	0,503	42	2,352
	10.000	0,035	0,4	274	3,646
100	1.000.000	23,3	38,57	3	38,62
	100.000	3,763	16,51	26	38,691
	10.000	0,279	0,597	356	28,071

Tabla4.7. Resultados de la prueba de solicitudes concurrentes en PostgreSQL.

Concurrent users	Maximum datablock size	Average Request Time (s)	Maximum Request Time (s)	Throughput (requests per second)	Total Time (s)
10	1.000.000	1,547	1,579	6	1,604
	100.000	0,232	0,296	42	2,36
	10.000	0,044	0,113	223	4,493
100	1.000.000	25,422	34,841	3	34,886
	100.000	9,316	23,556	10	95,749
	10.000	0,416	0,633	239	41,86

Tabla4.8. Resultados de la prueba de solicitudes concurrentes en MySQL.

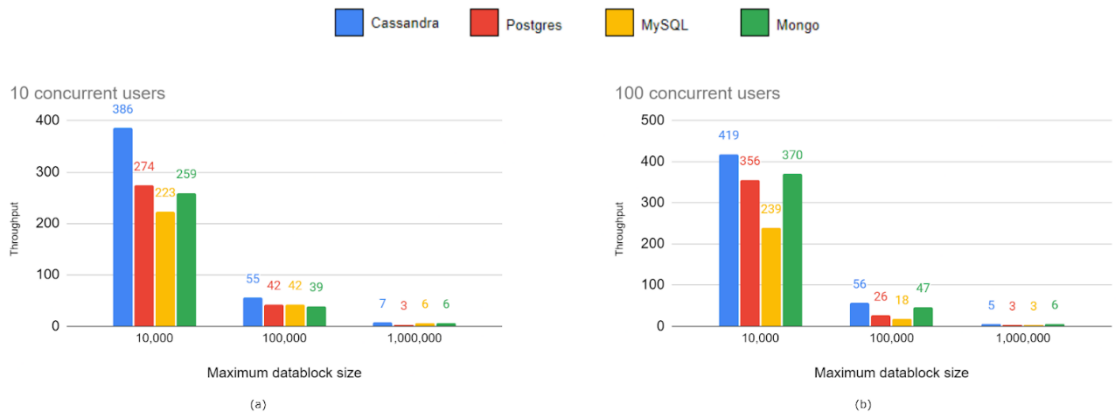


Figura4.8. Comparación del rendimiento de las diferentes implementaciones con (a) 10 usuarios concurrentes y (b) 100 usuarios concurrentes.

4.4. Discusión de resultados

Las bases de datos NoSQL son la principal opción de almacenamiento en las aplicaciones de big data. Nuestros experimentos con *datasets* LiDAR confirman en términos generales la ventaja de utilizar bases de datos NoSQL sobre las relacionales, aunque el rendimiento de MySQL es cercano o incluso superior en determinadas situaciones. En términos generales Cassandra es la ganadora, solo superada por MongoDB cuando la estructura de datos espacial se organiza en grandes bloques de datos (~1.000.000 puntos). Una posible explicación para este hecho puede ser el rendimiento superior de GridFS al manejar archivos LAZ más grandes.

De entre todos los experimentos, las pruebas de carga muestran resultados menos concluyentes. En términos generales, Cassandra y MySQL funcionan mejor con bloques de datos pequeños o medianos, y por tanto, octrees más grandes, mientras que MongoDB supera al resto de bases de datos con bloques de datos grandes (es decir, octrees más pequeños). En cuanto al espacio de almacenamiento requerido, las dos bases de datos relacionales claramente tienen el peor rendimiento. En particular, MySQL requiere hasta un 20% más de espacio que MongoDB cuando se utilizan tamaños de bloques de datos pequeños. Esto puede deberse a una organización interna más compleja de la información o a un uso más amplio de índices, necesarios para poder responder a consultas más complejas en el modelo relacional.

En la prueba de solicitudes simples, Cassandra y MySQL mostraron una latencia más baja con bloques de datos más pequeños y medianos, hasta un 25% mejores que MongoDB, aunque este último seguía siendo en general la base de datos más rápida con un tamaño de *datablock* de 1.000.000 de puntos. PostgreSQL es claramente la peor opción: hasta dos veces más lento que Cassandra en algunos experimentos.

En la tercera serie de experimentos relacionados con solicitudes concurrentes, Cassandra mostró el mayor rendimiento seguida de MongoDB. Sorprendentemente, el rendimiento de MySQL fue mucho menos satisfactorio en situaciones con solicitudes concurrentes que con solicitudes simples, ubicándose en la última posición, incluso detrás de PostgreSQL. Claramente es una debilidad importante de MySQL, que puede desalentar su uso en aplicaciones que necesitan soportar altas cargas de trabajo concurrentes.

En resumen, la recomendación para cualquier proyecto que requiera almacenamiento en una base de datos de grandes conjuntos de datos LiDAR es Cassandra. Su desempeño es excelente en todas las operaciones. MongoDB puede ser una alternativa interesante cuando es necesario almacenar grandes ficheros en la base de datos, gracias al excelente rendimiento de GridFS. Finalmente, si el uso de bases de datos relacionales es un requisito del proyecto, MySQL es la opción preferida, aunque se debe observar cuidadosamente su rendimiento bajo altas cargas de trabajo concurrente.

4.5. Escalado de la capa de almacenamiento: extensión a entornos distribuidos.

Cualquier solución para el almacenamiento de modelos de nube de puntos debe contemplar mecanismos que permitan su escalabilidad a medida que el volumen de datos aumenta. Independientemente de que utilicemos ficheros con formatos optimizados para reducir el espacio de almacenamiento (como LAZ), o de que el sistema de base de datos utilizado integre algoritmos de compresión para reducir el espacio necesario para las entidades del modelo SPSLiDAR, es necesario definir una arquitectura que facilite el incremento de la capacidad de almacenamiento del sistema. Es inevitable que, ante el aumento del volumen de información almacenada, llegue un momento en el que la infraestructura original necesite crecer para dar cabida a toda la información. Es un problema tanto de espacio como de rendimiento: al aumentar el número de *datasets* también lo hará normalmente el número de peticiones desde las aplicaciones cliente, lo que puede afectar a los tiempos de respuesta y generar cuellos de botella en el servidor de base de datos.

A la hora de escalar un sistema, nos encontramos dos enfoques principales: escalado vertical y escalado horizontal. El primero consiste en incrementar los recursos del servidor. Si el espacio de almacenamiento empieza a llegar a su límite podemos ampliar el hardware del equipo añadiendo nuevos dispositivos de almacenamiento para incrementar la capacidad total. Esta solución puede ser útil cuando el sistema está todavía en su fase inicial y los requisitos de almacenamiento total no son demasiado elevados. Sin embargo, conforme aumentan el número de clientes y la volumetría de los datos, el escalado vertical resulta inviable puesto que los equipos tienen limitaciones físicas (máxima memoria RAM instalable, espacio y conexiones para HDDs y SSDs, etc.). Además, el coste económico es más elevado, puesto que adquirir nuevo hardware de estas características resulta más costoso e implica retirar el ya existente, con el consiguiente desperdicio de recursos.

En el caso del escalado horizontal, múltiples servidores trabajan de forma interconectada, aunando sus recursos para poder satisfacer las demandas del sistema. Ante la necesidad de escalar, se introduce un nuevo servidor con los recursos necesarios para alcanzar de forma conjunta las prestaciones necesarias, pudiendo utilizar componentes mucho más modestos. Una ventaja de este esquema de escalado es que puede iniciarse en cualquier momento partiendo de una primera fase de escalado vertical. También existen ciertos problemas que debemos considerar, como una mayor complejidad para mantener todos estos equipos, y la necesidad de definir una estrategia apropiada para distribuir los datos entre los distintos servidores, así como una infraestructura de red más compleja.

Este segundo enfoque será el analizado en esta sección, centrándonos principalmente la distribución de los datos entre nodos. Este es un aspecto crítico para garantizar un correcto escalado horizontal y requiere un análisis de las entidades que componen SPSLiDAR y su volumetría para poder definir estrategias adecuadas. Si uno de los nodos almacena un porcentaje del volumen de información significativamente mayor que el resto, tendrá que atender más peticiones, creando un cuello de botella y degradando el rendimiento general del sistema.

Esta idea de dispersar los datos entre los diferentes nodos del clúster se suele llamar sharding. Las estrategias de sharding a su vez también se pueden diferenciar entre horizontales y verticales. En el primero, se divide una colección de registros de manera que cada elemento quede almacenado de forma íntegra en un mismo nodo. En el segundo, el particionamiento se hace a nivel de atributo o columna, es decir, todos los valores de un atributo concreto presente en múltiples registros se almacenan en el mismo nodo. Este último caso puede ser útil en sistemas centrados en llevar a cabo consultas de tipo analítico, ya que permite computar de forma eficiente valores estadísticos sobre un campo concreto, al estar todos los valores almacenados en la misma ubicación. Sin embargo, en el caso de SPSLiDAR, será el enfoque horizontal el que tenga sentido, ya que en nuestras consultas lo que buscamos es recuperar instancias concretas del modelo de forma completa.

A la hora de distribuir los datos debemos establecer un criterio de asignación de cada registro a un nodo. El procedimiento habitual pasa por definir una clave de sharding basada en los valores que tomen ciertos atributos del elemento que se va a insertar. El algoritmo que se utilice debe ser determinista y replicable, de manera que una vez insertamos un dato, este pueda ser recuperable posteriormente en las búsquedas utilizando la misma clave.

Sharding en MongoDB. Colección datablocks

Aunque los conceptos introducidos previamente son transversales, la forma en la que se implementan puede variar en función del sistema de base de datos utilizado. A continuación se detalla una solución de escalado horizontal basada en MongoDB, al ser el sistema de base de datos utilizado en la implementación de referencia de SPSLiDAR. Una de las principales ventajas de esta tecnología es la infraestructura que incorpora de forma nativa para facilitar la implementación de un modelo distribuido mediante sharding. Cassandra también es un sistema de base de datos que ha sido diseñado desde sus orígenes con el fin de adecuarse a entornos distribuidos (Lakshman & Malik, 2010). Dentro de los sistemas de base de datos relacionales, el soporte nativo para llevar a cabo el sharding es más limitado, y es habitual tener que delegar en la propia capa de aplicación este proceso, incrementando su complejidad: es necesario calcular la clave de sharding, acceder a los distintos nodos, llevar a cabo el redireccionamiento y finalmente la posterior agregación. Al mismo tiempo, el rendimiento global será inferior al de una solución que implemente estos aspectos de manera nativa. En el caso de PostgresSQL, en la actualidad existen extensiones con las que proveer un mecanismo de sharding, y se está trabajando para incorporar una solución de este tipo en el núcleo del sistema.

La arquitectura distribuida de MongoDB se basa en tres componentes principales. Las relaciones que mantienen entre ellos se ilustran en la Figura 4.9:

- ***shard***: define cada uno de los componentes del clúster en los que se almacenan datos.

¶ ***mongos***: es el componente encargado de redirigir las peticiones de consulta o escritura al shard adecuado que contiene los datos necesarios para satisfacerla.

¶ ***config server***: almacenan metadatos con la información necesaria para que las distintas instancias de mongos sepan a qué shard redireccionar la petición.

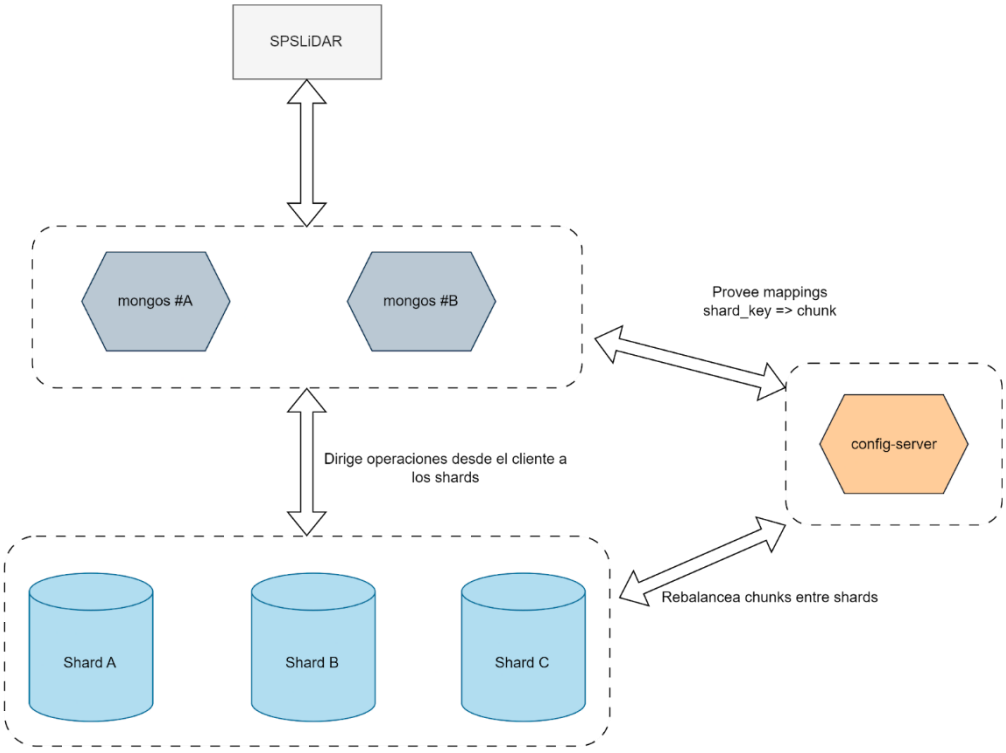


Figura4.9. Componentes de un cluster en MongoDB basado en sharding.

MongoDB introduce un concepto adicional a la hora de llevar a cabo el sharding, denominado chunk. Anteriormente en este capítulo hemos referenciado a otros elementos con un nombre idéntico (*chunk, chunk_order, etc.)* , pero con una semántica distinta y asociados a nuestro modelo. Un chunk dentro del contexto de sharding de MongoDB es un conjunto de datos almacenado de forma conjunta que estarán situados en el mismo shard. Es necesario introducir un matiz fundamental en el caso de MongoDB: la clave de sharding de un documento no especifica a qué shard está asociado, sino al chunk al que pertenece. En la Figura 4.10 se muestra como los documentos de una colección quedan asociados a un chunk y estos a su vez se reparten en distintos shards. Estos chunks, específicos de una colección, se pueden distribuir luego aleatoriamente entre los distintos shards, e incluso ser movidos a posteriori. Este proceso, llamado load balancing, es muy potente, ya que permite mover chunks desde shards con una mayor carga a otros shards con más recursos disponibles, evitando problemas de saturación.

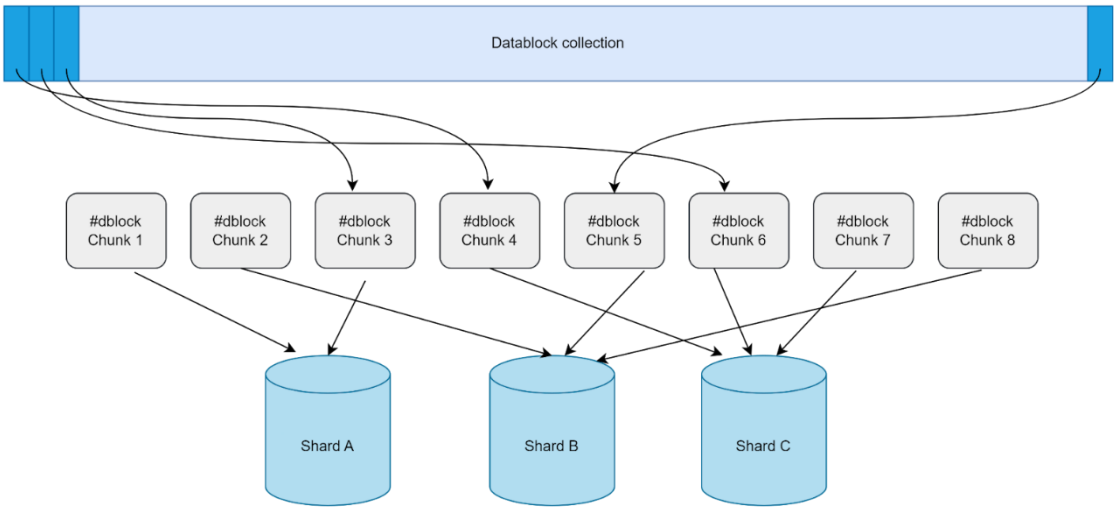


Figura 4.10. Distribución de la colección *Datablock* en chunks.

Por defecto, un chunk establece un tamaño máximo de 128 MBs. Aquí debemos considerar dos principios clave: todos los registros que tengan la misma clave de sharding estarán siempre asociados al mismo chunk, pero también distintos valores de una clave de sharding pueden ser mapeados al mismo chunk. Si el número de documentos se incrementa hasta superar el tamaño máximo de chunk porque múltiples claves han sido mapeadas a este, puede ser dividido, reasignando los documentos en función de sus claves a los nuevos chunk. Sin embargo, si fuese una única clave la que estuviera asociada a todos los documentos, el chunk permanecería inalterable, en lo que se denomina como *jumbo chunk*. Esta situación puede degradar el rendimiento del sistema, ya que al ser un chunk indivisible, puede generar desequilibrios. Por ejemplo, un chunk que termine alojando la mitad de los documentos de una colección puede implicar que un solo shard del clúster sea responsable de manejar la mitad de las operaciones llevadas a cabo.

Partiendo de esta premisa, una clave de sharding basada en un sistema de hashing puede ser una buena opción. Por ejemplo, en el caso de los *datablocks*, podríamos utilizar un identificador único a partir del *_id* de tipo ObjectId que establece MongoDB y aplicar un hashing a este, de manera que los chunks asignados sean prácticamente aleatorios y tiendan a una distribución equitativa. Además, como el valor de este campo será único, en caso de que un chunk supere el límite de tamaño podrá ser particionado en elementos de menor tamaño.

Sin embargo, este enfoque tiene un problema de diseño importante. Cuando realizamos una consulta para recuperar un *datablock*, el cliente tiene conocimiento sobre una serie de atributos: el *workspace* en el que está situado, el *dataset*, la celda de este a la que está asociada la estructura de datos en la que existe el *datablock* y el id del propio *datablock*. Sin embargo, el campo *_id* del documento registrado en MongoDB es un atributo interno desconocido para el cliente. Por tanto, al no poder suministrar este campo no es posible calcular la clave de sharding, determinar el chunk al que pertenece el documento y finalmente localizarlo. En estos casos, MongoDB no tiene otra opción que ejecutar una operación tipo *scatter/gather* para localizar el documento, lanzando la consulta en todos los shards, lo que supone un uso ineficiente de los recursos disponibles.

Por lo tanto, la clave de sharding que seleccionemos, además de tratar de distribuir de una forma equitativa los datos, debe ajustarse a los patrones de consulta habituales que soporta el repositorio. En el caso de la colección *datablock*, tenemos dos formas principales de acceder a los datos: por identificador único, definiendo una composición de los distintos identificadores de las entidades asociadas a este, y por ventana espacial. Este segundo tipo de consulta nos permite incluir un matiz adicional interesante. Con el fin de evitar dentro de lo posible operaciones que afecten a múltiples shards, idealmente los *datablocks* que se recuperen dentro de una misma ventana deberían estar en el mismo chunk.

La primera solución consiste en utilizar una clave de sharding definida a partir del hash del *workspaceName*, el *datasetName* y el *datablockId*, que identifican unívocamente un *datablock*. Sin embargo, el hash obtenido para cada *datablock* será diferente al de los *datablocks* contiguos espacialmente. Otra opción es reducir los atributos utilizados para generar el hash a *workspaceName* únicamente, de forma que todos los *datablocks* de los *datasets* presentes en el *workspace* se mapeen al mismo chunk. El problema ahora sería que podrían generarse chunks con un volumen muy grande que se adaptarían mal a la arquitectura distribuida, puesto que todas las consultas de datos relativas a un *workspace* serían atendidas por el mismo nodo.

Una apuesta intermedia sería usar únicamente el *workspaceName* y *datasetName*. También se incluye el campo *datasetCell*, con el objetivo de definir un mecanismo de control para *datasets* de una extensión considerable que puedan abarcar múltiples celdas. Analizaremos la viabilidad de esta propuesta con un pequeño caso de estudio: supongamos que contamos con *datasets* con densidades de escaneado de hasta 20 puntos/m², y una estructura de datos a nivel *Workspace-Dataset* basada el grid disperso de dos capas que hemos presentado en capítulos anteriores, con un primer nivel que utiliza zonas UTM y un segundo nivel que define celdas de 10 km². El identificador de cada celda dentro de la zona UTM sería el atributo *datasetCell*. Suponiendo que el escaneo ha sido consistente y esta densidad se aplica por toda la región, una nube de puntos de estas dimensiones tendría un total de 200 millones de puntos.

Dentro de cada una de estas celdas, generamos un octree, en el que hemos definido un límite máximo de 10.000 puntos por nodo. Podemos suponer que durante la construcción del octree no todos los nodos llegarán a este límite: los nodos hoja podrían tener un número de puntos menor. Por ejemplo, podríamos establecer un promedio de 8.000 puntos por *datablock*. Para distribuir el total de puntos de la región en el octree, necesitaríamos un total de 25.000 *datablocks*.

En MongoDB, un objeto *datablock* oscila en torno a los 400 bytes. Si multiplicamos el número de *datablocks* por el tamaño de un *datablock*, obtenemos unos 9,54MB aproximadamente. Este valor quedaría por debajo del umbral definido de 128 MB y podríamos asignar todos los *datablocks* de estas celdas sin problemas al mismo chunk. Si bajamos el límite de puntos a 1.000 o si aumentamos el tamaño de celda a 100 km² podríamos llegar a valores cercanos a dicho umbral (ya que el número de *datablocks* se multiplicaría en torno a 10 en ambos casos), aunque estos escenarios podrían considerarse como bastantes extremos. Este mecanismo no es perfecto, ya que las consultas de ventana que traspasen los límites de las celdas deberán acceder a otros chunks. Sin embargo, es una estrategia que permite optimizar las consultas de ventana en la mayoría de situaciones, al mismo tiempo que previene la generación de chunks de tamaño excesivo.

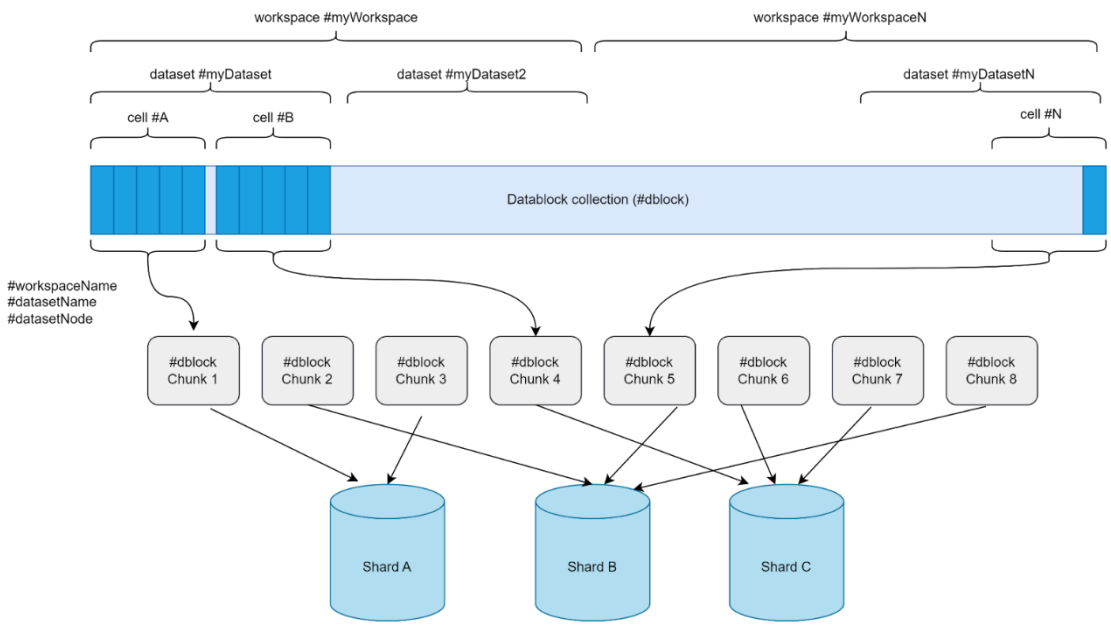


Figura4.11. Distribuciónde *datablocks*con una clave de sharding optimizada para consultas por identificador y consultas por ventana geográfica.

En base a los resultados obtenidos, es posible descartar el atributo *datasetCell* y limitar exclusivamente la clave de sharding a *workspaceName* y *datasetName*. En última instancia dependerá de si se prefiere un mecanismo de control adicional para prevenir chunks demasiado grandes, en casos de tener *datasets* que abarquen extensiones de terreno muy elevadas o si bien, se prefiere mantener todo el conjunto de *datablocks* dentro del mismo chunk, de manera que las consultas de ventana se puedan resolver en un único nodo sin necesidad de agregaciones posteriores.

En cuanto a los casos en los que la estructura *Workspace-Dataset* no establece ningún tipo de organización espacial básica que defina un conjunto de celdas usando criterios geográficos (por ejemplo, cuando usamos un array simple para indexar los *datasets* o estructuras que se basan en criterios temporales), debemos tener en cuenta que el campo *datasetCell* será el mismo para todos los *datablocks*. Si el *dataset* posee un tamaño dentro de los límites que hemos visto anteriormente, esto no supondrá un problema. En caso de que sea mayor, existirá la dificultad de establecer un criterio apropiado con el que limitar el tamaño del chunk. Una posible solución podría ser crear una colección de *datablocks* diferenciada para los *datablocks* en *workspaces* que utilicen estas estructuras, en las que podríamos incluir el identificador de este como clave de sharding. Esto evitará la generación de desequilibrios a cambio de reducir la eficiencia de las consultas de ventana.

Otras colecciones: workspaces, datasets y archivos en gridFS

Hasta el momento nos hemos centrado únicamente en la colección *datablocks*. Sin embargo, el modelo también cuenta con colecciones para otras entidades, como es el caso de *workspaces* y *datasets*. La realidad es que estas colecciones no poseen la volumetría suficiente para que justifique el diseño una estrategia de sharding elaborada. En estos casos, bastaría con asociar tanto

la entidad *workspace* como todos sus *datasets* a un único chunk a partir del hash del identificador del *workspace*.

Un caso aparte sería el referente a los archivos de nubes de puntos asociados a cada *datablock*. Puesto que hemos utilizado gridFS, los datos se almacenan en dos colecciones. En el caso de *fs.chunks*, esta colección almacena los fragmentos en los que se particiona cada fichero de nube de puntos. Para evitar confusiones con el término chunk que hemos visto dentro del sistema de sharding, nos referiremos a los *chunks* de los ficheros de nubes de puntos como *file-chunk* en esta sección. Aquí la estrategia para conseguir un buen rendimiento es simplemente que todos los *file-chunks* asociados a un fichero queden dentro del mismo shard. Esto lo podemos conseguir definiendo como clave de sharding el campo *files_id*. Este es único y compartido por todos los chunks.

En cuanto a *fs.files*, esta colección es mucho más pequeña y solo almacena metadatos de los ficheros. Aunque de acuerdo con la documentación de MongoDB, aplicar sharding sobre ella no es en general necesario, sí que es posible que en algunos escenarios de SPSLiDAR incremente excesivamente su volumen. Si bien son documentos de un tamaño más reducido que los que encontramos en la colección *datablock*, el número de documentos escalará de forma paralela al de *datablocks*. En este caso, podemos aplicar un *sharding* a nivel del campo *_id* sin que suponga demasiado problema, ya que es el único atributo por el cuál accederemos a estos documentos.

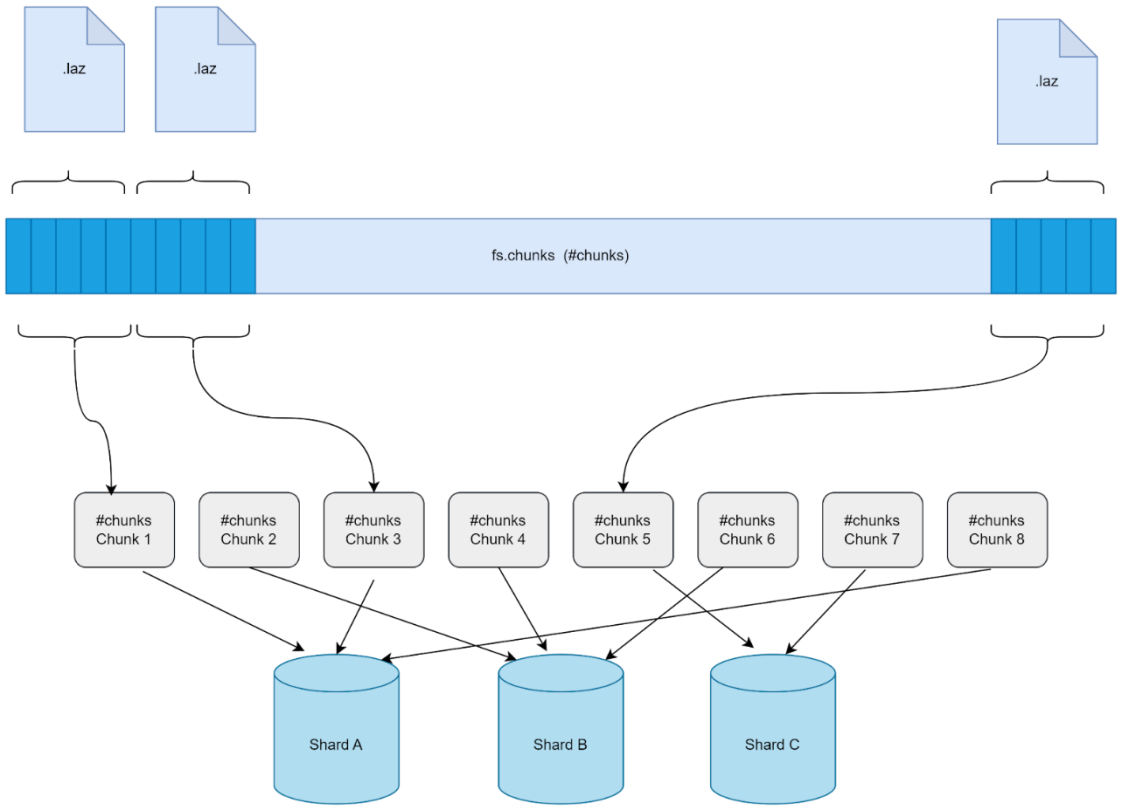


Figura 4.12. Sharding en la colección *fs.chunks* utilizando el campo *files_id*.

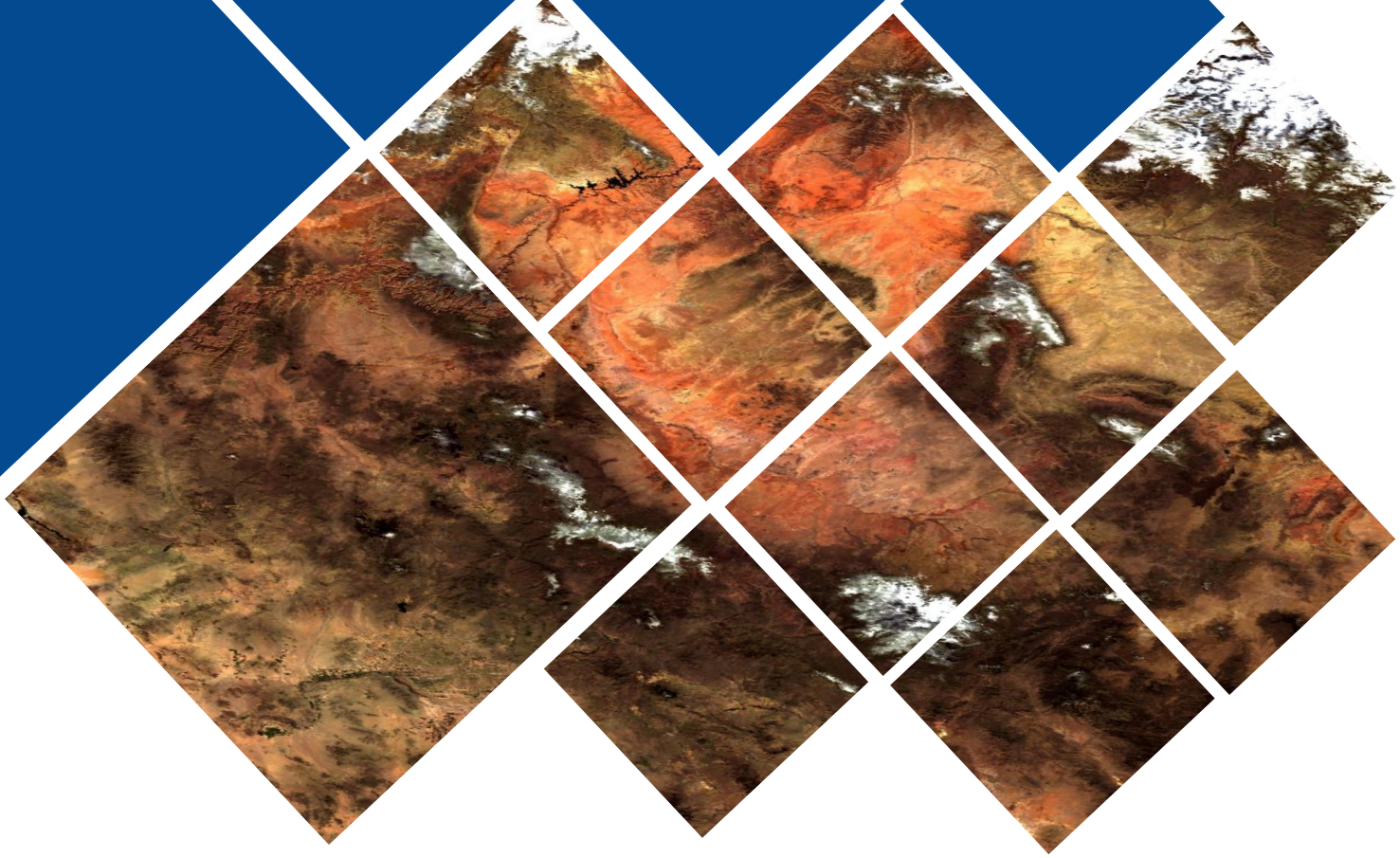
4.6. Conclusiones

Las bases de datos son una opción versátil y sólida a la hora de almacenar datos LiDAR. En este capítulo intentamos arrojar algo de luz sobre la elección más adecuada del sistema de gestión de bases de datos para este tipo de datos. La conclusión es que, aunque las bases de datos NoSQL funcionan mejor que los sistemas relacionales, la brecha, particularmente con MySQL, es estrecha. En general, Cassandra destaca en todas las áreas, y solo se queda por detrás de MongoDB cuando los *datasets* se dividen en bloques grandes.

Muchos enfoques utilizan una base de datos solo para almacenar metadatos de conjuntos de datos o índices espaciales, manteniendo los datos LiDAR en archivos externos. Nuestra apuesta por almacenar datos LiDAR en la base de datos se debe a sus claras ventajas: simplicidad, coherencia garantizada y almacenamiento distribuido cuando utilizamos motores de bases de datos que lo soportan. Sin embargo, una comparación más detallada del rendimiento de las dos opciones, almacenamiento en la base de datos frente a almacenamiento en ficheros externos, proporcionaría información útil para la toma de decisiones a la hora de diseñar e implementar sistemas que tengan que mantener grandes volúmenes de información LiDAR.

En cualquiera de las dos opciones anteriores, el almacenamiento y procesamiento de volúmenes masivos de datos LiDAR requiere normalmente una arquitectura distribuida. En este capítulo hemos discutido las estrategias de sharding necesarias para instalar SPSLiDAR en un clúster de MongoDB, que son fácilmente adaptables a cualquier otra base de datos con una arquitectura distribuida. También sería relevante una nueva experimentación similar a la llevada a cabo con las bases de datos centralizadas. Sin embargo, los sistemas de gestión de bases de datos relacionales analizados no son distribuidos de forma nativa, por lo que serían necesarias soluciones adicionales como MySQL Clúster. No obstante, el uso de esta infraestructura adicional y las decisiones tomadas durante la instalación, configuración y ajuste del clúster y los nodos de la base de datos pueden influir en la experimentación y limitar la validez de los resultados.

Por último, queremos destacar que, si bien hemos utilizado un modelo de almacenamiento basado en ficheros, un modelo de persistencia en base de datos a nivel de punto puede ser una alternativa en implementaciones de SPSLiDAR que requieran una mayor granularidad en el filtrado. Los problemas de volumetría a nivel de registros en una supuesta tabla o colección de puntos pueden ser aliviados mediante el uso de estrategias como el particionamiento del contenido en distintos nodos que contengan subconjuntos del total, o el uso de esquemas adicionales que permitan agrupar puntos similares. Algunas bases de datos como Oracle o PostgreSQL (vía la extensión PostGIS) implementan de forma nativa este segundo concepto (Van Oosterom et al., 2015).



Capítulo 5 - SPSLIDAR EN EL CONTEXTO DE LOS SERVICIOS WEB Y ESTÁNDARES DE LA OGC

5.1. Introducción

El Open Geospatial Consortium (OGC) es una organización internacional sin ánimo de lucro fundada en 1994 que cuenta con más de 500 socios del ámbito empresarial, gubernamental y de investigación, cuyo objetivo es la elaboración de estándares para la comunidad geoespacial que faciliten la interoperabilidad entre el software y los distintos servicios de datos. Por su importancia, cualquier iniciativa para el modelado y acceso a la información espacial como SPSLiDAR debe tener en cuenta los estándares ya propuestos por la OGC y los documentos de trabajo que anticipan las líneas de trabajo futuras. Este capítulo presenta una comparativa de las novedades presentadas por el OGC con las aportaciones de esta tesis, ya que comparten objetivos y filosofía.

5.2. Los servicios Web WMS, WMTS, WFS y WCS

Desde principios de los 2000 la OGC ha participado en múltiples iniciativas para facilitar el intercambio de información geoespacial en la WWW. El estándar Web Map Service (WMS) supone el primer acercamiento de la OGC a los servicios web de este tipo. Define una API para recuperar mapas centrada en torno a dos servicios principales. El servicio *GetCapabilities* devuelve metadatos extraídos de los mapas disponibles en formato XML. La información obtenida puede ser utilizada para construir peticiones al servicio *GetMap*, estableciendo ventanas espaciales y aspectos varios del archivo de salida como su resolución o formato (JPEG, PNG, TIFF, etc.). La respuesta de este servicio es un mapa construido de forma dinámica en base a estos parámetros.

A raíz de WMS, surgen otros formatos con una filosofía similar. Por ejemplo, el estándar Web Map Tile Service (WMTS) se centra en exponer imágenes pregeneradas, en contraposición al enfoque dinámico de WMS. Construye una jerarquía de niveles, dentro de cada cual existe un grid en el que cada celda tiene asociada una imagen ya construida. En este caso el servicio *GetCapabilities* presenta la información necesaria para explorar la estructura de niveles y grids.

El estándar OGC Web Feature Service (WFS) permite acceder a información geográfica vectorial, normalmente en forma de ficheros GML que representan distintas geometrías relevantes del terreno. El estándar OGC Web Coverage Service (WCS) expone servicios que permiten devolver información de coberturas, información geoespacial que incluye imágenes satelitales, escaneos de sensores o DEMs. La información puede estar representada por mallas o nubes de puntos y permite incluir atributos adicionales útiles para tareas analíticas. Dentro de una cobertura puede haber varios subconjuntos de datos que aporten información de distintas categorías.

5.3. El informe OGC Testbed-14 para la gestión de nubes de puntos

Cada uno de los estándares descritos anteriormente se centra en distintos tipos de datos: mapas, modelos vectoriales o ráster. En nuestro caso, el principal punto de interés son las nubes de puntos. La OGC ha definido una serie de recomendaciones y requisitos comunes para los servicios web orientados a este tipo de información (Boehler et al., 2018). En esta sección pretendemos analizar

cómo encaja nuestra propuesta dentro de las bases que presenta el informe, qué aspectos cumple y cuáles no, así como posibles extensiones y mejoras del modelo.

Esquema flexible

La OGC recomienda que un servicio web sea agnóstico al formato de las nubes de puntos, así como a los tipos utilizados para representar los atributos de los puntos. Esto implica que el servicio debe ser capaz tanto de procesar ficheros en distintos estándares para nubes de puntos, como LAS, NGA, etc., como de almacenar bajo distintos tipos de representaciones la información asociada a estos puntos. A nivel conceptual, SPSLiDAR no impone restricciones en el formato que se puede utilizar. El campo *datablockFormat* de la entidad *Dataset* facilita a los clientes la posibilidad de especificar el formato que desean utilizar para almacenar los ficheros correspondientes. Los formatos que finalmente se soportan dependen en última instancia de la implementación. Con respecto al ámbito de los puntos, como no se modelan bajo una entidad propia, utilizan la representación definida en la especificación del tipo de fichero elegido.

Fuentes de datos configurables

SPSLiDAR se ha desarrollado con la idea de soportar una única fuente de datos, basada en la implementación de un repositorio de nubes de puntos propio. En el futuro se podría plantear la integración con otros repositorios de nubes de puntos externos, definiendo cómo se pueden integrar sus representaciones propias y metadatos con SPSLiDAR con el fin de proveer una API para los clientes capaz de recuperar información de múltiples proveedores.

Particionado espacial

SPSLiDAR soporta el uso de distintas estructuras, incluidas aquellas que particionan la información del modelo en base a criterios espaciales. Durante la definición del modelo hemos presentado el uso de grids, quadtrees u octrees para generar la estructura de *datablocks*, permitiendo hacer consultas de ventana sobre estas para permitir filtrar a los clientes en base a criterios espaciales de forma eficiente. La versión actual de nuestra API solo permite realizar consultas por ventanas basadas en dos puntos, aunque se puede extender fácilmente para incluir geometrías y volúmenes más complejos.

Filtrado por atributos

El filtrado por atributos no está soportado de forma nativa en SPSLiDAR. Esto es una consecuencia de nuestro enfoque centrado en ficheros. Por definición, los ficheros generados para cada *datablock*, una vez creados, son opacos para el sistema y no se contempla ningún procesamiento a nivel interno. Una posible extensión del modelo podría soportar este requisito; el número de atributos disponibles en un formato como LAS 1.4 es elevado, pudiendo tener hasta

26 campos distintos. La utilidad de soportar operaciones de filtrado en base a múltiples de estos campos tendrá escasa utilidad en la mayoría de casos de uso; sin embargo, algunos atributos como la clasificación pueden ser interesantes. Se puede incluir un parámetro de consulta adicional en los servicios asociados a la recuperación de ficheros de nubes de puntos, de manera que permita indicar los tipos de clasificación a devolver; de este modo, la nube de puntos recuperada contendrá un subconjunto de los puntos presentes en el archivo original almacenado para un *datablock*, únicamente con aquellos puntos cuyo valor de clasificación coincide con los determinados en la petición del cliente.

Por ejemplo, la siguiente petición basada en el servicio **#8** de la especificación SPSLiDAR, devolvería una versión del fichero asociado al *datablock* indicado compuesto únicamente por puntos asociados a los tipos de clasificación 2 (tierra), 3 (vegetación) y 6 (edificios).

/**workspaces**/{workspace_name}/**datasets**/{dataset_name}/**datablocks**/{datablock_id}/**data**
?**cell**={dataset_cell}&**classification**=[2,3,6]

El uso de otros atributos también puede ser tenido en cuenta en la definición de nuevos algoritmos y estructuras de datos. De este modo, se podría directamente generar una jerarquía de *datablocks* en la que se consideren estos aspectos para construir ficheros que ya de por sí tengan, por ejemplo, puntos únicamente asociados a un tipo de clasificación. De este modo, no sería necesario llevar a cabo un posprocesamiento adicional en los servicios de recuperación, mejorando los tiempos de respuesta.

Representaciones alternativas

A partir de los modelos de nube de puntos podemos generar distintos tipos de productos que pueden ser de interés para los clientes. Se propone que el servicio sea capaz de responder con versiones rasterizadas o voxelizadas de las nubes solicitadas. SPSLiDAR no soporta esta funcionalidad de forma nativa, ya que su filosofía ha sido siempre centrarnos en el almacenamiento, organización y acceso eficientes a la información, sin incluir aspectos relacionados con la representación. No obstante, se podría extender la API para incluir fácilmente nuevos servicios que faciliten esta tarea. Por ejemplo, se podrían definir parámetros en un endpoint para generar representaciones alternativas para los datos asociados a un *datablock*, como ilustra este ejemplo:

/**workspaces**/{workspace_name}/**datasets**/{dataset_name}/**datablocks**/{datablock_id}
/**representations**?**format**=[IMG_PNG, IMG_JPG, IMG_TIFF, VOX_BINVOX]

Niveles de detalle

Una de las funcionalidades requeridas por los clientes, especialmente aquellos que requieren la visualización de nubes de puntos, es la representación de la nube en distintos niveles de detalle, de forma que se puedan cargar nuevos puntos progresivamente en función de la posición y punto de vista del usuario. SPSLiDAR soporta esta funcionalidad mediante las relaciones presentes entre *datablocks*. Cuando organizamos nuestros *datablocks* en base a una estructura jerárquica, los *datablocks* hijos permiten a los clientes profundizar en el modelo y adquirir volúmenes de puntos adicionales en aquellas regiones que son de interés.

Compatibilidad de formatos

El modelo no define una serie de valores específicos que este campo pueda tomar, y es la implementación la que debe interpretarlo y definir si es un formato válido para ser utilizado. Sería interesante incluir también un servicio adicional que permita exponer a una implementación concreta los formatos que soporta.

Compresión

SPSLiDAR delega la compresión de la información en el formato de fichero especificado en el campo *datablockFormat*. Por ejemplo, el formato LAZ (Isenburg, 2013) puede reducir el espacio requerido de una nube de puntos a un 7-20% de su equivalente en LAS.

Organización de metadatos

El soporte de metadatos SPSLiDAR 1.1 se limita al tipo y parametrización de la estructura de datos utilizada para organizar internamente los *workspaces* y *datasets*. Sería posible definir una nueva entidad a nivel de *workspace*, *dataset* o *datablock* para incluir información arbitraria que requieran los usuarios. El cliente podría especificar atributos adicionales mediante esta entidad, teniendo un fin meramente informativo.

Reconstrucción de superficies

Generar representaciones derivadas de la nube como pueden ser mallas de polígonos se puede plantear como un tipo más dentro del visto en el apartado anterior **Representaciones alternativas**, permitiendo seleccionar entre distintos algoritmos y parámetros para llevar a cabo el proceso en función de las capacidades de la implementación. En todo caso, como indicamos anteriormente, son aspectos no contemplados hasta ahora, ya que nuestro propósito ha sido mantener la especificación SPSLiDAR lo más sencilla y genérica posible.

Redundancia descentralizada

Los servicios web deberían permitir la descentralización de los datos para garantizar la redundancia y alta disponibilidad de estos. Para ello, es necesario que los recursos sean identificables y las peticiones idempotentes. En SPSLiDAR cada entidad cuenta con un identificador único, y se ha definido una API REST para el acceso que por definición debe ser idempotente. En un sistema como SPSLiDAR las operaciones de consulta predominan, con mucho, sobre las de escritura, pero aun así las operaciones de creación de *workspaces* o *datasets* deben implementarse de forma que si se ejecutan de manera repetida usando los mismos identificadores el estado final del servidor sea el mismo.

Como vimos en el capítulo anterior, a nivel de implementación de la capa de persistencia, SPSLiDAR es suficientemente flexible y sencillo como para admitir un esquema de sharding en una base de datos distribuida, simplemente repartiendo las entidades entre los distintos nodos en base a los identificadores. Naturalmente esto puede combinarse con una estrategia de replicación a nivel de base de datos y añadiendo uno o varios nodos adicionales con la API SPSLiDAR. Un balanceador de carga a la entrada como el popular nginx permitiría atender las llamadas al sistema y repartirlas entre los distintos nodos API.

Particionamiento temporal

El particionamiento temporal es un aspecto que hemos tenido en cuenta desde la concepción de SPSLiDAR. La entidad *Dataset* posee el atributo *dateOfAcquisition* que permite especificar la fecha de adquisición de este y permite llevar a cabo consultas de ventana por criterios temporales, como es el caso del método **#4**. Como vimos en el **Capítulo 3**, también es posible incorporar un esquema de indexación temporal para acelerar estas consultas.

5.4. Estándares y soluciones para modelos de nube de puntos

El informe anterior cita algunas soluciones ya existentes en el contexto de los modelos de nubes de puntos, algunas de las cuales cuentan con una gran popularidad a pesar de no ser estándares de la OGC. La mayoría de estas soluciones están especialmente orientadas al renderizado y se centran en establecer modelos optimizados para ello.

Por un lado, tenemos propuestas como PotreeConverter (Schütz, 2016; Schütz et al., 2020) y Entwine (Hobu Inc, 2016) . Estas herramientas permiten transformar una nube de puntos adquirida en una jerarquía de ficheros que solo es necesario construir una vez, normalmente un octree. Utilizando clientes especializados, se puede llevar un renderizado eficiente utilizando la estructura de niveles de detalle construida. Podemos llevar estos ficheros a un servidor y alojar un sitio web con un cliente que permita visualizar estas nubes de puntos accesible de forma remota, pero no existe una API que nos permita descargar estos mismos ficheros en nuestro equipo de una forma consistente y estandarizada.

Otro modelo para la organización de datos es el planteado por i3s (Reed & Belayneh, 2017), desarrollado por la empresa Esri y recogido por la OGC dentro de sus estándares. Presenta un enfoque más amplio, ya que está dirigido a modelos 3D heterogéneos, no limitándose únicamente a nubes de puntos. Presenta una API sencilla con la que sí que podemos navegar los metadatos que componen una escena, pudiendo recuperar información de los nodos y descargar ficheros asociados a estos. En contraposición con Potree o Entwine que solo permiten usar LAS/LAZ, i3s codifica los puntos de una escena en formato LEPCC. Algunos atributos se almacenan de forma agregada, como las coordenadas XYZ o los datos de color RGB, pero otros como la intensidad quedan registrados en un fichero propio. El uso de formatos menos habituales y la necesidad de realizar varias llamadas para obtener el contenido completo de la nube asociada a un nodo son dos desventajas de este estándar.

Dentro de todas estas opciones, queremos centrarnos en concreto en 3D Tiles. Las similitudes con SPSLiDAR y el hecho de que sea un estándar reciente de la OGC han motivado el desarrollo de realizar un análisis más detallado, donde resumimos sus aspectos más relevantes y los comparamos con los de nuestra propuesta.

Estándar 3D Tiles

El estándar 3D Tiles es una especificación desarrollada por Cesium GS Inc. y recogida dentro de los estándares promovidos por la OGC. Define un modelo con el que organizar información 3D heterogénea, facilitar el acceso a esta y facilitar su visualización en tiempo real. 3D Tiles cuenta con dos versiones de la especificación. La 1.0 fue publicada en 2019, mientras que la 1.1 lo hizo en 2023. A continuación, comentaremos las diferencias que consideramos más significativas entre ambas versiones, aunque para profundizar en un mayor detalle en todas las características de ambas versiones así como los cambios introducidos y sus justificaciones recomendamos consultar los documentos de especificación de estas dos versiones.

Cesium además trabaja en distintas herramientas orientadas al renderizado de información geoespacial. CesiumJS es una librería con la que representar este tipo de contenido en navegadores web en tiempo real, pudiendo utilizar 3D Tiles para la transmisión y organización de los datos. La información puede ser recuperada de distintas fuentes de datos, aunque con este fin también ofrecen una herramienta propia denominada Cesium ION, que actúa como un repositorio en el que poder almacenar datos bajo el modelo de 3D Tiles. También ofrece distintas librerías y plugins con los que integrar las funcionalidades base de CesiumJS con clientes móviles o con motores gráficos como Unreal Engine. El resultado es un ecosistema en el que podríamos definir una fuente de datos de contenido 3D renderizable al que puedan acceder distintos tipos de clientes.

Modelo

Al igual que en SPSLiDAR, 3D Tiles también introduce un modelo de datos con el cuál se generan distintos metadatos que ayudan a estructurar y organizar el contenido 3D. Introduce tres entidades clave, *Tileset*, *Tile* y *Content*. Un *tileset* es una colección de *tiles*. Los *tiles* se organizan en una estructura jerárquica siguiendo distintos niveles de detalle. Esto permite a los clientes renderizar

de forma progresiva la información, optimizando la cantidad de recursos utilizados en base a la posición y campo de visión del usuario. Cada *tile* puede contar con múltiples contenidos. La entidad *Content* se utiliza para representar un fichero de un *tile* con contenido 3D renderizable. En la Figura 5.1 se muestra un diagrama de clases con estas tres entidades. Hemos especificado solo aquellos atributos que son obligatorios (subrayados en la figura), o bien consideramos que son lo suficientemente relevantes para comentar a posteriori.

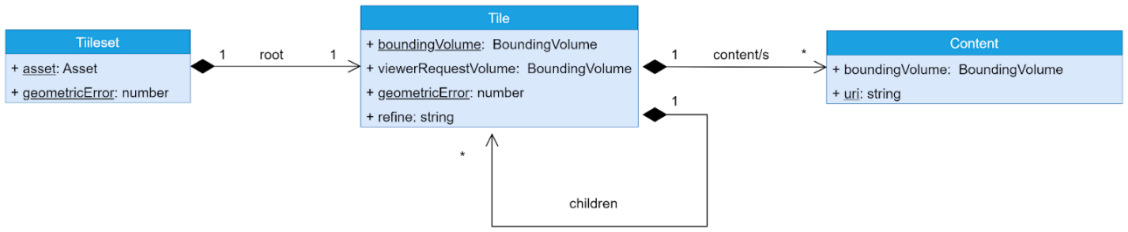


Figura 5.1. Diagrama de clases con las tres entidades clave del modelo definido por 3D Tiles.

Dentro de la entidad *Tileset*, el atributo *asset* se utiliza para almacenar información de tipo general sobre una instancia de *tileset* concreta, como la versión de la especificación que utiliza (1.0, 1.1) o para definir un sistema de versiones propio del *tileset* de cara a posibles cambios o actualizaciones sobre los contenidos de este. Encontramos también el atributo *geometricError*, que también aparece como un campo dentro de la entidad *Tile*. Tiene un papel clave para los clientes que interactúan con este modelo, ya que será utilizado a nivel interno para determinar durante el renderizado del modelo si se deben o no descargar los siguientes niveles de detalle de la jerarquía.

La entidad *Tile* incluye además los atributos *boundingVolume* y *viewerRequestVolume*, ambos basados en la clase *BoundingVolume*. El primero representa la región asociada a este *tile*, mientras que el segundo, define una región de la escena en la que el cliente debe estar situado para poder acceder a los contenidos, lo cual puede ser útil en los casos que un *tile* concreto represente contenido asociado a espacios interiores.

La clase *BoundingVolume* consta de tres implementaciones distintas: *box*, *region* y *sphere*. Tanto *box* como *region* representan un volumen rectangular: *region* toma seis valores, correspondientes al mínimo y máximo X, Y, Z y usando como referencia el sistema de coordenadas EPSG:4979, mientras que *box* toma doce valores, siendo los tres primeros las coordenadas X,Y, Z del centro de la región y los nueve restantes se dividen en tres ternas con las que, para cada uno de los ejes de coordenadas, se define la dirección del eje y la apotema en metros. Por su parte, *sphere* representa volúmenes esféricos, y toma cuatro valores, los tres primeros siendo el centro de la esfera y el cuarto el tamaño del radio en metros. En la Figura 5.2 se muestra una representación de cada una de ellas.

Figura 5.2. Representación de instancias de *BoundingVolume* de tipos *box*, *region* y *sphere* respectivamente.

Un *tile* puede tener a su vez múltiples *tiles* asociados a él a partir de una relación de parentesco. Esta se modela con el atributo *children*, de tipo array. No se imponen restricciones en el tamaño de este array, aunque los volúmenes de los hijos deben estar contenidos dentro del volumen del padre, con el fin de mantener la coherencia espacial.

La otra relación clave que se establece dentro de *Tile* es con la entidad *Content*. En la versión 1.0 de la especificación, esta relación era de tipo 0 a 1. Con la versión 1.1, se modifica esta restricción, de manera que un *tile* pueda tener múltiples contenidos asociados. El último atributo destacable dentro de la entidad *Tile* es *refine*. Este está íntimamente relacionado con las dos relaciones entre entidades que hemos mencionado previamente, ya que establece la estrategia de renderizado a utilizar cuando se recuperan los contenidos de los *tiles* hijos. Es un enumerado y soporta dos valores: *REPLACE* y *ADD*. El primero establece que cuando se recuperan los contenidos de los hijos, el motor de renderizado deberá reemplazar los contenidos que provee el *tile* padre por el de los hijos. Por otro lado, el segundo indica que la escena deberá seguir mostrando el contenido del *tile* padre y los contenidos de los hijos que se carguen complementarán a los suyos, en vez de sustituirlos. Estas dos estrategias permiten de forma nativa establecer una estrategia útil tanto para estructuras de datos redundantes, en las que los hijos poseen la información de los padres, como no-redundantes, en las que el contenido de los padres no está presente en los hijos.

Con respecto a la entidad *Content*, el atributo fundamental es *uri*. En el modelo de 3D Tiles, los contenidos a renderizar no se almacenan en las propias entidades, sino que están presentes en ficheros externos. Este atributo establece la dirección de dicho fichero, pudiendo ser tanto una dirección relativa como absoluta. También se puede incorporar un atributo *boundingVolume*, igual al ya mencionado previamente en el contexto de la entidad *Tile*. Este representa un volumen que modela de manera más ajustada el volumen real del contenido asociado. Mientras que el *boundingVolume* de la entidad *Tile* debe respetar el principio de coherencia espacial, de manera que sea mayor o igual que las regiones representadas por sus hijos, este *boundingVolume* no aplica esta restricción y puede ser ajustado de manera más precisa. En caso de que esté presente, es el volumen que usará el motor de renderizado de cara a sus cálculos internos. Si no, utilizará por defecto el *boundingVolume* establecido en el *tile* asociado al *Content*.

Formatos utilizados

A diferencia de SPSLiDAR, 3D Tiles no es un estándar centrado exclusivamente en nubes de puntos, sino que también soporta otros modelos heterogéneos, con distintos tipos de geometrías. Es un estándar que prioriza el renderizado, por lo que estos modelos suelen incorporar también otra información relevante de cara a este proceso, como texturas o materiales.

Para dar soporte a este tipo de información, se definen varios formatos soportados. Antes de comentarlos, debemos recalcar que el uso de estos formatos es una de las principales diferencias entre las versiones 1.0 y 1.1. En la versión de la especificación 1.0, se introducen tres formatos propios:

- ***Batched 3D Model*** (.b3dm): utilizado para representar modelos heterogéneos, como superficies o edificios.

- ***Instanced 3D Model*** (.i3dm): sirve para representar modelos sencillos que van a ser renderizados múltiples veces con ligeras variaciones, de manera que tengan atributos compartidos así como propiedades específicas de cada instancia. Por ejemplo, modelos 3D de árboles distribuidos a lo largo de una zona.

- ***Point Cloud*** (.pnts): representa un modelo de nube de puntos.

En la versión 1.1 de la especificación estos formatos pasan a ser deprecados y se adopta el estándar glTF 2.0. Ya en la versión 1.0 se utilizaba este formato en el caso del formato .b3dm y, opcionalmente, en el de .i3dm: estos ficheros se componen de una cabecera propia con metadatos varios y un cuerpo que corresponde a una representación binaria de glTF 2.0, en la que están descrita la geometría del modelo.

La implementación de .pnts también se basa en un esquema de cabeceras y cuerpo, aunque este último está constituido por una serie de arrays binarios que representan las distintas propiedades de los puntos (posición, RGB, normal, etc.). Con la versión 1.1, se pasa a generar nubes de puntos en glTF 2.0 utilizando la primitiva *POINTS*.

Finalmente, también existe un caso particular donde el recurso asociado a un *Content* podría no ser de ninguno de estos formatos. Esto ocurre cuando queremos establecer una jerarquía de *tilesets*, de manera que tengamos un *tileset* raíz del que cuelgan otros *tileset* con distintas características pero presentes dentro de la misma jerarquía. Aunque esta relación entre *tilesets* no está definida en el modelo de forma directa, se permite a través de un *Content* que referencia un fichero basado en el esquema *tileset.json* establecido en la documentación.

Estructuras de datos espaciales

Gracias a las relaciones establecidas entre *tiles*, se puede establecer diferentes estructuras de datos con las que organizar la información de un modelo. A nivel general, se puede utilizar cualquier estructura de datos espacial, siempre y cuando se respete la coherencia espacial en la jerarquía. Este aspecto es fundamental, ya que a nivel de implementación los *boundingVolume* son utilizados para llevar a cabo comprobaciones que asumen que, si el volumen asociado a un *tile* no

intersecta con un objeto, tampoco lo hará ninguno de sus hijos. Esto permite descartar de forma eficiente ramas de la estructura.

A partir de aquí, cualquier estructura espacial que cumpla este requisito puede ser válida: quadtrees, k-d trees u octrees son algunas de las opciones más populares como ya hemos visto a la hora de organizar información espacial. También podemos utilizar grids, de manera que tengamos una estructura no-jerárquica. En este caso, debido a la restricción de que la relación *Tileset-Tile* debe ser de 1 a 1, deberemos crear un *tile* vacío que actúa como raíz y mantiene un array con todos los *tiles* correspondientes a las distintas celdas del grid.

Representación alternativa mediante Implicit Tiling

En la versión 1.1 se introduce un modelo alternativo con el que representar estructuras de datos espaciales. Utilizando el enfoque clásico, el procedimiento habitual pasa por modelar un *tile* para cada nodo de la estructura, generando grandes ficheros JSON en los que cada *tile* guarda información de sus hijos. En este enfoque alternativo, denominado *Implicit Tiling*, se utiliza una entidad nueva llamada *Subtree*, que permite agrupar de forma compacta una serie de nodos sin requerir una representación individual de cada uno utilizando un *tile* propio.

Esta representación es solo válida para dos estructuras: quadtrees y octrees. Se basa en la idea de que las divisiones llevadas a cabo sobre estas son regulares y es posible inferir las cajas envolventes de todos los nodos a partir de su posición en la estructura y la caja definida en el nodo raíz. Para ello, es necesario definir un identificador único para cada nodo; esto se consigue mediante una curva de Morton, que recorre los nodos en cada nivel de la estructura y asigna un valor único a cada uno de ellos.

Para dar soporte a esta nueva característica, se introducen una serie de entidades presentadas en la Figura 5.3. En la entidad *Tile*, el atributo *implicitTiling* sirve de punto de entrada a una entidad homónima. Dentro de esta, se establece una jerarquía basada en la entidad *Subtree*. Un *subtree* mantiene una serie de arrays que indican los nodos de la estructura que tienen información (*tileAvailability)*, que tienen contenido renderizable asociado (*contentAvailability*) y que sirven como punto de entrada a otro *subtree* de la jerarquía (*childSubtreeAvailability*).

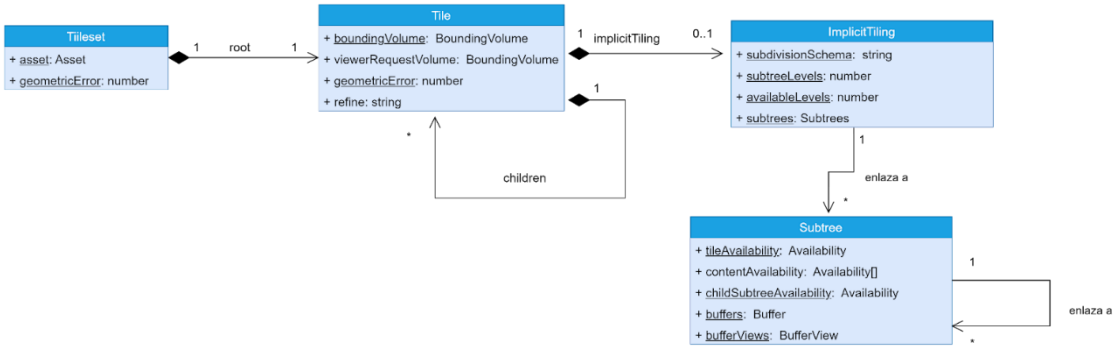


Figura5.3. Diagrama de clases con las entidades claves para la representación basada en *Implicit Tiling*.

Estos arrays representan una serie de bits que actúan como flags, indicando si la información asociada a ese campo está presente o no en el nodo. A partir del índice del array se infiere el identificador del nodo asociado a esa posición. En la Figura 5.4 se ilustra este proceso: el array inferior (IDX array) indica qué celda está asociada a cada posición: 0 hace referencia al nodo raíz, las posiciones 1 a 4 hacen referencia a los nodos 0 a 3 del nivel de profundidad 1 y así sucesivamente. El array superior (Bit array) indica mediante un bit si el nodo asociado tiene contenido o no: en la representación gráfica del quadtree, los nodos coloreados tienen un 1 asociado en este array y aquellos en blanco, un 0.

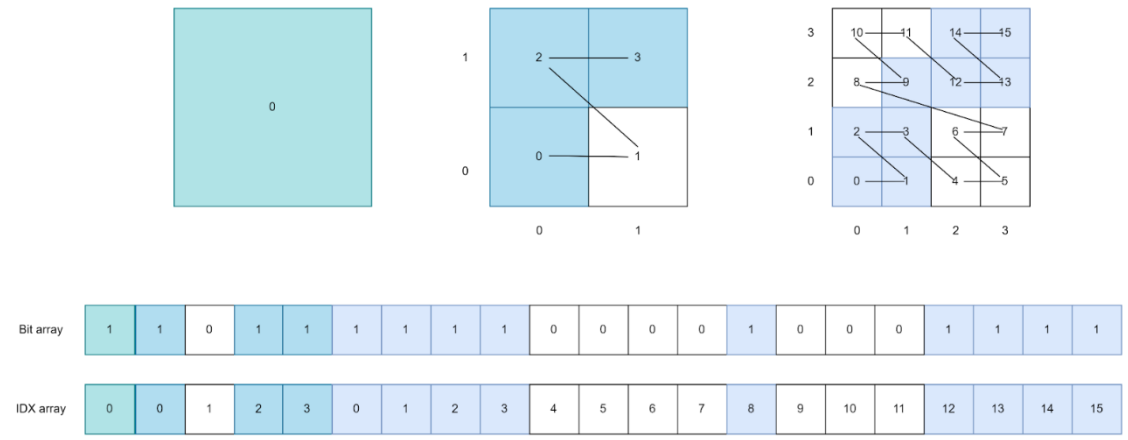


Figura 5.4. Proceso para obtener los arrays de disponibilidad a partir de los identificadores obtenidos a través de la curva de Morton.

Comparativa con SPSLiDAR

Tanto SPSLiDAR como 3D Tiles son dos estándares que permiten organizar información espacial georreferenciada. A partir del uso de un modelo propio, ambos permiten definir una serie de metadatos que facilite la transmisión y el acceso a estos contenidos de forma eficiente.

SPSLiDAR nace como una especificación centrada exclusivamente en modelos de nube de puntos. No impone una restricción en el formato a utilizar, aunque en la mayoría de nuestros ejemplos hemos utilizado LAS y LAZ al ser el principal estándar para este tipo de datos. 3D Tiles, por su parte, se centra en contenido 3D renderizable heterogéneo, siendo las nubes de puntos uno de los varios subtipos que soporta. Para poder dar soporte a estos datos, utiliza un formato genérico para representación de modelos 3D, glTF 2.0.

Cuando trabajamos con modelos de nube de puntos, además de atributos como la posición del punto o el color, encontramos una serie de campos adicionales. Dentro del estándar LAS, en la versión 1.4 de la especificación se da soporte hasta a 10 tipos de formatos distintos de punto, pudiendo almacenar campos con información específica del proceso de adquisición del punto. En un estándar como glTF, las primitivas pueden tener atributos como la posición, el color o referencias a distintas texturas o materiales, pero estos atributos específicos del dominio LiDAR son difícilmente reproducibles y requeriría el uso de extensiones del formato (como *EXT_structural_metadata*) con el que poder atributos adicionales para cada componente de la escena. En el contexto de glTF, esta decisión tiene sentido, ya que realmente no aportan valor

para el renderizado, pero en las aplicaciones que llevan a cabo análisis sobre nubes de puntos sí pueden ser relevantes. Además, sería necesario algún tipo de proceso de conversión de los ficheros originales obtenidos mediante el dispositivo de adquisición, que normalmente suelen ser LAS, al formato objetivo glTF 2.0.

El diseño de 3D Tiles dando prioridad a la operación de renderizado ya ha quedado reflejado en algunos de los atributos que hemos descrito anteriormente, como *geometricError* o *viewerRequestVolume*, que permiten determinar en base a la ubicación y vista de la cámara en la escena si un contenido debe ser seleccionado para ser descargado o no. Aunque se podría desarrollar alguna solución personalizada que ignorase estos aspectos y permitiese descargar información en base a otros criterios, no es inmediato ya que la especificación 3D Tiles se basa en ellos para implementar la navegación por el modelo.

A nivel de estructuras de datos soportadas, en ambas especificaciones se permite el uso de un catálogo amplio de estructuras. Debido a que 3D Tiles establece un único *tile* como raíz, el uso de estructuras no jerárquicas requiere establecer un primer nivel vacío, mientras que en SPSLiDAR podemos definir en el nivel 0 todos los nodos que se consideren. 3D Tiles también establece el requisito de mantener la coherencia espacial en la estructura, aspecto en el que SPSLiDAR es más laxo y que dependerá de la implementación y la estructura utilizada. En este sentido, SPSLiDAR también podría dar soporte a enfoques alternativos para organizar la información de una nube de puntos no basados en criterios espaciales, como generar *datablocks* agrupando puntos en base a su clasificación, por ejemplo. Siguiendo con esta misma idea de dar soporte a estrategias alternativas a la organización espacial, SPSLiDAR soporta de forma nativa la organización de nubes en la dimensión temporal, guardando la fecha de adquisición de este y exponiendo a través de su API una serie de servicios que permitan filtrar a través de ventana temporal.

La última diferencia que queremos subrayar pasa por el uso de la API y la importancia que toman el cliente y el backend en cada una de estas especificaciones. En el caso de 3D Tiles, la implementación de referencia usada como repositorio de recursos 3D, Cesium ION, consta de una API sencilla en forma de CRUD, que no entra en el detalle del modelo de 3D Tiles. Permite

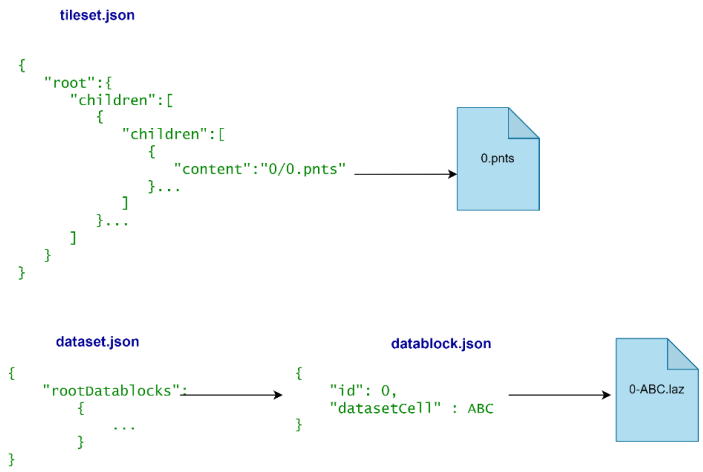


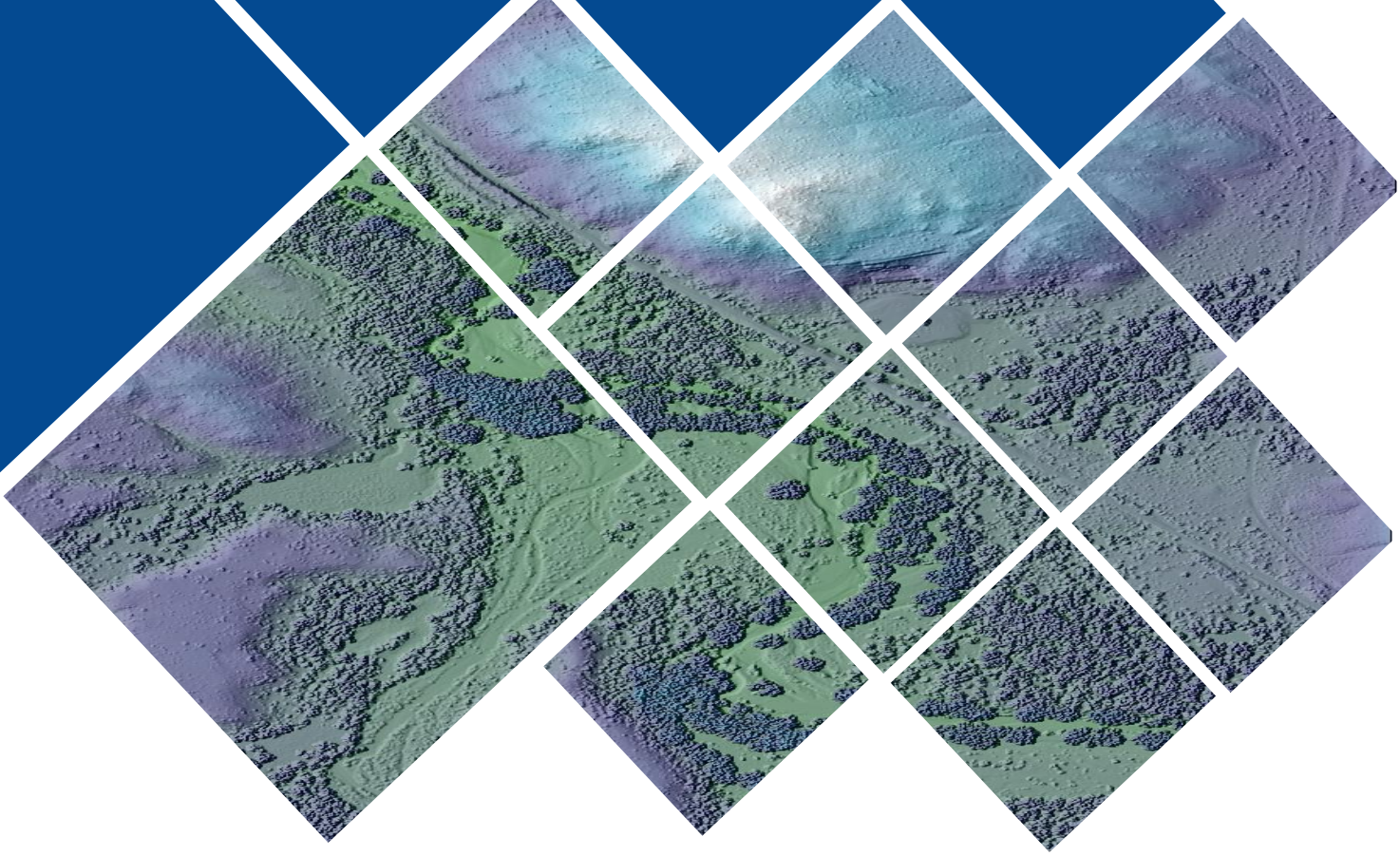
Figura5.5. . Relación entre objetos .json para poder acceder a los distintos elementos del modelo en SPSLi y 3DTiles.

listar los distintos recursos disponibles en un repositorio, los cuales pueden no solo ser archivos basados en 3D Tiles sino que también soporta otros formatos como glTF o GeoJSON. Cuando un cliente recupera un *tileset*, recupera al completo todos los *tiles* que lo conforman, así como las plantillas de *uri* necesarias para recuperar otros ficheros, como los de *content* o los de *subtree*. El acceso se hará de forma directa a estos ficheros. En el caso de SPSLiDAR, los metadatos se descargan de forma progresiva. Cuando un cliente recupera un *dataset*, solo tendrá acceso a la información de este y a las referencias a los identificadores que debe de llamar para empezar a navegar la jerarquía de *datablocks*. El acceso a cada *datablock* se puede realizar de forma individualizada e independiente, lo cual reduce el tamaño de las respuestas parciales pero implica un mayor número de peticiones al servidor (Figura 5.5). En caso de querer recuperar múltiples *datablocks* adheridos a una región específica, en SPSLiDAR el servicio definido para ello solo requiere la parametrización necesaria para definir los límites de esta región, siendo el servidor el encargado de la resolución de la consulta a nivel de implementación. En contraposición, en 3D Tiles el cliente es el encargado de implementar este tipo de funcionalidad, procesando el *tileset* devuelto por el servidor.

Cada enfoque presenta ciertas ventajas. En el caso de 3D Tiles, se reduce el número de peticiones a llevar a cabo al servidor, ya que conoce los metadatos al completo. Esto implica que se cede una mayor responsabilidad al cliente, que necesitará un mayor número de recursos y una implementación más compleja para poder resolver ciertas operaciones. En el caso de SPSLiDAR, se requiere un mayor número de peticiones para poder explorar el modelo, ya que habrá que realizar una petición al servidor por cada *datablock*. A cambio, el servidor expone una API con un mayor número de funcionalidades, lo cual permite al cliente delegar ciertas operaciones en él. En la Tabla 5.1 se realiza una comparativa con algunas de las diferencias fundamentales detectadas entre ambos estándares.

	SPSLiDAR	3DTiles
Casos de uso	Genérico	Especializado en renderizado. Uso de atributos específicos en el modelo.
Tipo de datos	Nubes de puntos	Modelos 3D heterogéneos (múltiples tipos de geometrías)
Formatos	Agnóstico, énfasis en estándares propios de nubes de puntos como LAS/LAZ	Formatos propios (1.0), glTF 2.0 (1.1)
Estructuras de datos	Agnóstico, principalmente estructuras de datos espaciales.	Estructuras de datos espaciales con coherencia espacial. Principalmente jerárquicas, también no jerárquicas utilizando un primer nivel vacío.
Interacción con el modelo	Instancias representadas en objetos .json independientes. Acceso a otras instancias de forma progresiva a través de la API expuesta por la fuente de datos.	Jerarquías *tileset-tile/subtree* embebidas en un único objeto .json. Acceso a fuente de datos para recuperar los contenidos 3D u otras jerarquías de objetos

Tabla 51. Tabla resumen en la que comparamos algunos de los aspectos fundamentales de SPSLiDAR y 3D Tiles



Capítulo 6 - CONCLUSIONES Y TRABAJOS FUTUROS

6.1. Introducción

En este capítulo se presentan las conclusiones del trabajo realizado, así como una exposición de los temas pendientes y derivados más relevantes. Como parte de una línea de trabajo mayor, es de esperar que en el futuro se desarrollen otras investigaciones que utilicen los avances aquí presentados.

6.2. Conclusiones

Los modelos de nube de puntos han adquirido un papel fundamental en multitud de aplicaciones, lo que ha llevado a un aumento exponencial en el número de modelos existentes y en su tamaño, ayudado por la mejora de la precisión y velocidad de los dispositivos de captura. Esto plantea retos en la gestión de grandes colecciones de modelos de nube de puntos que no pueden resolverse de manera manual sin una estrategia clara, y que, a medio plazo, conforme aumenta el volumen de información, pueden dificultar su explotación efectiva. Estos retos están relacionados con el almacenamiento, la catalogación usando distintos metadatos, la organización para facilitar las consultas por diferentes criterios, tanto a nivel de colecciones de modelos como internamente dentro de cada nube de puntos, la distribución a cualquier tipo de cliente, la transmisión eficiente por la red y la visualización de cualquier modelo de forma total o parcial.

El objetivo de SPSLiDAR es proveer una solución integral para estos problemas, definiendo un modelo sencillo a la par que flexible. A diferencia de otros estándares similares propuestos por la OGC como 3D Tiles, SPSLiDAR presenta una solución generalista, no solo orientada a visualización sino también a otros tipos de flujo de trabajo. La potencia de SPSLiDAR radica en su capacidad para representar bajo una misma especificación distintas estrategias para organizar la información de estos modelos, adaptándose a cada caso. Atendiendo a las características de los datos y las necesidades de las aplicaciones, se pueden proveer implementaciones del modelo que organicen la información en múltiples formas (Figura 6.1), como se describe en detalle en el **Capítulo 3**. Este capítulo muestra como estructuras de datos comunes como grids, quadtrees u octrees pueden ser integradas fácilmente en SPSLiDAR a distintos niveles, indicando los casos de uso más adecuados para cada una de ellas.

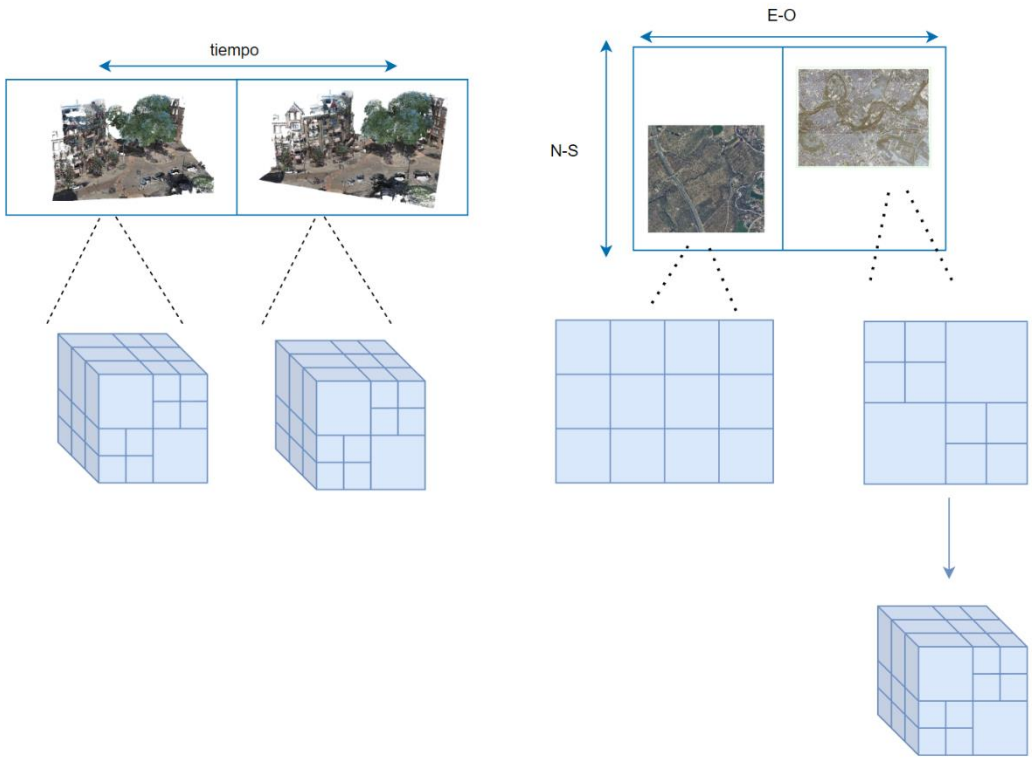


Figura 6.1. SPSLiDAR permite modelar nubes de puntos utilizando múltiples criterios y estructuras de datos.

La principal aplicación del modelo SPSLiDAR es la implementación de repositorios remotos para grandes colecciones de nubes de puntos. En el **Capítulo 2** se ha definido una API con este cometido, adhiriéndose a muchas de las recomendaciones realizadas por el OGC para el diseño e implementación de servicios de provisión de nubes de puntos. El diseño planteado permite navegar por las diferentes entidades y llevar a cabo consultas por ventana espacial y temporal. Junto a las operaciones de consulta, esta especificación también ofrece los servicios necesarios para incluir nuevos modelos de nubes de puntos en el sistema o transmitir de forma total o parcial un modelo a la aplicación cliente. Comparado con las propuestas tradicionales de la OGC (WMS, WMTS, WFS, WCS) o el moderno estándar 3D Tiles, en el **Capítulo 5** mostramos las ventajas de SPSLiDAR frente a estos estándares, concluyendo que la API de SPSLiDAR resulta más sencilla y flexible, además de ser más adecuada para el acceso a modelos de nube de puntos.

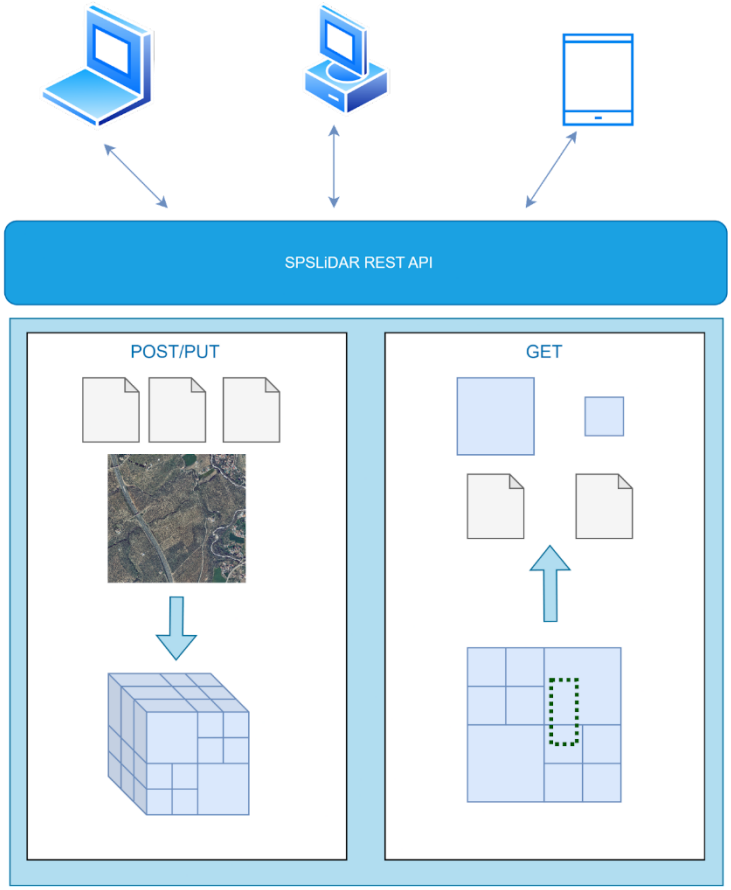


Figura6.2. La implementación de SPSLiDAR en repositorios permite procesar y exponer información de nubes de puntos a través de una API REST.

La capa de persistencia representa uno de los elementos críticos en un repositorio, particularmente cuando está orientado al manejo de datos a gran escala como es el caso de SPSLiDAR. Esa problemática se ve acentuada en el caso de almacenar cada punto de una nube de forma individualizada: el número de registros a consultar en una consulta es extremadamente elevado. Es por ello por lo que SPSLiDAR utiliza un enfoque basado en bloques de puntos codificados en forma de ficheros para el almacenamiento.

Frente a algunas alternativas que apuestan por el almacenamiento de estos ficheros en sistemas de ficheros distribuidos, SPSLiDAR apuesta por el uso de sistemas de base de datos capaces de alojar estos ficheros. Son tecnologías versátiles que permiten unificar en un solo lugar el almacenamiento tanto de las entidades que conforman SPSLiDAR como los bloques de puntos. Además, la mayoría de bases de datos modernas soportan de forma nativa o mediante extensiones el escalado desde un servidor individual a un entorno distribuido.

Dentro del amplio espectro de tecnologías de este tipo disponibles, se ha llevado a cabo la integración de SPSLiDAR con las bases de datos MySQL y PostgreSQL en el ámbito relacional, y MongoDB y Cassandra en el no relacional. Para cada una de ellas se ha diseñado un esquema de base de datos y se ha llevado a cabo una serie de experimentaciones, evaluando el rendimiento en operaciones de inserción y consulta en el **Capítulo 4**. Cassandra presenta en general los mejores resultados. MongoDB también demuestra un buen rendimiento, lo cual nos

hace considerar a las alternativas no relacionales como la solución preferida para el almacenamiento de ficheros de nubes de puntos.

6.3. Trabajos futuros

A continuación, se describen algunas de las líneas de trabajo a seguir de cara a extender y complementar el trabajo realizado. Distinguimos dos líneas principales de trabajo: por un lado, la inclusión de nuevos servicios y funcionalidades que extiendan los casos de uso en los que podemos utilizar el modelo SPSLiDAR y las implementaciones desarrolladas, y por el otro, la definición y adaptación de SPSLiDAR a nuevas arquitecturas y entornos con el fin de conseguir una mayor escalabilidad, eficiencia y optimización de los recursos utilizados.

Integración de nuevos servicios y estructuras de datos

SPSLiDAR es una especificación ideada para ser extensible. Ya sea a través de cambios menores necesarios para adaptarlo a casos de uso concretos, como el uso de una estructura de datos particular, y su parametrización, o bien definiendo una nueva versión con modificaciones más significativas como en el caso de SPSLiDAR 1.1.

Bajo esta premisa, las entidades del modelo pueden ser enriquecidas con nuevos metadatos. En esta tesis se ha hecho especial énfasis en la componente geoespacial de las nubes de puntos, analizando en el **Capítulo 3** la aplicación de múltiples de estructuras de esta índole al contexto de SPSLiDAR. Sin embargo, existen múltiples datos adicionales asociados a las nubes de puntos que pueden ser considerados. La entidad *Dataset* puede incluir información adicional asociada al proceso de adquisición (tecnología utilizada, agentes involucrados en el proceso), de manera que permita capas de filtrado adicional a las ya establecidas. La entidad *Datablock* permite un gran rango de información adicional, definida por las múltiples características que tienen los puntos, más allá de su posición en el espacio: clasificación, tipo de retorno, color, etc. Un *datablock* puede incluir atributos adicionales que aporten información de estos datos, indicando, por ejemplo, los tipos de clasificación presentes en la nube asociada o la cantidad de puntos para cada tipo de retorno del escáner.

En esta misma línea, se ha visto que los criterios utilizados para particionar una nube de puntos en *datablocks* son espaciales. En casos de usos orientados a visualización, este enfoque es el estándar gracias a la posibilidad de resolver consultas de forma eficiente de acuerdo con el viewport del usuario que interactúa con la nube. Sin embargo, otras estrategias pueden ser interesantes para otros casos de uso, ya sea de forma alternativa o complementaria, utilizando algoritmos que particionen la nube en base a otros atributos, como la clasificación. De este modo se pueden generar *datablocks* que contengan de forma específica los puntos asociados a los distintos tipos de clasificación: terreno, vegetación, edificios, etc. La flexibilidad del modelo permite la posibilidad de utilizar distintas estructuras para organizar las categorías de puntos dentro del mismo *dataset*. Por ejemplo, podría establecerse un quadtree para organizar los puntos asociados a fuentes de agua, que por lo general tendrán una naturaleza más plana, frente a otros tipos con un carácter más volumétrico, como los asociados a edificios.

En el **Capítulo 5** se han descrito múltiples aspectos del informe OGC Testbed-14 (Boehler et al., 2018). En este se identifican varias funcionalidades deseables en cualquier servicio web orientado a la provisión de modelos de nube de puntos. En este capítulo se discuten estas funcionalidades y se proponen extensiones para dar soporte a aquellas que actualmente no están contempladas en SPSLiDAR. El incorporar estas mejoras y extensiones sin duda mejoraría el modelo y la implementación de referencia propuestos, ampliando su campo de aplicación.

Edición y trabajo colaborativo sobre modelos de nube de puntos

Dentro del desarrollo de nuevas funcionalidades se puede incluir la implementación de una nueva línea de trabajo, que permita no solo incluir nuevos modelos y consultarlos, sino también la edición de los datos insertados. Esto permite establecer flujos de trabajo con los que corregir o eliminar puntos con información incorrecta o redundante, clasificar la información o incluir nuevos atributos.

Esta línea de trabajo presenta claros desafíos en lo que respecta a la gestión de la concurrencia, persistencia de los cambios y el histórico de estos, así como la notificación en tiempo real y actualización a distintos clientes que trabajen sobre el mismo modelo. La edición concurrente puede implicar conflictos ante la realización de cambios en el mismo modelo por parte de dos clientes distintos. Del mismo modo, la actualización de elementos en el modelo deberá ser transmitida al resto de usuarios, ya sea mediante un sistema de notificaciones o utilizando protocolos de comunicación bidireccional para reflejar estos cambios en tiempo real. La realización de cambios en el modelo abre la puerta a la inclusión de un sistema de control de versiones que garantice la trazabilidad de estos cambios y permita revertir o rehacer actualizaciones (Ogayar-Anguita et al., 2023b).

Almacenamiento distribuido, alternativas en la capa de persistencia y nuevas experimentaciones

Uno de los grandes retos de un sistema destinado a procesar modelos de nube de puntos de forma masiva es poder satisfacer de forma concurrente un número elevado de peticiones. En esta tesis se han abordado desde un punto de vista teórico varios de los aspectos a tener en cuenta: la definición de un modelo lógico que permita convivir en un repositorio múltiples modelos de nube de puntos, el uso de distintas estructuras de datos que permitan acceder de forma eficiente a esta información, la arquitectura de un sistema backend para almacenar esta información y las APIs para poder interactuar con él, así como distintas alternativas a nivel de sistema de base de datos con las que persistir esta información.

En el **Capítulo 4** se presenta un análisis más detallado de los aspectos a tener en cuenta para conseguir escalar la base de datos de nuestro sistema, en concreto utilizando MongoDB como referencia y teniendo en cuenta la naturaleza de los datos a persistir. Sería interesante realizar una experimentación adicional sobre un sistema de almacenamiento puramente distribuido, ingestando volúmenes de información mayores y utilizando distintas estrategias de sharding para validar las ideas propuestas, analizando cómo se adaptan las distintas bases de datos analizadas a este escenario.

Aunque los esfuerzos realizados en esta tesis se han centrado en el uso de sistemas de base de datos para el almacenamiento de la información, también es interesante plantear experimentaciones adicionales que involucren sistemas de ficheros distribuidos. Tecnologías como HDFS son populares para el almacenamiento de nube de puntos y su integración con SPSLiDAR, así como la comparativa con las tecnologías ya analizadas, tanto en un entorno de un único servidor como en uno distribuido, pueden ser interesantes. Del mismo modo, también será útil evaluar el rendimiento de nuevas estructuras de datos y cómo varía el rendimiento del sistema en función del sistema de almacenamiento seleccionado y el tipo de consultas realizadas.

Escalado en la capa de aplicación y nuevas arquitecturas

Si bien la capa de persistencia es uno de los aspectos más críticos del sistema, la capa de procesamiento también debe ser tenida en cuenta. Aunque el stack tecnológico elegido permite soportar un alto grado de concurrencia con recursos reducidos, es interesante definir y plantear una serie de alternativas que permitan adecuar el sistema a un entorno distribuido y escalable. De base, esto puede ser llevado a cabo alojando la aplicación desarrollada en contenedores y desplegándola en un entorno en la nube o en un clúster propio. Utilizando tecnologías de orquestación como Kubernetes, es posible incrementar el número de instancias de la aplicación disponibles en función de la demanda a satisfacer. En este tipo de casos, el uso de un balanceador de carga para repartir el tráfico es absolutamente necesario.

En una arquitectura de estas características, es interesante reevaluar la estructura de nuestra implementación y diferenciar entre servicios de consulta y servicios de procesamiento. Los primeros son servicios ligeros, en los que el mayor coste computacional se delega a la base de datos. Sin embargo, los servicios de procesamiento, en los que incluimos principalmente la operación de importación de un *dataset*, son en comparación mucho más exigentes a nivel de recursos debido al elevado número de transformaciones que debe de llevar a cabo sobre los ficheros de nubes de puntos. Incluso utilizando un modelo programático reactivo, los recursos utilizados son lo suficientemente considerables como para poder repercutir en el rendimiento del sistema, especialmente cuando se lleva a cabo más de una tarea de este tipo.

Hay que tener en cuenta que estas operaciones de importación de *datasets* no se realizan frecuentemente, ya que se ejecutan una única vez durante su creación. Aun así, es una buena idea dividir nuestro sistema en dos microservicios, uno para los servicios de consulta y otro para los nodos de procesamiento de la nube. Esto permite a ambas escalar de forma aislada: debido al volumen de peticiones anticipado para cada uno, podemos suponer que para los servicios de consulta serán necesarios más nodos. Al mismo tiempo, podemos configurar máquinas específicas destinadas exclusivamente para nuestros nodos de procesamiento, con componentes más potentes para poder llevar a cabo estas tareas de manera más rápida y un mayor espacio en disco local para poder almacenar los ficheros parciales que requiere este proceso. En este caso, el sistema deberá contar con un componente enrutador (Ingress) para redirigir el tráfico a un servicio u otro en función de la URI de la petición. Un ejemplo de esta arquitectura se puede ver en la Figura 6.3.

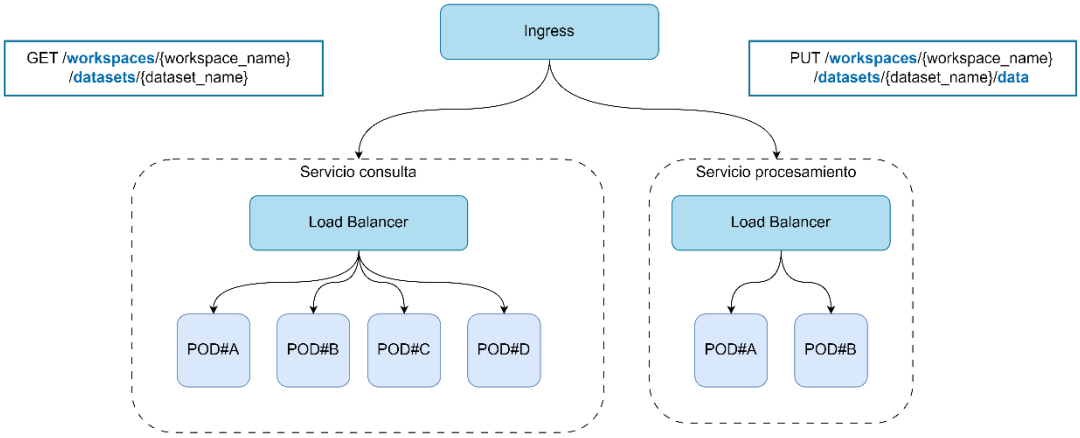


Figura 6.3. División de los servicios de consulta y procesamiento.

Estos nodos de procesamiento se encargan por lo tanto de cualquier tarea que requiera trabajar directamente sobre un fichero de nube de puntos. Algunas de las nuevas funcionalidades propuestas en el **Capítulo 5**, como la generación de productos derivados de nubes como BIMs o rásteres, o la aplicación de un filtrado por atributos sobre el fichero de nube de puntos pueden llevarse a cabo sobre ellos.

Otra posible solución que podemos explorar en un futuro es el uso de mecanismos de comunicación asíncronos para llevar a cabo la construcción de las estructuras de los *datablocks*. En las distintas soluciones abordadas hasta ahora, una instancia de la aplicación se hace responsable de forma completa de construir el modelo. El procesamiento de nubes de puntos masivas se puede paralelizar dentro de los límites del nodo en el que se encuentre la máquina, aprovechando sus recursos lo máximo posible. Escalando nuestro sistema con más nodos y servidores, podemos plantear una solución en la que la carga de la construcción de estas estructuras se reparta entre ellos, agilizando el proceso. Para ello, será necesario un nodo capaz de orquestar el proceso de generación de estas estructuras, manteniendo el progreso de su construcción, y delegando en nodos de procesamiento las operaciones concretas a llevar a cabo sobre los ficheros. Por ejemplo, en el caso de un octree, una de estas tareas será particionar el fichero en ocho regiones, y procesar la respuesta de estos para definir los siguientes pasos a acometer.

Estos mecanismos dependerán de sistemas de comunicación asíncronos. Por un lado, el orquestador no puede quedar bloqueado a la espera de que un nodo termine su tarea, ya que puede tener latencias elevadas (orden de minutos e incluso horas dependiendo del caso). Por otro lado, los nodos que se encargan de llevar a cabo estas operaciones tendrán unos recursos limitados y no podrán llevar a cabo todos los procesamientos demandados de forma simultánea. Las tecnologías orientadas a la transmisión de eventos, como Kafka, pueden ser una alternativa interesante a analizar.

Uno de los aspectos clave a analizar en esta arquitectura es cómo afecta el tiempo de transmisión de ficheros. Para que estos nodos puedan realizar transformaciones sobre ficheros de nubes de puntos, estos deben de estar primero presentes en su memoria secundaria. Sin embargo, en un entorno distribuido no podemos asumir que sea tal el caso, ya que el archivo que necesite como entrada puede haber sido generado en un nodo diferente. Será necesario responder preguntas sobre cómo almacenar estos ficheros temporales, analizar si el incremento en tiempo

adicional debido a la transmisión de estos ficheros al nodo destino supone una desventaja crítica como para que la arquitectura sea válida, así como plantear posibles estrategias que optimicen el paralelismo y minimicen dentro de lo posible la necesidad de transmitir ficheros (por ejemplo, haciendo que el mismo nodo encadene ciertas operaciones que requieran ficheros que el propio nodo ha generado previamente y tiene ya disponibles en disco). Para ello, habrá que llevar a cabo una serie de experimentaciones sobre las distintas arquitecturas y variantes que se planteen.

En la Figura 6.4 mostramos un ejemplo posible de lo que podría ser una de estas arquitecturas. Un nodo de tipo orquestador publica una serie de eventos que definen una operación a llevar a cabo sobre un fichero. Un grupo de consumidores, compuesto por varios nodos, procesa estos eventos y genera una respuesta, que es consumida a su vez por el orquestador para actualizar el estado de la jerarquía de *datablocks*.

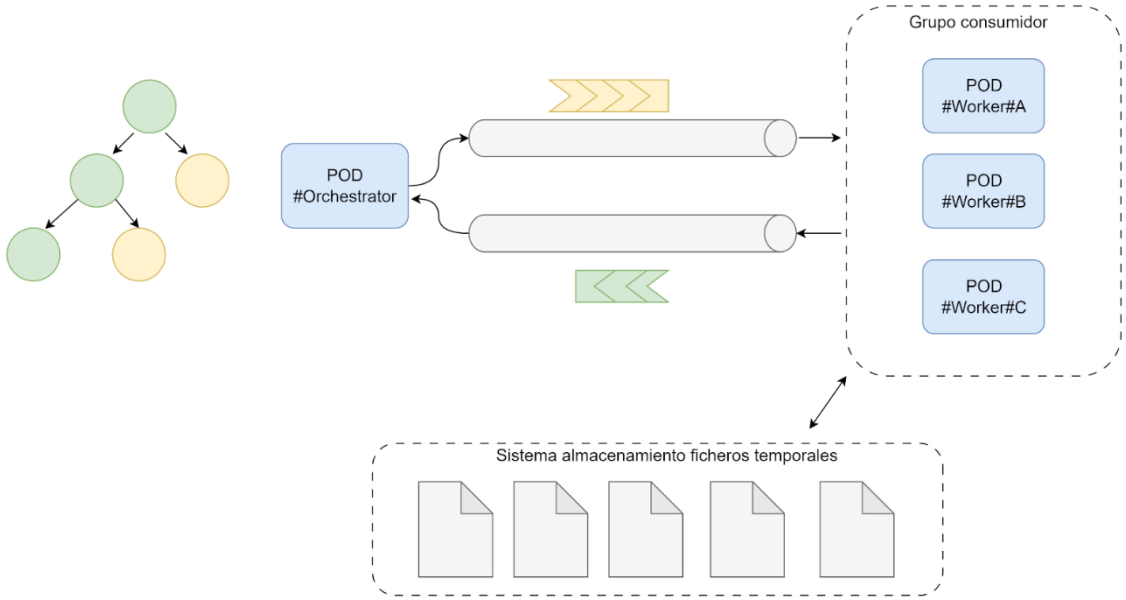


Figura6.4. Modelo de publicación de eventos asíncrona para la realización de tareas de procesado.

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

BIBLIOGRAFÍA

3D Systems, Inc. (1988). STL (Stereolithography) File Format Specification. https://www.3dsystems.com

Agarwal, S., & Rajan, K. S. (2017). Analyzing the performance of NoSQL vs. SQL databases for Spatial and Aggregate queries. https://api.semanticscholar.org/CorpusID:54529614

Akenine-Möller, T., Haines, E., & Hoffman, N. (2008). Real-Time Rendering 3rd Edition. A. K. Peters, Ltd.

ASPRS. (2019). LAS Specification 1.4—R14.

Baert, J., Lagae, A., & Dutré, P. (2013). Out-of-core construction of sparse voxel octrees. Proceedings of the 5th High-Performance Graphics Conference, 27-32. https://doi.org/10.1145/2492045.2492048

Barrile, V., Candela, G., & Fotia, A. (2019). POINT CLOUD SEGMENTATION USING IMAGE PROCESSING TECHNIQUES FOR STRUCTURAL ANALYSIS. The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, XLII-2/W11, 187-193. https://doi.org/10.5194/isprs-archives-XLII-2-W11-187-2019

Béjar-Martos, J. A., Rueda-Ruiz, A. J., Ogayar-Anguita, C. J., Segura-Sánchez, R. J., & López-Ruiz, A. (2022). Strategies for the Storage of Large LiDAR Datasets—A Performance Comparison. Remote Sensing, 14(11). https://doi.org/10.3390/rs14112623

Boehler, W., Marbs, A., & Bordas, V. (2018). Ogc testbed-14: Point cloud data handling engineering report. Technical Report, I3mainz.

Boehm, J. (2014). File-centric organization of large LiDAR Point Clouds in a Big Data context. Proceedings of the IQmulus First Workshop on Processing Large Geospatial Data, Cardiff, UK, 8, 69-76.

Boehm, J., & Liu, K. (2015). NOSQL FOR STORAGE AND RETRIEVAL OF LARGE LIDAR DATA COLLECTIONS. The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, XL-3/W3, 577-582. https://doi.org/10.5194/isprsarchives-XL-3-W3-577-2015

Boehm, J., Liu, K., & Alis, C. (2016). SIDELOADING – INGESTION OF LARGE POINT CLOUDS INTO THE APACHE SPARK BIG DATA ENGINE. The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, XLI-B2, 343-348. https://doi.org/10.5194/isprs-archives-XLI-B2-343-2016

Bosché, F. (2012). Plane-based registration of construction laser scans with 3D/4D building models. Adv. Eng. Inform., 26(1), 90-102. https://doi.org/10.1016/j.aei.2011.08.009

Boulch, A. (2020). ConvPoint: Continuous Convolutions for Point Cloud Processing. Computers & Graphics, 88. https://doi.org/10.1016/j.cag.2020.02.005

Bunting, P., Armston, J., Lucas, R., & Clewley, D. (2013). Sorted Pulse Data (SPD) Library. Part I: A generic file format for LiDAR data from pulse laser systems in terrestrial environments. Computers and Geosciences, 56, 197-206. https://doi.org/10.1016/j.cageo.2013.01.019

Cao, C., Preda, M., & Zaharia, T. (2019). 3D Point Cloud Compression: A Survey. 1-9. https://doi.org/10.1145/3329714.3338130

Chen, D., Zhang, L., Mathiopoulos, P. T., & Huang, X. (2014). A Methodology for Automated Segmentation and Reconstruction of Urban 3-D Buildings from ALS Point Clouds. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, 7(10), 4199-4217. https://doi.org/10.1109/JSTARS.2014.2349003

Cura, R., Perret, J., & Paparoditis, N. (2015). POINT CLOUD SERVER (PCS): POINT CLOUDS IN-BASE MANAGEMENT AND PROCESSING. ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences, II-3/W5, 531-539. https://doi.org/10.5194/isprsannals-II-3-W5-531-2015

Cyber-archaeology, big data and the race to save threatened cultural heritage sites—Phys.org. (s. f.). https://phys.org/news/2016-02-cyber-archaeology-big-threatened-cultural-heritage.html

Dai, A., Ritchie, D., Bokeloh, M., Reed, S., Sturm, J., & Nießner, M. (2017). ScanComplete: Large-Scale Scene Completion and Semantic Segmentation for 3D Scans.

de Gélis, I., Lefèvre, S., & Corpetti, T. (2021). Change Detection in Urban Point Clouds: An Experimental Comparison with Simulated 3D Datasets. Remote Sensing, 13(13). https://doi.org/10.3390/rs13132629

de Queiroz, R. L., & Chou, P. A. (2016). Compression of 3D point clouds using a region-adaptive hierarchical transform. IEEE Transactions on Image Processing, 25(8), 3947-3956.

Deibe, D., Amor, M., & Doallo, R. (2018). Big data storage technologies: A case study for web-based LiDAR visualization. 3831-3840. https://doi.org/10.1109/BigData.2018.8622589

Deibe, D., Amor, M., & Doallo, R. (2019). Supporting multi-resolution out-of-core rendering of massive LiDAR point clouds through non-redundant data structures. International Journal of Geographical Information Science, 33(3), 593-617. https://doi.org/10.1080/13658816.2018.1549734

Deiotte, R., & La Valley, R. (2017). COMPARISON OF SPATIOTEMPORAL MAPPING TECHNIQUES FOR ENORMOUS ETL AND EXPLOITATION PATTERNS. ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences, IV-4/W2, 7-13. https://doi.org/10.5194/isprs-annals-IV-4-W2-7-2017

Du, J., Jiang, Z., Huang, S., Wang, Z., Su, J., Su, S., Wu, Y., & Cai, G. (2021). Point Cloud Semantic Segmentation Network Based on Multi-Scale Feature Fusion. Sensors, 21(5). https://doi.org/10.3390/s21051625

Elseberg, J., Borrmann, D., & Nüchter, A. (2013). One billion points in the cloud–an octree for efficient processing of 3D laser scans. ISPRS Journal of Photogrammetry and Remote Sensing, 76, 76-88.

Evans, M. R., Oliver, D., Zhou, X., & Shekhar, S. (2014). Spatial big data: Case studies on volume, velocity, and variety. En Big Data (pp. 149-176). CRC Press. https://doi.org/10.1201/b16524

Fernandez, A. (2021). Loadtest [Software]. https://github.com/alexfernandez/loadtest

Gong, J., Zhu, Q., Zhong, R., Zhang, Y., & Xie, X. (2012). An Efficient Point Cloud Management Method Based on a 3D R-Tree. Photogrammetric Engineering and Remote Sensing, 74, 373-381. https://doi.org/10.14358/PERS.78.4.373

Gonzalez, M., Lucena López, M., Fuertes, J. M., Rueda, A., & Segura, R. (2012). Automatic detection of unstructured elements in 3D scanned scenes. Automation in Construction, 26, 11-20. https://doi.org/10.1016/j.autcon.2012.05.005

Goswami, P., Erol, F., Mukhi, R., Pajarola, R., & Gobbetti, E. (2012). An efficient multi-resolution framework for high quality interactive rendering of massive point clouds using multi-way kd-trees. The Visual Computer, 29. https://doi.org/10.1007/s00371-012-0675-2

Grilli, E., Menna, F., & Remondino, F. (2017). A REVIEW OF POINT CLOUDS SEGMENTATION AND CLASSIFICATION ALGORITHMS. ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, XLII-2/W3, 339-344. https://doi.org/10.5194/isprs-archives-XLII-2-W3-339-2017

Gross, M., & H. Pfister (eds.), M. K. (2007). Point-Based Graphics. Elsevier.

Günther, C., Kanzok, T., Linsen, L., & Rosenthal, P. (2013). A GPGPU-based Pipeline for Accelerated Rendering of Point Clouds. Journal of WSCG, 21, 153.

Hackel, T., Savinov, N., Ladicky, L., Wegner, J. D., Schindler, K., & Pollefeys, M. (2017). Semantic3D.net: A new Large-scale Point Cloud Classification Benchmark. CoRR, abs/1704.03847. http://arxiv.org/abs/1704.03847

Hobu Inc. (2016). Entwine. https://entwine.io/en/latest/

Hongchao, M., & Wang, Z. (2011). Distributed data organization and parallel data retrieval methods for huge laser scanner point clouds. Computers & Geosciences, 37(2), 193-201. https://doi.org/10.1016/j.cageo.2010.05.017

Huang, H., Li, D., Zhang, H., Ascher, U., & Cohen-Or, D. (2009). Consolidation of unorganized point clouds for surface reconstruction. ACM Trans. Graph., 28(5), 1-7. https://doi.org/10.1145/1618452.1618522

Huang, L., Wang, S., Wong, K., Liu, J., & Urtasun, R. (2021). OctSqueeze: Octree-Structured Entropy Model for LiDAR Compression.

Isenburg, M. (2013). LASzip: Lossless compression of LiDAR data. Photogrammetric Engineering & Remote Sensing, 79. https://doi.org/10.14358/PERS.79.2.209

Józsa, O., Börcs, A., & Benedek, C. (2013). TOWARDS 4D VIRTUAL CITY RECONSTRUCTION FROM LIDAR POINT CLOUD SEQUENCES. ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences, II-3/W1, 15-20. https://doi.org/10.5194/isprsannals-II-3-W1-15-2013

Kadobayashi, R., Kochi, N., Otani, H., & Ryo, F. (2004). COMPARISON AND EVALUATION OF LASER SCANNING AND PHOTOGRAMMETRY AND THEIR COMBINED USE FOR DIGITAL RECORDING OF CULTURAL HERITAGE. https://api.semanticscholar.org/CorpusID:10459478

Kämpe, V., Sintorn, E., & Assarsson, U. (2013). High resolution sparse voxel DAGs. ACM Trans. Graph., 32(4). https://doi.org/10.1145/2461912.2462024

Kass, M., Witkin, A., & Terzopoulos, D. (1988). Snakes: Active contour models. International journal of computer vision, 1(4), 321-331.

Khan, W., Kumar, T., Zhang, C., Raj, K., Roy, A. M., & Luo, B. (2023). SQL and NoSQL Database Software Architecture Performance Analysis and Assessments—A Systematic

Literature Review. Big Data and Cognitive Computing, 7(2). https://doi.org/10.3390/bdcc7020097

Krämer, M., & Senner, I. (2015). A modular software architecture for processing of big geospatial data in the cloud. Computers & Graphics, 49, 69-81. https://doi.org/10.1016/j.cag.2015.02.005

Lakshman, A., & Malik, P. (2010). Cassandra: A decentralized structured storage system. SIGOPS Oper. Syst. Rev., 44(2), 35-40. https://doi.org/10.1145/1773912.1773922

Lee, J.-G., & Kang, M. (2015). Geospatial Big Data: Challenges and Opportunities. Big Data Research, 2(2), 74-81. https://doi.org/10.1016/j.bdr.2015.01.003

Li, J., Chen, B. M., & Lee, G. H. (2018). SO-Net: Self-Organizing Network for Point Cloud Analysis. 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, 9397-9406. https://doi.org/10.1109/CVPR.2018.00979

Li, M. (2018). A SUPER VOXEL-BASED RIEMANNIAN GRAPH FOR MULTI SCALE SEGMENTATION OF LIDAR POINT CLOUDS. ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences, IV-3, 135-141. https://doi.org/10.5194/isprs-annals-IV-3-135-2018

Li, Y., Ma, L., Zhong, Z., Cao, D., & Li, J. (2019). TGNet: Geometric Graph CNN on 3-D Point Cloud Segmentation. IEEE Transactions on Geoscience and Remote Sensing, PP, 1-13. https://doi.org/10.1109/TGRS.2019.2958517

Liu, Y., & Xiong, Y. (2008). Automatic segmentation of unorganized noisy point clouds based on the Gaussian map. Computer-Aided Design, 40(5), 576-594. https://doi.org/10.1016/j.cad.2008.02.004

Lu, B., Wang, Q., & Li, A. (2019). Massive Point Cloud Space Management Method Based on Octree-Like Encoding. Arabian Journal for Science and Engineering, 44(11), 9397-9411. https://doi.org/10.1007/s13369-019-03968-7

Lu, D., Mausel, P., Brondízio, E., & Moran, E. (2004). Change detection techniques. International Journal of Remote Sensing, 25(12), 2365-2401. https://doi.org/10.1080/0143116031000139863

Maturana, D., & Scherer, S. (2015). VoxNet: A 3D Convolutional Neural Network for real-time object recognition. 922-928. https://doi.org/10.1109/IROS.2015.7353481

Morell, V., Orts, S., Cazorla, M., & Garcia-Rodriguez, J. (2014). Geometric 3D point cloud compression. Pattern Recognition Letters, 50, 55-62. https://doi.org/10.1016/j.patrec.2014.05.016

Nguyen, A., & Le, B. (2013). 3D point cloud segmentation: A survey. 2013 6th IEEE Conference on Robotics, Automation and Mechatronics (RAM), 225-230. https://doi.org/10.1109/RAM.2013.6758588

Nguyen, V. S., Bac, A., & Daniel, M. (2014). Triangulation of an elevation surface structured by a sparse 3D grid. 2014 IEEE Fifth International Conference on Communications and Electronics (ICCE), 464-469. https://doi.org/10.1109/CCE.2014.6916749

Niemeyer, G. (2008). Geohash. Retrieved June, 6, 2018.

Nina, A., Jianhua, W., Mark, P., Gaël, G., Mathias, P., Marc, A., Leif, K., Renato, P., & Loïc, B. (2007). 4—FOUNDATIONS AND REPRESENTATIONS. En M. GROSS & H.

PFISTER (Eds.), Point-Based Graphics (pp. 94-184). Morgan Kaufmann. https://doi.org/10.1016/B978-012370604-1/50005-5

Ogayar-Anguita, C. J., López-Ruiz, A., Rueda-Ruiz, A. J., & Segura-Sánchez, R. J. (2023). Nested spatial data structures for optimal indexing of LiDAR data. ISPRS Journal of Photogrammetry and Remote Sensing, 195, 287-297. https://doi.org/10.1016/j.isprsjprs.2022.11.018

Ogayar-Anguita, C. J., López-Ruiz, A., Segura-Sánchez, R. J., & Rueda-Ruiz, A. J. (2023). A Version Control System for Point Clouds. Remote Sensing, 15(18), 4635. https://doi.org/10.3390/rs15184635

OGC. (2019). OGC Hierarchical Data Format Version 5. https://docs.ogc.org/is/18-043r3/18-043r3.html

Pääkkönen, P., & Pakkala, D. (2015). Reference Architecture and Classification of Technologies, Products and Services for Big Data Systems. Big Data Research, 2(4), 166-186. https://doi.org/10.1016/j.bdr.2015.01.001

Park, H., Chang, M., & Park, S. (2007). A slicing algorithm of point cloud for rapid prototyping. Summer Computer Simulation Conference 2007, SCSC'07, Part of the 2007 Summer Simulation Multiconference, SummerSim'07, 2, 24. https://doi.org/10.1145/1357910.1358130

PNOA. (2024). Centro de descarga de nubes de puntos LiDAR del Plan Nacional de Ortofotografía Aérea (PNOA). Organismo Autónomo Centro Nacional de Información Geográfica: [dataset]. http://centrodedescargas.cnig.es/CentroDescargas/catalogo.do?Serie=MDT02

Poux, F. (2019). The Smart Point Cloud: Structuring 3D intelligent point data. https://doi.org/10.13140/RG.2.2.20457.75367

Preiner, R., Jeschke, S., & Wimmer, M. (2012). Auto Splats: Dynamic Point Cloud Visualization on the GPU. Eurographics Symposium on Parallel Graphics and Visualization, 10 pages. https://doi.org/10.2312/EGPGV/EGPGV12/139-148

Qi, C. R., Yi, L., Su, H., & Guibas, L. J. (2017). PointNet++: Deep Hierarchical Feature Learning on Point Sets in a Metric Space. CoRR, abs/1706.02413. http://arxiv.org/abs/1706.02413

Qin, R., Tian, J., & Reinartz, P. (2016). 3D change detection – Approaches and applications. Isprs Journal of Photogrammetry and Remote Sensing, 122, 41-56.

Rabbani, T., Heuvel, F. A., & Vosselman, G. (2006). Segmentation of point clouds using smoothness constraint. International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences, 36.

Rachit Pandey. (2020). Performance Benchmarking and Comparison of Cloud-Based Databases MongoDB (NoSQL) Vs MySQL (Relational) using YCSB. https://doi.org/10.13140/RG.2.2.10789.32484

Reed, C., & Belayneh, T. (2017). OGC Indexed 3d Scene Layer (I3S) and Scene Layer Package Format Specification. Tech. rep. Open Geospatial Consortium.

Reitz, K. (2023). Requests [Software]. https://requests.readthedocs.io/en/latest/

Reutegger, M. (2023). Laszip4j [Software]. https://github.com/mreutegg/laszip4j

Richter, R., Discher, S., & Döllner, J. (2015). Out-of-Core Visualization of Classified 3D Point Clouds. En M. Breunig, M. Al-Doori, E. Butwilowski, P. V. Kuper, J. Benner, & K. H. Haefele (Eds.), 3D Geoinformation Science (pp. 227-242). Springer International Publishing. https://doi.org/10.1007/978-3-319-12181-9_14

Rueda-Ruiz, A. J., Ogáyar-Anguita, C. J., Segura-Sánchez, R. J., Béjar-Martos, J. A., & Delgado-Garcia, J. (2022). SPSLiDAR: Towards a multi-purpose repository for large scale LiDAR datasets. International Journal of Geographical Information Science, 36(5), 992-1011. https://doi.org/10.1080/13658816.2022.2030479

Rueda-Ruiz, A. J., Sánchez, R. J. S., Ogáyar, C. J., Muñoz, M. J. G., Zafra, J. M. V., Hoyas, A. E., Ramírez, A. R., & Mandly, F. N. (2009). Tratamiento de datos procedentes de escáneres láser de alcance medio con scancyr. Mapping, 132, 6-10.

Rusu, R. B., & Cousins, S. (2011, mayo 9). 3D is here: Point Cloud Library (PCL). IEEE International Conference on Robotics and Automation (ICRA).

Scheiblauer, C., & Wimmer, M. (2011). Cultural Heritage: Out-of-core selection and editing of huge point clouds. Comput. Graph., 35(2), 342-351. https://doi.org/10.1016/j.cag.2011.01.004

Schmohl, S., & Sörgel, U. (2019). SUBMANIFOLD SPARSE CONVOLUTIONAL NETWORKS FOR SEMANTIC SEGMENTATION OF LARGE-SCALE ALS POINT CLOUDS. ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences, IV-2/W5, 77-84. https://doi.org/10.5194/isprs-annals-IV-2-W5-77-2019

Schnabel, R., Degener, P., & Klein, R. (2009). Completion and Reconstruction with Primitive Shapes. Computer Graphics Forum, 28(2), 503-512. https://doi.org/10.1111/j.1467-8659.2009.01389.x

Schnabel, R., & Klein, R. (2006). Octree-based Point-Cloud Compression. En M. Botsch, B. Chen, M. Pauly, & M. Zwicker (Eds.), Symposium on Point-Based Graphics. The Eurographics Association. https://doi.org/10.2312/SPBG/SPBG06/111-120

Schön, B., Mosa, A. S. M., Laefer, D. F., & Bertolotto, M. (2013). Octree-based indexing for 3D pointclouds within an Oracle Spatial DBMS. Computers & Geosciences, 51, 430-438. https://doi.org/10.1016/j.cageo.2012.08.021

Schütz, M. (2016). Potree: Rendering Large Point Clouds in Web Browsers [PhD Thesis].

Schütz, M., Ohrhallinger, S., & Wimmer, M. (2020). Fast Out-of-Core Octree Generation for Massive Point Clouds. Computer Graphics Forum, 39(7), 155-167. https://doi.org/10.1111/cgf.14134

Schutz, M., & Wimmer, M. (2019). Rendering Point Clouds with Compute Shaders. SIGGRAPH Asia 2019 Posters. https://doi.org/10.1145/3355056.3364554

Shi, W., Zhang, M., Zhang, R., Chen, S., & Zhan, Z. (2020). Change Detection Based on Artificial Intelligence: State-of-the-Art and Challenges. Remote Sensing, 12(10), 1688. https://doi.org/10.3390/rs12101688

Si Salah, H., Ait-Aoudia, S., Rezgui, A., & Goldin, S. E. (2019). Change detection in urban areas from remote sensing data: A multidimensional classification scheme. International

Journal of Remote Sensing, 40(17), 6635-6679. https://doi.org/10.1080/01431161.2019.1583394

Song, S., Yu, F., Zeng, A., Chang, A. X., Savva, M., & Funkhouser, T. (2016). Semantic Scene Completion from a Single Depth Image. https://doi.org/10.48550/ARXIV.1611.08974

Sugimoto, K., Cohen, R. A., Tian, D., & Vetro, A. (2017). Trends in efficient representation of 3D point clouds. 2017 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC), 364-369. https://doi.org/10.1109/APSIPA.2017.8282059

Terzopoulos, D., & Szeliski, R. (1992). Tracking with Kalman snakes. Active vision, 20, 3-20.

The HDF Group. (2024). Hierarchical Data Format, version 5 [Software]. https://github.com/HDFGroup/hdf5

Tian, S., Li, X., Zeng, J., & Wei, Z. (2019). The Organization of Point Cloud Data Based on the Compact Octree Model. Journal of Physics: Conference Series, 1302(2), 022047. https://doi.org/10.1088/1742-6596/1302/2/022047

Tóvári, D., & Pfeifer, N. (2005). Segmentation based robust interpolation-a new approach to laser data filtering. International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences, 36(3/19), 79-84.

Tran, T., Ressl, C., & Pfeifer, N. (2018). Integrated Change Detection and Classification in Urban Areas Based on Airborne Laser Scanning Point Clouds. Sensors, 18(2), 448. https://doi.org/10.3390/s18020448

Turk, G., & Levoy, M. (1994). Zippered polygon meshes from range images. Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques, 311-318. https://doi.org/10.1145/192161.192241

Van Oosterom, P., Martinez-Rubi, O., Ivanova, M., Horhammer, M., Geringer, D., Ravada, S., Tijssen, T., Kodde, M., & Gonçalves, R. (2015). Massive point cloud data management: Design, implementation and execution of a point cloud benchmark. Computers & Graphics, 49, 92-125. https://doi.org/10.1016/j.cag.2015.01.007

Vosselman, G., Gorte, B. G. H., Sithole, G., & Rabbani, T. (2004). Recognising structure in laser scanning point clouds. https://api.semanticscholar.org/CorpusID:7469373

Wand, M., Berner, A., Bokeloh, M., Jenke, P., Fleck, A., Hoffmann, M., Maier, B., Staneker, D., Schilling, A., & Seidel, H.-P. (2008). Processing and interactive editing of huge point clouds from 3D scanners. Computers & Graphics, 32(2), 204-220.

Wang, C., Hu, F., Sha, D., & Han, X. (2017). EFFICIENT LIDAR POINT CLOUD DATA MANAGING AND PROCESSINGIN A HADOOP-BASED DISTRIBUTED FRAMEWORK. ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences, IV-4/W2, 121-124. https://doi.org/10.5194/isprs-annals-IV-4-W2-121-2017

Wang, X., Liu, X., & Qin, H. (2013). Robust Surface Consolidation of Scanned Thick Point Clouds. 2013 International Conference on Computer-Aided Design and Computer Graphics, 38-43. https://doi.org/10.1109/CADGraphics.2013.12

Wang, Y., Lv, H., & Ma, Y. (2020). Geological tetrahedral model-oriented hybrid spatial indexing structure based on Octree and 3D R*-tree. Arabian Journal of Geosciences, 13(15), 728. https://doi.org/10.1007/s12517-020-05752-6

Wang, Y., Sun, Y., Liu, Z., Sarma, S. E., Bronstein, M. M., & Solomon, J. M. (2018). Dynamic Graph CNN for Learning on Point Clouds. https://doi.org/10.48550/ARXIV.1801.07829

Wavefront Technologies, Inc. (1990). OBJ (Wavefront Object) File Format Specification. https://www.fileformat.info/format/wavefrontobj/

Yan, X., Zheng, C., Li, Z., Wang, S., & Cui, S. (2020). PointASNL: Robust Point Clouds Processing using Nonlocal Neural Networks with Adaptive Sampling. https://doi.org/10.48550/ARXIV.2003.00492

Yang, B., Huang, R., Li, J., Tian, M., Dai, W., & Zhong, R. (2016). Automated Reconstruction of Building LoDs from Airborne LiDAR Point Clouds Using an Improved Morphological Scale Space. Remote Sensing, 9(1), 14. https://doi.org/10.3390/rs9010014

Yang, J., & Huang, X. (2014). A Hybrid Spatial Index for Massive Point Cloud Data Management and Visualization. Transactions in GIS, 18(S1), 97-108. https://doi.org/10.1111/tgis.12094

Yoo, J. H., Kim, Y., Kim, J., & Choi, J. W. (2020). 3D-CVF: Generating Joint Camera and LiDAR Features Using Cross-View Spatial Feature Fusion for 3D Object Detection. https://doi.org/10.48550/ARXIV.2004.12636

Zhang, R., Li, G., Li, M., & Wang, L. (2018). Fusion of images and point clouds for the semantic segmentation of large-scale 3D scenes based on deep learning. ISPRS Journal of Photogrammetry and Remote Sensing, 143, 85-96. https://doi.org/10.1016/j.isprsjprs.2018.04.022

Zhou, Y., & Tuzel, O. (2017). VoxelNet: End-to-End Learning for Point Cloud Based 3D Object Detection. https://doi.org/10.48550/ARXIV.1711.06396