ISPRS Journal of Photogrammetry and Remote Sensing 195 (2023) 287–297

Contents lists available at ScienceDirect

ISPRS Journal of Photogrammetry and Remote Sensing

journal homepage: www.elsevier.com/locate/isprsjprs

# Nested spatial data structures for optimal indexing of LiDAR data ☆

Carlos J. Ogayar-Anguita *, Alfonso López-Ruiz, Antonio J. Rueda-Ruiz, Rafael J. Segura-Sánchez

*Department of Computer Science, University of Jaén, EPS Jaén, 23071, Spain*

ARTICLE INFO

*Keywords:*
Spatial data structure
Spatial big data
Ubiquitous Point Cloud
LiDAR

ABSTRACT

In this paper we present a flexible framework for creating spatial data structures to manage LiDAR point clouds in the context of spatial big data. For this purpose, standard approaches typically include the use of a single data structure to index point clouds. Some of them use a hybrid two-tier solution to optimize specific application purposes such as storage or rendering. In this article we introduce a meta-structure that can have unlimited depth and a custom, user-defined combination of nested structures, such as grids, quadtrees, octrees, or kd-trees. With our approach, the out-of-core indexing of point clouds can be adapted to different types of datasets, taking into account the spatial distribution of the data. Therefore, the most suitable spatial indexing can be achieved for any type of dataset, from small TLS-based scenes to planetary-scale ALS-based scenes. This approach allows us to work with overlapping datasets of different resolutions from different acquisition technologies in the same structure.

## 1. Introduction

LiDAR scanning is one of the most powerful tools in fields such as civil engineering, surveying, archaeology or environmental engineering. Due to the evolution, popularization and cheapening of LiDAR sensors, nowadays it is very common to have large datasets, especially in the context of the so-called Geospatial Big Data. Handling this amount of data brings with it a number of problems, related to its storage, transmission, organization, visualization, edition and analysis. The situation worsens when it comes to managing data obtained from different sources with different precision levels, such as ALS (Airborne Laser Scanning) and TLS (Terrestrial Laser Scanning), managing data acquired at different times, or having different layers of information. Thus, the data structure used to index all this spatial data is paramount in order to efficiently access the desired information for any processing task.

Typically, LiDAR data is stored as files in a common format, such as LAS/LAZ. The usual way to define the work area is by means of a polygon or a bounding box. As data is usually distributed in tiles (Boehm, 2014), it may be necessary to process much more data than actually affected by the operation. Furthermore, datasets eventually take up terabytes of information scattered across diverse storage media in hundreds of files. This makes it difficult to perform spatial and temporal analyses and data editing. To alleviate this, it is common practice to have different levels of detail of the dataset, by using sub-sampling. In contrast, some approaches use a hierarchical data structure to have a more efficient spatial distribution of the data (Deibe et al., 2019; Huang et al., 2020; Lu et al., 2019; Schuetz, 2016; Schütz et al., 2020), which implies having levels of details (LODs) in an intrinsic way. However, some problems remain, especially when combining multiple datasets generated with different scanning precision or different attributes. This is common in the context of Geospatial Big Data, and therefore, specific approaches and techniques must be applied (Deng et al., 2019; Evans et al., 2014; Lee and Kang, 2015; Pääkkönen and Pakkala, 2015).

The Open Geospatial Consortium (OGC) Testbed-14, in the Point Cloud Data Handling Engineering Report (Boehler et al., 2018), identifies some applications and tasks that need spatially accelerated accesses, such as modeling and simulation, measurement, feature extraction, change detection or bathymetric exploration among others. Therefore, the mere data storage is not the only important issue, but also efficient data search, retrieval and editing. The main problem with standard spatial indexing solutions is their poor adaptation to some datasets, due to the spatial distribution of the data. For example, indexing large terrains scanned with ALS using an octree will produce an excessive depth of the structure, due to its inadequate adaptation to mostly flat geometry. On the other hand, a quadtree will perform poorly when the dataset contains a large number of vertical structures (see

☆ This result is part of the research project RTI2018-099638-B-I00 funded by MCIN/AEI/10.13039/501100011033/ and ERDF funds "A way of doing Europe". Also, the work has been funded by the Spanish Ministry of Science, Innovation and Universities via a doctoral grant to the second author (FPU19/00100), and the University of Jaén (via ERDF funds) through the research project 1265116/2020.

* Corresponding author.
*E-mail addresses:* cogayar@ujaen.es (C.J. Ogayar-Anguita), allopezr@ujaen.es (A. López-Ruiz), ajrueda@ujaen.es (A.J. Rueda-Ruiz), rsegura@ujaen.es (R.J. Segura-Sánchez).

https://doi.org/10.1016/j.isprsjprs.2022.11.018
Received 8 February 2022; Received in revised form 12 September 2022; Accepted 21 November 2022
Available online 9 December 2022

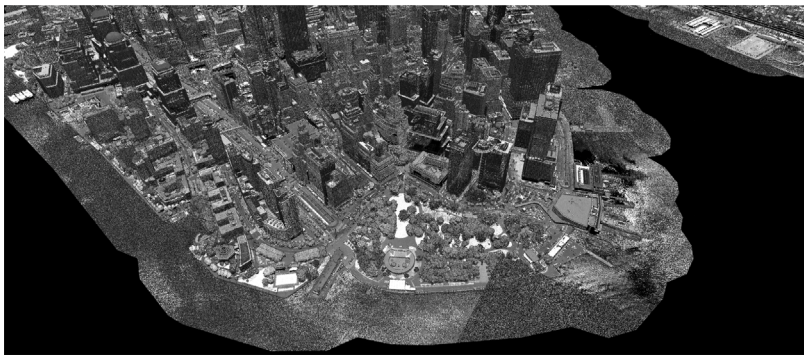

**Fig. 1.** The Manhattan Island LiDAR dataset used in the experiments. ©2022 City of New York, NYC Open Data.

Fig. 1), as happens with urban scenarios where data from the interior of buildings is included through TLS. Unfortunately past experience has shown that the appropriate spatial structure is highly dependent on the type of dataset (Poux, 2019). The main problem appears when working with data coming from various sources (ALS, TLS, mobile mapping, etc.), multiple accuracy levels, and especially several types of spatial distributions, all in the same dataset and eventually the same applications.

The objective of this work is the definition of a flexible spatial meta-structure for the managing of LiDAR point clouds in the context of Big Data at a variable scale, ranging from small TLS-based scenes to planet-scale ALS-based datasets. This structure serves as a spatial indexer, and is used for partitioning the data in a way that speeds up spatial queries and data accesses. It allows a user-defined combination of nested structures with an unlimited depth, such as grids, quadtrees, octrees, kd-trees and others. With this approach, the out-of-core indexing of point clouds can be optimized to a given dataset, by adapting the spatial partition to the morphology and distribution of points in space. For example, large portions of land can be treated efficiently with 2D-based spatial indexing, while vertical structures such as buildings can be optimally partitioned and indexed in 3D.

A significant advantage of mixing multiple data structures is the optimal adaptation to levels of detail with a certain purpose, for example reserving a type of spatial structure only to index most dense zones (e.g. buildings). With this approach, it is easier to adjust the level of detail of the environment, which improves all dataset processing tasks, including editing, data analysis and visualization (Poux, 2019). This also improves the integration of data obtained from TLS scans together with the more dispersed data obtained with ALS.

The rest of the paper is organized as follows. Section 2 reviews the most relevant work related to point cloud storage and spatial indexing. Section 3 presents the description of the proposed approach of nested spatial data structures, which is tested and analyzed in Section 4. Finally, Section 5 presents the conclusions and outlines future work.

## 2. Previous works

Point cloud models are a widely used resource in all types of decision-making processes (Poux, 2019). With the rapid evolution of 3D data capture hardware, larger and larger datasets are becoming available, which pose a major challenge for efficient data management and processing (Poux, 2019). In addition to the increase in the volume of spatial information captured, there is also a great variety in its scale, from small objects to large areas of the territory (Bräunl, 2020). This allows us to acquire massive digital information from the real environment, which is integrated into the Geospatial Big Data (Deng et al., 2019; Evans et al., 2014; Lee and Kang, 2015; Pääkkönen and Pakkala, 2015). Related to this, the concept of Ubiquitous Point Clouds (Liang et al., 2016; Šašak et al., 2019; Lin et al., 2020) introduces a new perspective for point cloud management in a multidimensional way (3D coordinates, timestamps and additional attributes). Its applications include urban planning, archaeological sites, civil engineering, surveying, forestry, BIM and Scada systems, digital twins, among others. It highlights not only the volume of data but also the dispersion throughout the territory and over time, the variety of capture devices (and therefore the quality) and the purpose of the capture itself.

Point cloud processing involves a series of steps from its capture to the extraction of the desired information, including registering, filtering, segmentation, classification and conversion to other 2D/3D representation schemas (Józsa et al., 2013; Ullrich and Pfennigbauer, 2019). Some advanced functionalities that benefit from improvements in this regard are temporary information management (4D) and collaborative editing work, where it is essential to have efficient space partitions that allow editing locks and the setting of access permissions.

The spatial data structure used for point cloud management has a determining influence on all the processes that can be carried out. Most research proposals define a data structure that is strongly tied to a specific application, leaving aside other important uses. This is very common in rendering-oriented systems (Deibe et al., 2019; Goswami et al., 2013; Preiner et al., 2012; Schuetz, 2016; Richter et al., 2015; Schutz et al., 2019; Ströter et al., 2020), which in some cases are not suitable for other purposes such as optimal storage or data analysis. On the other hand, there are several papers that focus on mass storage in secondary memory, in the network or in the cloud (Baert et al., 2014; Deibe et al., 2019; Richter et al., 2015; Scheiblauer and Wimmer, 2011; Schütz et al., 2020).

Most authors propose solutions for the spatial indexing mainly based on grids and quadtrees for planar data (Boehm et al., 2016; Deibe et al., 2019; Hongchao and Wang, 2011), or octrees for volumetric data (Elseberg et al., 2013; Huang et al., 2020; Lu et al., 2019; Scheiblauer and Wimmer, 2011; Schön et al., 2013; Schütz et al., 2020; Ströter et al., 2020; Tian et al., 2019). These spatial indexing solutions are perhaps somewhat simple, taking into account the increasing complexity of the datasets and the processes to be carried out on them. The most widely used data structure for point cloud storage and rendering is the octree. Other approaches are based on the k-d tree, which optimizes space partitioning (Goswami et al., 2013), a sparse voxel structure (Baert et al., 2014; Kämpe et al., 2013; Poux and Billen, 2019) or a 2D grid (Hongchao and Wang, 2011). There are other data structures that work better with whole objects, such as the r-tree. They are normally used with polygon meshes, although there are works that successfully use some variants with point clouds (Gong et al., 2012), as well as others that could be adapted for this purpose (Wang et al., 2020). Fig. 2 shows examples of some basic spatial structures.

Some of the cited proposals include the use of a nested or hybrid spatial structure, mainly oriented to rendering (Deibe et al., 2019; Schuetz, 2016; Yang and Huang, 2014) and to the optimization of search operations (Lu et al., 2019; Scheiblauer and Wimmer, 2011). Most of these methods tend to keep the spatial data structure as simple

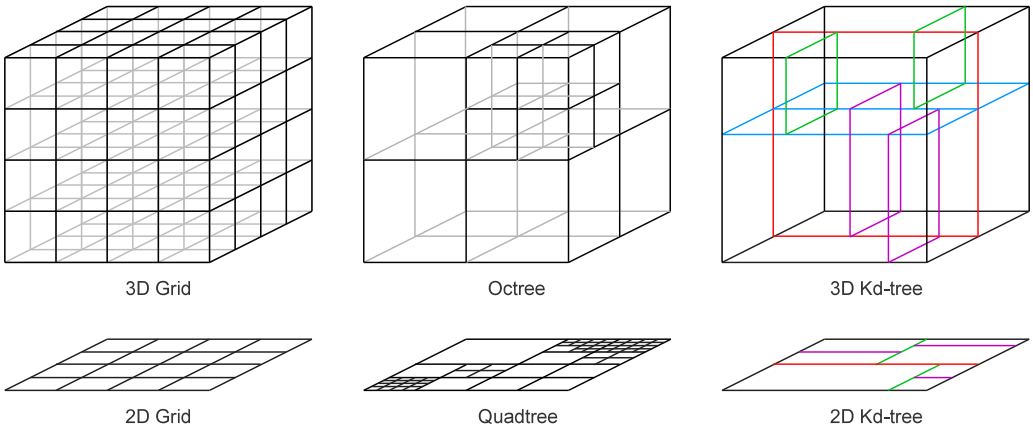


**Fig. 2.** Commonly used basic spatial data structures.

as possible, while allowing to work with LODs (Level of Details) (Deibe et al., 2019; Schuetz, 2016). The main drawback is that the spatial indexing does not adapt well to all possible point spatial distributions. The most relevant case is the poor adaptation of the octree to the spatial distribution of points along large areas of scanned territory. In these cases quadtree and grid planar structures are usually more appropriate (Boehm et al., 2016; Deibe et al., 2019; Hongchao and Wang, 2011). However, if a dataset has multiple point distribution patterns, no single structure will be able to optimally fit in all zones, that is, with the lowest possible node count.

Some of the papers mentioned above only address point cloud indexing in memory, while others do it at the out-of-core storage level as well. Simplest approaches use direct storage in secondary memory using local files (Scheiblauer and Wimmer, 2011; Schütz et al., 2020), while others rely on using distributed file systems (Deibe et al., 2018; Krämer and Senner, 2015), typically indexed through a spatial structure, which is implemented as a master index file or stored in a database. There are several well-known file formats for storing point clouds and LiDAR data. The most common options are non-standard ASCII formats, the LAS format of the American Society for Photogrammetry and Remote Sensing (ASPRS), its compressed variant LAZ (Isenburg, 2013), SPD (Bunting et al., 2013), PCD, HDF5 and other general 3D data file formats such as OBJ, STL or PLY. From a practical standpoint, the most interesting formats are those that allow data compression (Cao et al., 2019; Isenburg, 2013; Sugimoto et al., 2017). Any of these options is valid for storing groups of points contained in the nodes of a spatial structure used in secondary memory. In this work we use the LAZ format (Isenburg, 2013), that achieves very efficient compression of LiDAR data and is the de facto standard based on the LAS format (Boehm, 2014). In addition to file-oriented storage, the usual alternative is to use databases (Boehm, 2014; Deibe et al., 2018; Hongchao and Wang, 2011; Lee and Kang, 2015; Schön et al., 2013). Although there are several approaches to storing LiDAR points in a database, one of the most efficient methods is to store point clouds as a block in the database or as files in a file system, though indexed in the database.

## 3. Nested spatial data structures schema

Our proposal consists of the creation of ad-hoc combinations of nested data structures to take full advantage of the spatial distribution of each point cloud dataset. The nesting of data structures is a problem that has a certain complexity. In this section we present a framework for achieving that goal. The particularities of each type of spatial structure, such as the grid, the octree or the kd-tree, are outside the scope of this work. Here we will focus on how to adapt them to make a combination of nested structures.

The key aspect is to design what we denote as the *global structure*, that is, a combination of nested structures. Each of the component structures is a *inner structure*, which can be any user-defined spatial structure. In this work we have implemented adapted versions of grids, kd-trees, quadtrees and octrees, but other spatial structures can also be integrated into this framework. Each possible combination of inner structures (a global structure) must be defined by the user following certain criteria that should depend on the specific dataset and application requirements. Throughout this section the criteria to be taken into account will also be discussed. Fig. 3 shows all the concepts presented here through an example. Each inner structure has its own depth levels, which are denoted as *local levels*. Those same levels also take a place in the global scheme, where they are considered as *global levels*. Spatial structure nesting implies that data is propagated down the structure until leaf nodes are reached. Then, for each leaf node it must be calculated whether a new lower level is needed in order to distribute the data into smaller subsets. Each time a new level is created during the spatial subdivision, the inner structure that owns a given node creates a new level at that node, only if it is a hierarchical structure and the maximum depth is not reached. If it is the case, the global structure schema is queried for the next inner structure in order to instantiate and attach it to that node (see an example in Fig. 3).

In order to combine and connect all the inner structures, some design principles must be applied to their implementation. In this way, we have implemented specific adaptations of the structures, so grid, quadtrees, octrees, etc. must be adapted versions whose particularities are explained next. Firstly, we assume that all the nodes in every spatial structure are of the same size in every dimension, that is, square for planar structures and cubic for volumetric structures. This greatly simplifies the rest of decisions to be made, and almost all other solutions in the literature follow this same convention. Secondly, the global hierarchical structure schema, that is defined by the user, must be the same throughout the complete dataset. It is also fixed throughout the life of the dataset. That means that during its construction, when a given spatial structure reaches a leaf node, a new spatial structure is created and attached to it (if the maximum global depth is not reached), and the type of that new structure is the same, depending only on their global level. Inner spatial structures are nested by using references. In our implementation we have used C++ pointers to instances that inherit from an abstract class that contains interfaces to common operations.

Inner structures, and consequently their nodes, can have fixed dimensions, or vary when their content changes. This also depends on its location in the global structure. When a hierarchical structure occupies the first position in the global structure, its spatial limits must be established explicitly, or by using the bounding box of the entire dataset. This affects quadtrees, octrees and kd-trees. However, non-hierarchical structures such as grids can cover an infinite amount of

space, and they must be created with a given size for their nodes. On the other hand, when an inner structure is below another in the global hierarchy, its spatial limits are defined by the bounding volume of the parent node. For example, a 3D grid within a leaf node of a quadtree will have fixed width and depth dimensions, the same as the size of that same quadtree node. However, the height is not bounded, and as a result can have infinite nodes in $(-\infty, \infty)$ (an example is presented in Fig. 4).

### 3.1. Insertion of point clouds

Inserting points into the nested spatial data structure is one of the most important operations in the proposed approach, because it also drives the creation or modification of the structure itself. This is also the case with deleting points, although this operation is presented later. As previously stated, a global structure scheme is composed of a series of inner structures defined by some parameters. The configuration of the global structure schema is fixed throughout the life of the dataset, and determines how data is propagated through the structure and when new inner structures instances are created at insertion time. When point clouds are inserted into the global structure, the first inner structure is created if it does not exist. The required inner nodes are created as data is propagated down to the lowest level allowed. Then, for each leaf node reached in the current inner structure, the global schema is checked for the next inner structure type to be created and linked. If it exists and data is to be propagated down, a new instance of that structure is created by using the needed parameters specified in the global schema, and then it is attached to the leaf node. This process is repeated until no more data is left to be propagated down, or the maximum global depth is reached (see Fig. 5).

During point cloud insertion, when the number of points that a node contains is greater than a given value, a subset of points is selected to be propagated down the hierarchical structure. The criteria for doing this depend on the maximum number of points that a structure level can have, as configured by the user. This value can be 0 only for spatial indexing levels, which do not store any points. In this way, a part of the point set (or the whole set) remains in the node and another part is moved to new child nodes. The partition of space in those nodes depends on how the inner structure works, so for a quadtree up to four child nodes are created, while for an octree up to eight nodes are created (empty nodes are not even created). The points are then distributed among the new nodes, based on their position in space. All this point distribution process is repeated until one of the following termination conditions is reached: (a) the maximum global depth is reached, (b) the number of points in the node does not exceed its maximum or (c) an additional user-defined function decides it. All these values are specified by the user in the global schema configuration for the dataset. More details are presented in the next section.

The subset of the original point cloud stored at each node is processed to change the global point coordinates to a local coordinate system centered at the node. This is a common technique when using spatial structures to organize point data. In this way, the 3D coordinated of the points can be encoded using floats with lower precision. Therefore it is possible to work with 32-bit floats without losing precision, while using 64-bit floats for the coordinates of the nodes, which represents a significant memory saving. Furthermore, having small numbers for the 3D coordinates of the points could also improve the performance of many data compression algorithms. In this work we use the LAZ format for storing the point clouds contained in each node.

### 3.2. Point distribution and levels of detail

There are mainly two strategies for point-in-node storage: store unique points along the hierarchical structure, or replicate some of them in intermediate nodes. Point replication is a strategy mainly aimed at having LODs for rendering purposes. In this way, there are sets

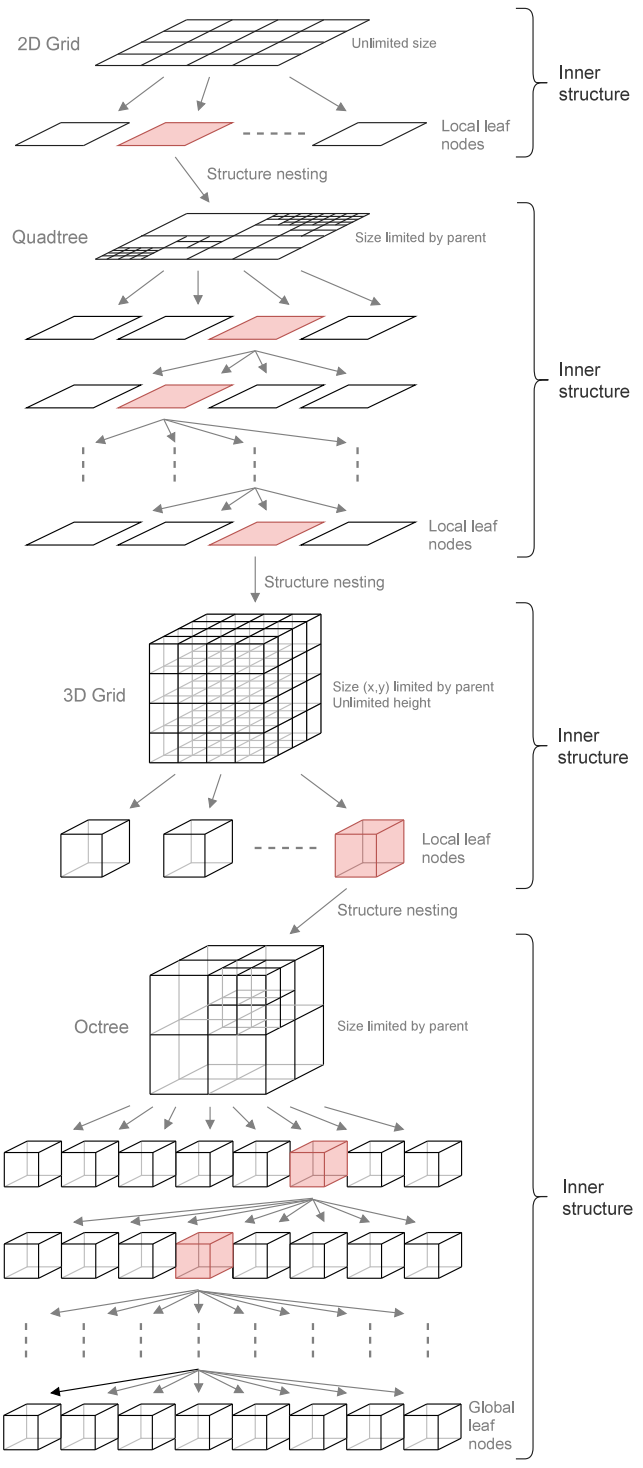


**Fig. 3.** Example of a global structure composed of a 2D grid of quadtrees of 3D grids of octrees. The 2D grid is used as the top-level structure. Octrees are used for the smallest regions.

of representative points in intermediate nodes that can be efficiently rendered (as a single buffer) without the need to load the structure to the deepest level and update the buffers in the GPU. In general this approach has very important disadvantages. First, the storage footprint increases dramatically due to data redundancy. Second, point cloud editing operations are greatly complicated by updating duplicate points. Although our framework allows both approaches, we have performed all the experiments with a non-redundant configuration,

**Table 1**
Parameters and node encoding scheme of each spatial structure implemented for the tests.

| Spatial structure | Configuration parameters | Node ID coding scheme |
|---|---|---|
| 2D grid | If top-level: nodes size (squared)<br>If not: resolution (w, d) | [x; y] (integers)<br>*Example: [153, −45]* |
| 3D grid | If top-level: nodes size (cubic)<br>If not: resolution (w, d, h) | [x; y; z] (integers)<br>*Example: [120, 5, 78]* |
| Kd-tree | 2D or 3D mode<br>Maximum depth | [node local depth {0–n} (byte);<br>[(node side {l=0, r=1} (byte); median displacement (float)); ...] ]<br>*Example: [3, 0, −34.56, 0, 10.27, 1,−0.79] (path of 3 nodes)*<br>*Example: [1, 1, 87.99] (path of 1 node)* |
| Quadtree | Maximum depth | [1 bit marking root node; [node number {0–3}; ... ] ]<br>(readed from the left, high to low)<br>*$Example_{binary}$: 1 00 01 11 (path of 3 nodes)*<br>*$Example_{binary}$: 1 10 01 (path of 2 nodes)* |
| Octree | Maximum depth | [1 bit marking root node; [node number {0–7}; ... ] ]<br>(readed from the left, high to low)<br>*$Example_{binary}$: 1 000 010 111 001 (path of 4 nodes)*<br>*$Example_{binary}$: 1 011 (path of 1 node)* |

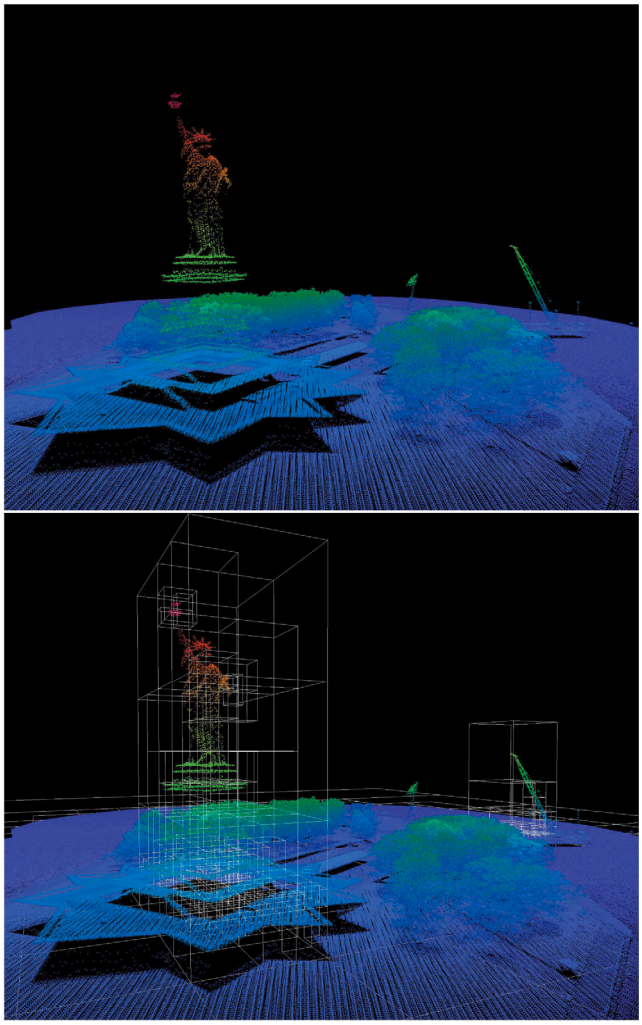

**Fig. 4.** An example of nested spatial structures. A quadtree organizes the top levels of points, mainly the landscape. A nested 3D grid allows having vertical structures without height limitations. Inside each grid voxel an octree is created for indexing data of high structures.

because storage footprint is a big concern when working with huge point clouds.

In addition to the above, the distribution of points in the hierarchical structure can be adapted for different purposes. Some levels

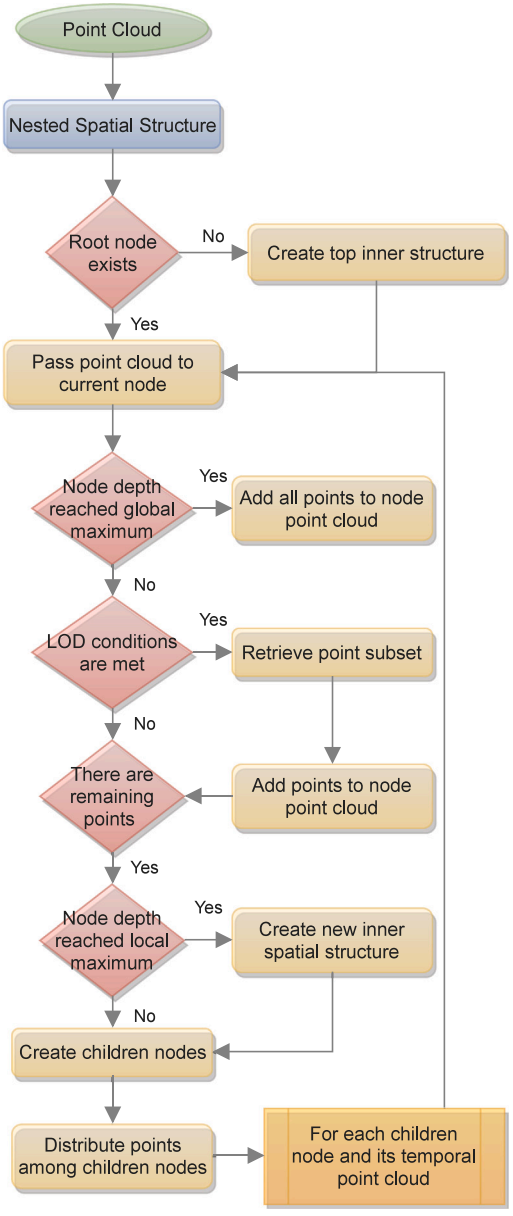


**Fig. 5.** Diagram showing the process of inserting a point cloud into the global structure.

can be used only for spatial indexing. For example, a two-level nested octree that uses the top-level one only for spatial partitioning without storing any points, and the inner one that indexes all the points. In our framework any hierarchy level can be set for indexing points or not. The strategy of distribution depends on the application. Usually, top-level structures store few or no points, while bottom-level structures store all or the majority of the points. Thus, a wise design of the global structure schema would use sparse structures only for indexing large regions at the upper levels, and dense structures for storing the data.

Finally, the method for keeping a set of points at each level in the hierarchy during the spatial subdivision is defined by the user. Typically, the deeper the level, the more points are stored in its nodes. But it does not always have to be that way. There are different strategies for the point sub-sampling step, in order to select a set of points for a given intermediate level of detail (LOD). In our framework this process is completely defined by the user. Lambda functions are the easiest way to specify the logic to be executed when each node of the structure is created. This way, we can use not only fixed values but also custom methods for determining the maximum number of points to keep at each level.

### 3.3. Data storage

In main memory the structure can be implemented in a fairly standard way, that is, by using references between nodes. However, storing data in secondary memory or transmitting it over the network is not straightforward. The main problem is the need for a unique identifier for the nodes, which will be used to create file names in a file system, key values in databases or names for network transmissions. In this regard, we have implemented two options. The first one uses a hierarchical structure that can also be used in the storage system (such as folders or directories), which intrinsically encodes the hierarchical relationship between nodes. The second option encodes the required information in the node identifier to locate it uniquely in the hierarchical structure. In this way we can use local IDs or global IDs, which include information about the relative path or full path of nodes in the structure, respectively. One or the other is used depending on the medium where the data is stored or transmitted.

Whatever the case, paths must be encoded in each node ID. Table 1 shows the coding pattern for each spatial structure implemented for the tests. Local IDs are much shorter and not redundant. As a disadvantage, to create the nodes, the information from the top nodes must be accessed with separate queries, which in database-based storage means a significant increase in the number of queries. Moreover, this also penalizes the storage in a file system. On the other hand, a global ID contains the full node path, which is a concatenation of all the local paths of the inner structures involved. Global IDs allows us to directly rebuild all the path of the global structure in memory using a single node ID. Nevertheless, at the beginning inner nodes will have no point clouds in memory until they are loaded with a separate read operation. In this case, the only real drawback is the redundancy of the paths, since all the path of the nodes of the global structure is repeated in sibling nodes (at the same depth). However, global IDs simplify the storage in both file systems and databases, especially NoSQL ones, and also support the navigation along the global structure. In addition, when data streaming is used over the network, nodes can arrive out of order, since their reconstruction is independent from the rest of the ancestor nodes. This enables more efficient implementations based on distributed, parallel, and above all, asynchronous approaches.

A binary encoding of node IDs is useful when used in memory. However, file names or database keys must follow other formats. There are multiple solutions for converting a buffer of binary data to plain text. In this work we have used a base32 encoding for IDs, which is compatible with case-insensitive file systems. IDs can also be encoded as human readable text for debugging purposes. In addition to this, IDs can also be compressed with Huffman or arithmetic coding, both in binary and text format. This could alleviate the limitations of some file systems regarding the maximum length of file names, or the length of database keys, especially in the case of using global IDs.

For the storage of the dataset, both in file systems or databases, LAZ files are used for each point cloud associated with each node of the hierarchical structure. The structure itself is stored using special files or their equivalent in a database table. There is a file or master record that specifies the global parameters of the dataset, and it is the first one that must be read in order to start working with the data. This master record includes the global schema of the data structure, the global bounding box of the entire dataset, the inventory of elements (total number of points, polygons, and other elements that will be added in the future), as well as the location of the root node file or database record. Nodes are stored as separate files or records that contain information about the associated LAZ files and the IDs of the child nodes.

### 3.4. Navigation and spatial queries

Once the dataset is stored in the global data structure, all the necessary editing, visualization, analysis, etc. operations can be performed from the application. First, the application must access the master record that contains the global dataset information. With the global scheme of the structure, nodes can be loaded from secondary storage. With large datasets, progressive and partial loading is common practice, due to typical memory capacity and performance limitations. Part of the hierarchical structure is therefore asynchronously mapped or replicated in memory through a dedicated thread, and the application can request more nodes o dispose them from memory as needed.

Apart from traversing the structure for basic operations such as visualization, it is also necessary to support additional operations such as queries based on rectangles or volumes, 2D projections, calculation of the nearest neighbors (KNN) or ray tracing. Region-based queries are performed on a first pass through the structure of nodes, as their bounding box can be calculated from the global schema definition in a straightforward manner. In this way, the total or partial inclusion nodes in a rectangle or cut volume can be easily calculated. In a second pass, all point clouds from the selected nodes can be loaded. Logically, this operation is also limited to a maximum number of points when working with LODs.

Ray tracing operations must be adapted to the nested global structure. The main difficulty is to make compatible the different traversing methods of each basic spatial structure (grid, quadtrees, octrees, etc.). The variety of available ray-tracing algorithms applicable to each spatial structure is beyond the scope of this paper. In general the adaptation of any of these methods requires recomputing the origin of the ray along its path at the intersecting point with the boundary of each inner structure traversed. This enables the location of the correct node in the global structure. Additionally, 3D rays must be projected into the XY plane in order to correctly traverse planar structures such as 2D grids or quadtrees.

### 3.5. Dataset editing operations

The global structure can be iterated to perform global operations, such as a histogram or a filtering. In our system, the minimum granularity to operate with points is the entire point cloud associated with each node. Therefore, a cloud must be fully loaded into the application to make changes to any of its points. Once the operation is finished, it is compressed again and saved in the storage system.

One of the most interesting operations is point deletion. It can be done in two ways: (a) by labeling the point as deleted using an attribute of the point, such as the attribute *withheld* of the LAS format (Isenburg, 2013) or the attribute *user data*, or (b) effectively deleting the point from the cloud point. In this case, deleted points can be discarded completely or stored in a separate file so that they can be recovered at any time (for example, by an undo operation).

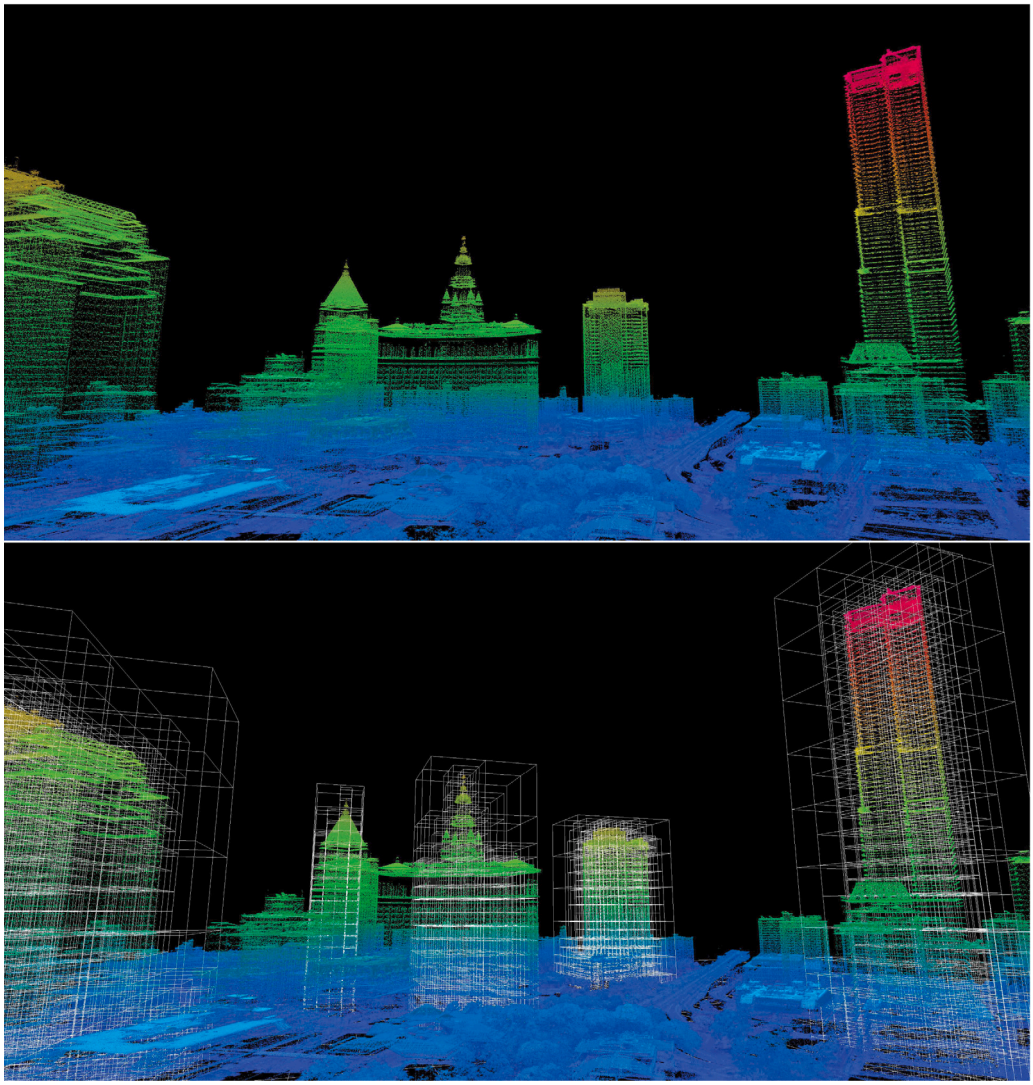

**Fig. 6.** An example of adaptive nested structure tailored for urban datasets. For the sake of clarity, only the inner structures for the tallest buildings are shown.

This option has the advantage of improving loading operations and network transfers. It must be noted that some spatial structures will become unbalanced when points are removed. This is the case of kd-trees, that are even sensitive to insertion operations. In this case, a reconfiguration of the sub-structure from the affected node must be carried out. However, sub-structure reconfiguration is more an optional optimization than an mandatory operation. Keeping empty nodes has a memory cost, but it could also save processing time during editing. This is a compromise between memory optimization and overall performance. Logically, deleting or modifying nodes of the structure implies the modification of the corresponding files in secondary memory. Our system performs this step only after all modifications to the structure have been made, or when it is forced by the user.

Related to the above, the framework also supports the use of layers that allows separating sets of points according to criteria determined by the user, such as data captured with different devices or in different campaigns. This system is complementary to the attribute of point classification that can be found in the LAS specification, and consists of storing separated clouds for each layer in each node. So far, and for the sake of simplicity, we described the meta-structure as having a single point cloud per node, but actually our implementation allows more than one cloud per node to support the layer system. In addition, as future work, this functionality will allow saving a history of point cloud modifications in each node, which could be useful for implementing the undo operation or supporting collaborative work.

### 3.6. Design guidelines for nested spatial structures

Deciding the optimal combination of spatial structures for a given dataset is a complex problem in itself. The proposed framework provides great versatility for the definition of nested structures, but the actual choice of the specific structures to be used must be decided by the user. In the experiments carried out we propose a specific schema for a nested spatial structure based on our experience that is well suited to datasets that present large extensions of territory with areas of dense and complex data (e.g., buildings). In future work, some pre-made schemas would be available for different types of datasets, such as urban planning, civil engineering, surveying, forestry, BIM, archaeological sites, etc., which have different spatial distribution of the points and editing requirements. In this section we present some design guidelines for optimizing the design of a global schema. It must be noted that the framework can mimic other standard solutions, such as nested octrees (Wimmer and Scheiblauer, 2006), with the advantage of being able to choose the spatial indexing configuration for each dataset separately, all with the same system.

In general, datasets exclusively derived from ALS tend to be efficiently indexed with planar structures, while TLS scans benefit more from 3D structures. When both types of data coexist in the same dataset is when our meta-structure is most useful. In general, it is necessary to ensure that the structures used adapt well to the spatial distribution of

**Table 2**
Datasets used in the experimentation.

| Datset | Year | WGS84 Zone center | Width | Height | Owner | Format |
|---|---|---|---|---|---|---|
| Madrid (Instituto Geográfico Nacional, 2021) | 2016 | −3.684214E, 40.473152N | 6 km | 3 km | PNOA | LAZ |
| New York (City of New York, 2017) | 2017 | −74.002447E, 40.708918N | 4.42 km | 2.27 km | NYC OpenData | LAZ |
| Liberty Island, for figure 4 (City of New York, 2017) | 2017 | −74.044761E, 40.689803N | 1.07 km | 1.05 km | NYC OpenData | LAZ |

the points. For example, if the vertical structures are mostly buildings, it will be necessary to use 3D structures that allow adapting to volumes of the typical dimensions of buildings. In this case, a reasonable option would be a quadtree of 3D grids of octrees (see Fig. 6). This approach is also suitable for other structures such as bridges, dams, tunnels, caves, underground facilities, etc.

As previously stated, kd-trees and r-trees are not the best options when updating operations are required on point clouds because the hierarchical structure becomes unbalanced. This does not mean that they cannot be used, but they should be used with care with datasets enabled for intensive editing. These structures should only be used as inner leaf structures, that is, there should not be further inner structures below them on the hierarchy. Each type of spatial data structure has its strengths and weaknesses. In general, sparse structures such as grids should only be used for indexing large regions at the upper levels, while hierarchical structures should be used in the lower levels for storing large amounts of dense data, such as the octree. Another use for sparse structures is covering an infinite space. The examples in Figs. 4 and 6 show how nested 3D grids within leaf nodes of quadtrees allow the coverage of infinite vertical space to efficiently index buildings and their content. In addition, as a first-level structure, a grid allows the dataset to be expanded with new data in new areas without having to rebuild the entire structure. This is not possible with hierarchical structures such as the quadtree and the octree, which must be created knowing in advance the dimensions of the space that the dataset takes, and which cannot then be varied without altering the indexing structure.

Another important decision is the value adjustment of the properties of each structure used in the global scheme, especially the maximum depth and the maximum number of points for each level. The maximum depth is directly related to the size of the smallest possible leaf nodes. On the other hand, as described in Section 3.2, in our framework any hierarchy level can be set for storing points or not. It is recommended for larger datasets to use few or no points in the first levels, so that these serve only as spatial indices to speed up queries and discard large areas when navigating. Mid-levels should store a moderate number of points, allowing fast rendering under conditions of low performance, bandwidth, or memory capacity. The lower levels will therefore contain the bulk of the dataset. In general, a linear or exponential increase in the number of points with respect to depth is the most common choice. However, this can be tailored for specific purposes. For example, the meta-structure used in the example in Fig. 6 only unfolds the structure to the lowest level (octrees) when the height of the points contained in the leaf nodes of the quadtree of the upper level exceeds a certain height. This is used to spatially index only tall buildings and structures. The rest is indexed in the plane with the base quadtree. All these steps can be further customized with optional user-defined functions that specify the logic to be executed when each node in the structure is created.

## 4. Results and discussion

In this section we present a comparison of several spatial data structures built with our framework. Two of them, the quadtree and the octree, are well-known basic structures (see Fig. 2), and have been selected to highlight the differences in terms of performance and storage footprint compared to standard solutions. The other structure follows the same schema presented in the example of Fig. 3, a 2D grid of quadtrees of 3D grids of octrees, which is well suited for urban scenes. The 2D grid indexes tiles of points at a landscape level. Quadtrees are used to map the content of each tile of terrain. 3D grids and their inner octrees are nested into the leaf nodes of the quadtrees for mapping vertical structures, mainly buildings. In any case, our system allows any other combination of nested structures. The experimentation was carried out on a PC with Intel Core i7-8700 3.3 GHz, 32 GB RAM, GTX 1060 GPU with 6 GB RAM. The proposed system is implemented in C++17, and uses OpenGL for rendering.

Two urban datasets, with different point densities, have been tested (City of New York, 2017; Instituto Geográfico Nacional, 2021). We have used urban environments as they present a perfect situation for the union of LiDAR data from different capture technologies, which is a greater challenge for spatial indexing. This means working with different point densities and spatial distributions. The datasets features are presented in Table 2. For each one, an additional augmented dataset has been used. These variants have several buildings populated with random points that simulate a TLS scanning result. This simulation is done because we have not found public real-based data suitable for this benchmark. In any case, this serves our purpose of populating the datasets with highly concentrated points in some areas.

Table 3 shows the benchmarks results for the creation of each structure, including reading the LAS/LAZ source files, building the structure and storing all data in secondary memory (a file system on an SSD drive). The depth of the data structures used has been adjusted so that the dimensions of leaf nodes in the corresponding leaf inner structure are in the range of a few meters. The dimensions of the leaf nodes are the same for all the structures in order to see how they behave in terms of the number of necessary nodes and the number of points that can be organized in a single node. The goal is to get as few nodes as possible while distributing the points as evenly as possible. As can be seen in Table 3, the quadtree is better suited to datasets without augmentation, since the information is mostly planimetric. In this case, the octree is not much of a benefit. However, when there is simulated TLS data, the quadtree is very inefficient, as there is a large amount of vertical data converging on the same leaf nodes, increasing the maximum number of points to several millions in the augmented Manhattan dataset. This results in a poor performance in all display and management operations on those leaf nodes. Octree performs much better in these cases due to its volumetric nature, and is better suited to the morphology of the buildings. However, the meta-structure is the one that best adapts to these datasets, managing to minimize the number of nodes and at the same time keeping the number of average points within good limits for the editing algorithms (neither too low nor too high).

The total net storage footprint of all tested variants is similar. This value refers to secondary memory usage, and does not include the actual space used by the data in the database or file system due to the occupancy of minimum allocation units. This means that the meta-structure, despite being conceptually more complicated, does not result in an increase in storage, which we could consider as a positive point. Furthermore, since the number of nodes is smaller, the effective storage footprint will be also smaller than the rest of the proposals, especially in file systems where the predefined block size penalizes a high number of small files. The performance of the creation step is also very similar in all the tests carried out, although the meta-structure is somewhat faster in all cases. This is due to the smaller number of nodes to be created. Regarding the amount of main memory needed to perform operations with points (create, read, write and delete), it greatly depends on the configuration of the top-level application using the framework. The

**Table 3**
Test results. Maximum points per node are calculated with an exponential function based on depth.

| | | Madrid | Madrid (aug[a]) | Manhattan | Manhattan (aug[b]) |
|---|---|---|---|---|---|
| | Points | 48,255,227 | 443,055,227 | 325,434,652 | 1,141,434,652 |
| | Average density (per square meter) | 2.68 | 24.61 | 32.58 | 114.28 |
| Quadtree | Maximum depth | 10 | 10 | 10 | 10 |
| | Net storage footprint (compressed) | 352 MB | 2.13 GB | 2.15 GB | 6.04 GB |
| | Total nodes | 38,877 | 40,326 | 154,019 | 154,719 |
| | Leaf nodes | 29,094 | 30,136 | 115,340 | 115,826 |
| | Points in leaf nodes (max; avg) | 4358; 986 | 589,467; 13,989 | 8088; 1495 | 3,370,942; 8526 |
| | Building time | 114.76 s | 693.61 s | 632.49 s | 1927.05 s |
| | Building throughput (points/s) | 420,488 | 638,767 | 514,529 | 592,322 |
| Octree | Maximum depth | 10 | 10 | 10 | 10 |
| | Net storage footprint (compressed) | 352 MB | 1.95 GB | 2.11 GB | 5.55 GB |
| | Total nodes | 48,363 | 131,704 | 197,864 | 228,241 |
| | Leaf nodes | 39,347 | 110,897 | 161,528 | 187,232 |
| | Points in leaf nodes (max; avg) | 3676; 787 | 11,701; 3346 | 8088; 1352 | 110,164; 5411 |
| | Building time | 122.1 s | 793.02 s | 668.79 s | 2207.34 s |
| | Building throughput (points/s) | 395,210 | 558,693 | 486,602 | 517,108 |
| Multi[c] | Maximum depth | 10 (1+5+1+3) | 10 (1+5+1+3) | 10 (1+5+1+3) | 10 (1+5+1+3) |
| | Net storage footprint (compressed) | 338 MB | 1.99 GB | 2.15 GB | 5.6 GB |
| | Total nodes | 3904 | 16,493 | 89,443 | 120,450 |
| | Leaf nodes | 2876 | 13,127 | 69,270 | 94,817 |
| | Points in leaf nodes (max; avg) | 47,794; 16,670 | 96,144; 33,015 | 97,592; 4323 | 108,485; 11,529 |
| | Building time | 76.12 s | 655.09 s | 505.64 s | 1826.09 s |
| | Building throughput (points/s) | 633,936 | 676,327 | 643,609 | 625,070 |

[a]Dataset augmented using 5 simulated TLS buildings.
[b]Dataset augmented using 10 simulated TLS buildings.
[c]Multi-structure: 2D grid + quadtree + 3D grid + octree.

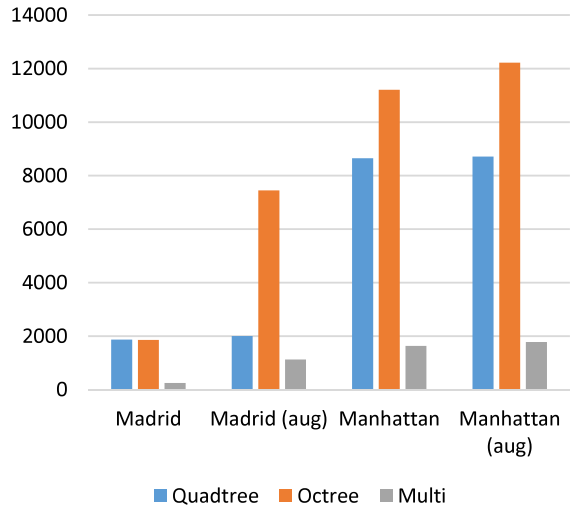


**Fig. 7.** Reading test results: average number of nodes loaded for the same query volumes (3D).

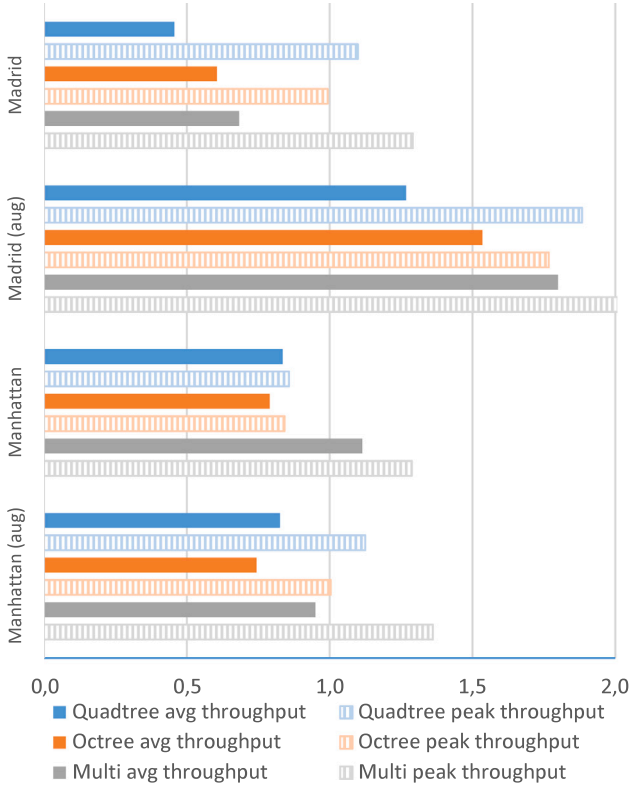


**Fig. 8.** Reading test results: throughput in millions of points per second for the same query volumes.

framework itself only keeps in memory the data that is needed at any given time. The memory policy is highly adaptable, so a limit can be specified. Least used nodes will be switched to secondary memory if needed, no matter the global scheme used. We have not included a comparison of this because this matter does not suppose a remarkable difference between datasets or schemas.

Figs. 7 and 8 show the performance of spatial query operations based on 100 random volumes defined by boxes with sizes between 20% and 40% of the dataset in each dimension (x, y, z). Both the quadtree and the octree require more nodes to be read to complete the queries than the meta-structure (Fig. 7). This is due to the fact that the meta-structure better adapts to the distribution of the data, as mentioned before, which means that fewer nodes are generated during creation (Table 3). The number of nodes required in spatial queries is especially relevant with network communications, where the overhead of multiple accesses and requests can be a significant performance issue. Fig. 7 also presents differences between datasets. This is due to the difference in the number of nodes, which increases in relation to the extension of the dataset and the maximum depth of the structure (in the experiments this has remained fixed). For this reason there is almost no difference in the number of nodes, and therefore in the number of nodes accessed in read operations, between the normal datasets and

their augmented versions. The only discordant value in this sense is the one corresponding to the quadtree in the case of Madrid (augmented), which should be higher, as happens to the other structures. This is because augmenting this dataset using vertical structures hardly affects a 2D spatial structure. In the case of Manhattan, this difference is much less appreciable, because the starting dataset already had enough complexity in the spatial distribution of the points, and therefore adding virtual buildings only increases the concentration of points in certain areas.

Fig. 8 shows the performance results in millions of points per second. This throughput includes the entire process of opening the dataset, loading the nodes into memory, accessing the indexed point clouds, decompressing them, loading them into memory and selecting the set of points that falls strictly within the query volume. Point clouds are compressed in LAZ format. Both average and maximum (peak) values are included. As can be seen, there are differences in performance, both between data structures and between datasets, although they are not very significant. There are several factors that affect performance. One already explained above is the adjustment of the structure to the spatial distribution of the data, which affects the number of nodes accessed. For the majority of situations, the multi-structure performs better than the other structures in this aspect. Another factor is the relative simplicity of the spatial structure, which causes nodes to index large areas. This is the case of the quadtree, which stores a large number of points in each node, which increases the peak throughput value with some queries. When datasets are complex, particularly with highly dense areas such as buildings, simpler schemas tend to be less efficient. Every structure usually has a very good peak throughput, which is much higher than the average value. However the peak value is only reached on very few occasions, when the query area coincides with very few nodes that contains a large number of points. In general, the average throughput is the measure that better reflects the performance of the spatial structure under real conditions.

The augmented version of the Madrid dataset shows different performance compared to the basic dataset. It seems that, having a shallow spatial structure, the queries return few nodes but have a large number of points, so the throughput increases remarkably. The exception to this argument is the octree, which has a larger number of nodes (see Fig. 7). However, those nodes tend to be nearly empty in most cases due to their poor spatial fit to the dataset topology, and thus do not affect the same throughput gain that the other structures have.

In general, the multi-structure proposed in the experiments performs better than the other basic structures, especially with more complex data sets. This should be considered as a positive result, a priori lower performance could be expected due to the use of a more complicated indexing structure. It seems that, in general, the highest impact on performance falls on the loading and decompression of the point clouds. We can conclude that the proposed framework can optimally solve the spatial indexing of LiDAR datasets, maintaining or even improving performance, while offering complete freedom for defining the spatial structure. Furthermore, it can also implement standard non-nested schemes for spatial organization such as quadtree, octrees etc., without incurring a performance penalty.

## 5. Conclusions and future work

In this paper we have presented a new approach for defining an ad-hoc nested spatial data structure for indexing LiDAR data. Several basic spatial data structures can be combined and nested in order to adapt the indexing of the data to its spatial distribution, ranging from full ALS datasets to high-precision small-scale TLS datasets, or the mixing of both. We have also presented some guidelines for choosing the correct combination of data structures, depending on the nature of the data and the operation to be performed.

As future work, there are some improvements that can be made to our approach. Maybe the most relevant is the possibility of having different structure schemas at different zones of a dataset instead of a global one, that is, a dynamic meta-structure. This could be interesting for working with datasets corresponding to continental or planet-size areas, where there could be different usage goals between areas. However, the potential benefits of this variant should be weighed against the current method.

It would be also desirable to have a heuristic for automatically setting the optimal configuration of the global structure schema for a given dataset. The automatic algorithm should provide a solution in a reasonable time that maximizes the use of the space occupied by the nodes of the indexing structures, and taking into account a series of objectives to optimize, such as memory used globally, the maximum number of points per node, the maximum global depth, etc.

Finally, this framework can be used with other geometric entities, such as polygonal meshes. This would have significant benefits for tasks such as data analysis or visualization. Also, it would be interesting to explore new options for automatically obtaining triangulations from the point clouds stored, at different levels of details. There are several proposals in previous work for the triangulation of massive point clouds that could be adapted to our system. Of course, the most interesting option would be the coexistence of point clouds and triangle meshes in the same structure.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## References

Baert, J., Lagae, A., Dutré, P., 2014. Out-of-core construction of sparse voxel octrees: Out-of-core construction of sparse voxel octrees. Comput. Graph. Forum 33 (6), 220–227.

Boehler, W., M., B.V., Marbs, A., 2018. OGC Testbed-14: Point Cloud Data Handling Engineering Report. Technical Report, i3mainz, Institute for Spatial Information and Surveying Technology, FH Mainz, Holzstrasse 36, 55116 Mainz, Germany, URL http://www.opengis.net/doc/PER/t14-D013.

Boehm, J., 2014. File-centric organization of large LiDAR point clouds in a big data context. In: Workshop on Processing Large Geospatial Data. Cardiff, UK.

Boehm, J., Liu, K., Alis, C., 2016. Sideloading ingestion of large point clouds into the Apache spark bid data engine. ISPRS - Int. Arch. Photogram. Remote Sens. Spatial Inform. Sci. XLI-B2, 343–348.

Bräunl, T., 2020. Lidar sensors. In: Robot Adventures in Python and C. Springer International Publishing, Cham, pp. 47–51.

Bunting, P., Armston, J., Lucas, R.M., Clewley, D., 2013. Sorted pulse data (SPD) library. part I: A generic file format for LiDAR data from pulsed laser systems in terrestrial environments. Comput. Geosci. 56, 197–206.

Cao, C., Preda, M., Zaharia, T., 2019. 3D point cloud compression: A survey. In: The 24th International Conference on 3D Web Technology. ACM, LA CA USA, pp. 1–9.

City of New York, 2017. Topobathymetric LiDAR data. URL https://data.cityofnewyork.us/City-Government/Topobathymetric-LiDAR-Data-2017-/7sc8-jtbz.

Deibe, D., Amor, M., Doallo, R., 2018. Big data storage technologies: a case study for web-based LiDAR visualization. In: 2018 IEEE International Conference on Big Data (Big Data). IEEE, Seattle, WA, USA, pp. 3831–3840.

Deibe, D., Amor, M., Doallo, R., 2019. Supporting multi-resolution out-of-core rendering of massive LiDAR point clouds through non-redundant data structures. Int. J. Geogr. Inf. Sci. 33 (3), 593–617.

Deng, X., Liu, P., Liu, X., Wang, R., Zhang, Y., He, J., Yao, Y., 2019. Geospatial big data: New paradigm of remote sensing applications. IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens. 12 (10), 3841–3851.

Elseberg, J., Borrmann, D., Nüchter, A., 2013. One billion points in the cloud – an octree for efficient processing of 3D laser scans. ISPRS J. Photogramm. Remote Sens. 76, 76–88.

Evans, M.R., Oliver, D., Zhou, X., Shekhar, S., 2014. Spatial big data. case studies on volume, velocity, and varitety. In: Big Data: Techniques and Technologies in Geoinformatics. CRC Press.

Gong, J., Zhu, Q., Zhong, R., Zhang, Y., Xie, X., 2012. An efficient point cloud management method based on a 3D R-tree. Photogramm. Eng. Remote Sens. 78, 373–381.

Goswami, P., Erol, F., Mukhi, R., Pajarola, R., Gobbetti, E., 2013. An efficient multi-resolution framework for high quality interactive rendering of massive point clouds using multi-way kd-trees. Vis. Comput. 29 (1), 69–83.

Hongchao, M., Wang, Z., 2011. Distributed data organization and parallel data retrieval methods for huge laser scanner point clouds. Comput. Geosci. 37 (2), 193–201.
Huang, L., Wang, S., Wong, K., Liu, J., Urtasun, R., 2020. OctSqueeze: Octree-structured entropy model for LiDAR compression. ArXiv:2005.07178 [Cs, Eess].
Instituto Geográfico Nacional, 2021. PNOA-LiDAR. URL https://pnoa.ign.es/el-proyecto-pnoa-lidar.
Isenburg, M., 2013. LASzip. Photogramm. Eng. Remote Sens. 79 (2), 209–217.
Józsa, O., Börcs, A., Benedek, C., 2013. Towards 4D virtual city reconstruction from lidar point cloud sequences. ISPRS Ann. Photogram. Remote Sens. Spatial Inform. Sci. II-3/W1, 15–20.
Kämpe, V., Sintorn, E., Assarsson, U., 2013. High resolution sparse voxel DAGs. ACM Trans. Graph. 32 (4), 1–13.
Krämer, M., Senner, I., 2015. A modular software architecture for processing of big geospatial data in the cloud. Comput. Graph. 49, 69–81.
Lee, J.-G., Kang, M., 2015. Geospatial big data: Challenges and opportunities. Big Data Res. 2 (2), 74–81.
Liang, X., Kankare, V., Hyyppä, J., Wang, Y., Kukko, A., Haggrén, H., Yu, X., Kaartinen, H., Jaakkola, A., Guan, F., Holopainen, M., Vastaranta, M., 2016. Terrestrial laser scanning in forest inventories. ISPRS J. Photogramm. Remote Sens. 115, 63–77, Theme issue 'State-of-the-art in photogrammetry, remote sensing and spatial information science'.
Lin, A., Wu, H., Liang, G., Cardenas-Tristan, A., Wu, X., Zhao, C., Li, D., 2020. A big data-driven dynamic estimation model of relief supplies demand in urban flood disaster. Int. J. Disas. Risk Reduct. 49, 101682.
Lu, B., Wang, Q., Li, A., 2019. Massive point cloud space management method based on octree-like encoding. Arab. J. Sci. Eng. 44 (11), 9397–9411.
Pääkkönen, P., Pakkala, D., 2015. Reference architecture and classification of technologies, products and services for big data systems. Big Data Res. 2 (4), 166–186.
Poux, F., 2019. The Smart Point Cloud: Structuring 3D intelligent point data. (Ph.D. thesis). Université de Liège, Liège, Belgique.
Poux, F., Billen, R., 2019. Voxel-based 3D point cloud semantic segmentation: Unsupervised geometric and relationship featuring vs deep learning methods. ISPRS Int. J. Geo-Inf. 8 (5), 213.
Preiner, R., Jeschke, S., Wimmer, M., 2012. Auto splats: Dynamic point cloud visualization on the GPU. In: Eurographics Symposium on Parallel Graphics and Visualization. p. 10.
Richter, R., Discher, S., Döllner, J., 2015. Out-of-core visualization of classified 3D point clouds. In: Breunig, M., Al-Doori, M., Butwilowski, E., Kuper, P.V., Benner, J., Haefele, K.H. (Eds.), 3D Geoinformation Science. In: Lecture Notes in Geoinformation and Cartography, Springer, Cham, pp. 227–242.
Šašak, J., Gallay, M., Kaňuk, J., Hofierka, J., Minár, J., 2019. Combined use of terrestrial laser scanning and UAV photogrammetry in mapping alpine Terrain. Remote Sens. 11 (18).
Scheiblauer, C., Wimmer, M., 2011. Out-of-core selection and editing of huge point clouds. Comput. Graph. 35 (2), 342–351.
Schön, B., Mosa, A.S.M., Laefer, D.F., Bertolotto, M., 2013. Octree-based indexing for 3D pointclouds within an oracle spatial DBMS. Comput. Geosci. 51, 430–438.
Schuetz, M., 2016. Potree: Rendering Large Point Clouds in Web Browsers. Ph.D. thesis. Vienna.
Schutz, M., Krosl, K., Wimmer, M., 2019. Real-time continuous level of detail rendering of point clouds. In: IEEE VR 2019. IEEE, Osaka, Japan, pp. 103–110.
Schütz, M., Ohrhallinger, S., Wimmer, M., 2020. Fast out-of-core octree generation for massive point clouds. Comput. Graph. Forum 39 (7), 155–167.
Ströter, D., Mueller-Roemer, J.S., Stork, A., Fellner, D.W., 2020. OLBVH: octree linear bounding volume hierarchy for volumetric meshes. Vis. Comput. 36 (10–12), 2327–2340.
Sugimoto, K., Cohen, R.A., Tian, D., Vetro, A., 2017. Trends in efficient representation of 3D point clouds. In: 2017 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC). IEEE, Kuala Lumpur, pp. 364–369.
Tian, S., Li, X., Zeng, J., Wei, Z., 2019. The organization of point cloud data based on the compact octree model. J. Phys. Conf. Ser. 1302, 022047, Publisher: IOP Publishing.
Ullrich, A., Pfennigbauer, M., 2019. Advances in lidar point cloud processing. In: Turner, M.D., Kamerman, G.W. (Eds.), Laser Radar Technology and Applications XXIV. SPIE, Baltimore (USA), p. 19.
Wang, Y., Lv, H., Ma, Y., 2020. Geological tetrahedral model-oriented hybrid spatial indexing structure based on octree and 3D R*-tree. Arab. J. Geosci. 13 (15), 728.
Wimmer, M., Scheiblauer, C., 2006. Instant points: Fast rendering of unprocessed point clouds. In: Symp. on Point-Based Graphics.
Yang, J., Huang, X., 2014. A hybrid spatial index for massive point cloud data management and visualization: Massive point cloud management and visualization. Trans. GIS 18, 97–108.