Computers & Graphics 130 (2025) 104254

Contents lists available at ScienceDirect

Computers & Graphics

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

Special Section on CGVC2024

Real-time indexing and visualization of LiDAR point clouds with arbitrary attributes using the M^{3}NO data structure

Paul Hermann, Michel Krämer *, Tobias Dorra, Arjan Kuijper

Fraunhofer Institute for Computer Graphics Research IGD, Fraunhoferstr. 5, 64283, Darmstadt, Germany
Technical University of Darmstadt, Karolinenplatz 5, 64277, Darmstadt, Germany

ARTICLE INFO

Keywords:
Point clouds
LiDAR
Data structures
Data processing
Real-time indexing
Spatial indexing
Visualization
Geospatial data

ABSTRACT

In previous work, we have presented an approach to index 3D LiDAR point clouds in real time, i.e. while they are being recorded. We have further introduced a novel data structure called M^{3}NO, which allows arbitrary attributes to be indexed directly during data acquisition. Based on this, we now present an integrated approach that supports not only real-time indexing but also visualization with attribute filtering. We specifically focus on large datasets from airborne and land-based mobile mapping systems. Compared to traditional indexing approaches running offline, the M^{3}NO is created incrementally. This enables dynamic queries based on spatial extent and value ranges of arbitrary attributes. The points in the data structure are assigned to levels of detail (LOD), which can be used to create interactive visualizations. This is in contrast to other approaches, which focus on either spatial or attribute indexing, only support a limited set of attributes, or do not support real-time visualization. Using several publicly available large data sets, we evaluate the approach, assess quality and query performance, and compare it with existing state-of-the-art indexing solutions. The results show that our data structure is able to index 5.24 million points per second. This is more than most commercially available laser scanners can record and proves that low-latency visualization during the capturing process is possible.

1. Introduction

Geospatial point clouds are becoming increasingly important in many areas. For example, architects and urban planners use them in building construction projects [1] or for urban infrastructure planning [2]. Other applications include the rollout of fiber optic lines [3], district heating planning [4], or flood modeling [5]. On a larger scale, point clouds are used to generate digital elevation models [6] and for the mapping and regular monitoring of forests [7] or railway lines [8].

Point clouds are captured using airborne or land-based mobile mapping systems equipped with LiDAR (Light Detection And Ranging) technology [9,10]. LiDAR scanners are becoming more and more precise and record millions of points per second. These points contain not only spatial coordinates but a variety of attributes such as color, GPS time, and the intensity of the reflected laser beam. More attributes such as the 3D normal for each point or its semantic classification can be derived from the recorded information and are often also included in the final result. The LAS specification [11], a widely used point cloud file format, defines a range of possible attributes. Fig. 1 shows screenshots of a few selected ones.

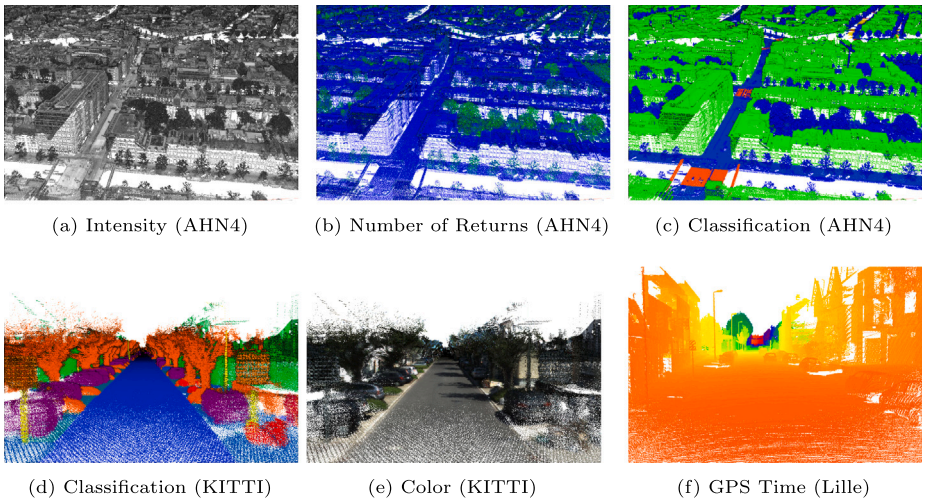
LiDAR-recorded geospatial point clouds are usually very large, both in terms of data volume and area. It is not uncommon that datasets reach sizes of many terabytes. Some of them even cover whole countries, e.g. the AHN4 dataset of the Netherlands [12]. Analyzing and visualizing such large datasets requires sophisticated out-of-core acceleration structures. While spatial indexes enable fast access to specific areas of the point cloud in different levels of detail, attribute indexes allow the points to be quickly filtered based on given criteria or value ranges. A combined index, for example, would be able to return all points that lie within a given spatial extent (or bounding box), have been recorded in a certain period of time, and belong to a specified class (e.g. *Tree* or *Building*).

The process of creating such an index is called *indexing*. The current common practice is to index point clouds offline, i.e. some time after they have been recorded with tools such as *Schwarzwald*, *PotreeConverter*, or *Entwine* [13–15]. This out-of-core process often takes several hours and doing it later unnecessarily extends the time it takes from recording the data until it can actually be utilized.

Many applications, however, could benefit from a more immediate access. For example, if regular processes such as railway monitoring for

* Corresponding author at: Fraunhofer Institute for Computer Graphics Research IGD, Fraunhoferstr. 5, 64283, Darmstadt, Germany.
E-mail addresses: paul.hermann@igd.fraunhofer.de (P. Hermann), michel.kraemer@igd.fraunhofer.de (M. Krämer), tobias.dorra@igd.fraunhofer.de (T. Dorra), arjan.kuijper@tu-darmstadt.de (A. Kuijper).

https://doi.org/10.1016/j.cag.2025.104254
Received 17 January 2025; Received in revised form 20 April 2025; Accepted 12 May 2025
Available online 28 May 2025



(a) Intensity (AHN4) (b) Number of Returns (AHN4) (c) Classification (AHN4)

(d) Classification (KITTI) (e) Color (KITTI) (f) GPS Time (Lille)

Fig. 1. Different point attributes of the datasets used in this paper colored by value.

predictive maintenance could make use of the data right after it has been acquired, they would be able to react much quicker to imminent faults. The same applies to emergency cases (e.g. earthquakes or floods) where immediate availability of analysis results is key.

Going one step further, analyzing and visualizing a point cloud already while it is being recorded would enable real-time quality assurance. Errors in the recording or missing spots could immediately be displayed to the operator (e.g. on a tablet computer) who could then directly fix them. This would avoid having to repeat the data acquisition process later, which is time-consuming, cost-intensive and, again, extends the time until the data can be put into use.

Processing LiDAR point cloud data in real time is, however, rather challenging as modern laser scanners record millions of points per second. A real-time indexing algorithm has to be able to process points at least as fast as the laser scanner produces them. In addition, querying the index during acquisition (e.g. for live quality assurance) requires it to be responsive and to return results quickly.

In previous work, we have already presented an approach to index 3D LiDAR point clouds in real time, i.e. while they are being recorded [16]. We have further introduced a novel data structure called M^{3}NO, which allows arbitrary attributes to be indexed directly during data acquisition [17]. Based on this, we now present an integrated approach that supports not only real-time indexing but also interactive visualization with attribute filtering. It allows the point cloud to be spatially queried based on a bounding box or a view frustum, and to be filtered by ranges of attribute values. At the same time, the index contains a level-of-detail structure and can therefore be used for interactive visualization. Spatial and attribute filters determine which parts of the point cloud should be returned at the desired level of detail.

For reproducibility, our implementation called *Lidarserv* is available under an open-source license on GitHub [18].

1.1. Goals

For the design and evaluation of our integrated approach, we define the following research goals:

G1 – Real-time indexing. Our indexing algorithm should be able to insert points into the M^{3}NO structure at least as fast as they are being recorded by a typical commercially available laser scanner.

G2 – Query data reduction. The amount of data that needs to be loaded into memory during querying relative to the size of the result set is a quality metric for every index. Our goal is to eliminate as many octree nodes as possible early in the query process to reduce the number of points that need to be loaded from disk.

G3 – Query time reduction. Compared to naive sequential filtering without an index, our data structure should substantially decrease the query execution time. Furthermore, the query times should be comparable or better than those of existing approaches.

G4 – Real-time visualization. Our data structure should allow users to interactively explore the data while it is being recorded and to perform filtering queries that update the scene in real time. The time that passes between the recording of a point until it is displayed on screen (latency) should be minimized.

1.2. Contributions

As mentioned above, we put together the building blocks from our previous works [16,17] into an integrated system. Based on the results of the extended evaluation of the indexing performance and visualization latency (see Section 7), we are now able to show that *real-time visualization of 3D point clouds and filtering based on the values of arbitrary attributes* is indeed possible. Newly arriving points only need 31 ms until they are indexed and displayed.

Furthermore, we have extended the M^{3}NO data structure and *developed a new index* for cases where attribute values are not uniformly distributed across the value range. The *Bin List Index* is a compact data structure that automatically adapts to the data being recorded (see Section 3.2.2). It helps reduce the number of false positives and the amount of data that needs to be loaded during querying.

1.3. Differences to the conference paper

This paper represents an extended version of our previous work with the title "Min-Max Modifiable Nested Octrees (M^{3}NO): Indexing Point Clouds with Arbitrary Attributes in Real Time", which we presented at the CGVC 2024 [17]. Due to the new research aspects we were able to explore since the original publication, the paper underwent significant changes and contains the following edits and additional material:

- One of the key improvements is a *new and efficient file format* that enables not only high-performance storage and data retrieval but also data reduction through real-time compression (see Section 3.4). This also allows our system to now support any attribute, even those not defined in the LAS file format specification (see Section 3.3).
- We added a *query language that allows users to define filtering criteria in an intuitive way* (see Section 5). This enables spatial queries based on bounding boxes and camera view frustum, LOD queries (level of detail), and includes filters for arbitrary attributes that can be combined with boolean operators.

- Since our implementation has evolved quite significantly, we have performed *a whole new evaluation* that is *much more detailed* than the one from our previous publication (see Section 7). We now apply our approach to various publicly available real-world datasets (a subset of AHN4, the Kitti dataset, and the Paris-Lille-3D benchmark) and compare it to the state-of-the art software pgPointCloud and PotreeConverter 2.0.
- Since we now also focus on real-time visualization, we included *latency measurements* in our evaluation (see Section 7.2.4) to assess how long it takes for a point to be displayed on screen after it was recorded and if interactivity can be achieved.
- Finally, in comparison to our previous paper, we have *updated the section on related work* to include recently published research (see Section 2). We have also added *more details to the technical descriptions* of our data structure (see Section 3).

1.4. Outline

The remainder of the paper is structured as follows. We first summarize related work in the area of point cloud indexing and compare it to our approach (Section 2). We then present the M^{3}NO data structure and give details on its components (Section 3). After this, we describe the indexing process (Section 4) and how querying works (Section 5). We present the software architecture for real-time visualization (Section 6) and evaluate our approach based on the research goals defined above (Section 7). The paper finishes with a conclusion and an outlook on future work (Section 8).

2. Related work

Research in the area of point cloud indexing can be divided into two sub-areas: data structures for spatial indexing (Section 2.1) and approaches that allow arbitrary point cloud attributes to be indexed or that combine spatial and attribute indexing (Section 2.2). In this section, we summarize existing approaches and discuss their suitability for real-time indexing.

2.1. Spatial indexing

A well-known data structure for spatial indexing is the k-d-tree [19]. Each node in such a tree represents a plane that subdivides a k-dimensional space along one axis. For each tree level $l \in \mathbb{N}_{\geq 0}$, a different axis $i \in [0, k)$ is selected, such that $i = l \bmod k$. If the position of the partition plane is chosen based on the distribution of the points in space, the tree becomes well-balanced. This requires all points to be known and sorted along the corresponding axis. Adding points later either results in an unevenly distributed tree or causes expensive rebalancing. This renders the k-d-tree unsuitable for real-time indexing where points might arrive at any time.

The same applies to the R-tree, a data structure similar to a one-dimensional B-tree but supporting multiple dimensions [20]. In an R-tree, each node represents a bounding rectangle that encloses a group of points or other bounding rectangles. Since the number and area of the bounding rectangles depend on the points in the tree and inserting new points requires rebalancing, real-time indexing cannot be implemented efficiently.

An efficient data structure for $\mathbb{R}^3$ is the octree. It divides the space independently of the points to be indexed and centrally along all three axes into eight sub-spaces [21,22], which are then further divided recursively. The octree does not require rebalancing, but it still can be unbalanced if the data is unevenly distributed. Building an octree requires the bounding box of all points to be known in advance, which makes it unsuitable for real-time applications.

Another way to index a spatially distributed set of points in linear time is to use a space-filling curve such as a Z-order curve [23] or a Hilbert curve [24]. As the name implies, space-filling curves traverse the entire n-dimensional space and assign an ascending index to each point (or cell) visited. This maps the n-dimensional space to one dimension, which in turn allows points to be inserted and queried through binary search in $\mathcal{O}(n \log n)$ time. Most space-filling curves preserve locality to a certain degree, so that points that are close to each other in space are also close to each other on the one-dimensional curve. They have been used for point cloud indexing [25], distributed processing [26], as well as to build continuous levels of detail [27]. Since curve indexes can typically be calculated in constant time and independently of each other, space-filling curves enable efficient and massively parallel point cloud indexing [28]. Interestingly, there is a close relation between Z-order curves and octrees. The curve index that should be assigned to a point P in $\mathbb{R}^3$ corresponds to its Morton code, which can be computed by interleaving the bits of its x, y, and z values. This is equivalent to traversing an octree, where in each level, all axes are divided centrally, resulting in $2^3 = 8$ subspaces. Using a fast algorithm to sort the points after assigning indexes to them, such as Radix sort, it is therefore possible to build a compact representation of an octree as a linear array in $\mathcal{O}(n + nw)$ time, where w is constant and equals the desired word length of the Morton code.

The Modifiable Nested Octree (MNO) is a special type of octree, where each node can contain a set of points stored in a regular three-dimensional grid [29]. This allows the space to be divided both spatially and into levels of detail. The points in the root node correspond to the lowest level of detail (LOD 0), and each lower level in the tree corresponds to a higher level of detail. The authors also present a way to add points outside the bounding box by creating new parent nodes of the current root node until the points are contained. For our real-time indexing, we want to incrementally build the level of detail structure in a top-down manner, which does not work when creating new root nodes.

The SimLOD approach, developed by Schütz et al., enables GPU accelerated indexing of points in a data structure similar to MNOs [30]. As an incremental construction, it allows visualization during the indexing process and achieves a speed of 580 million points per second. However, true real-time indexing during LiDAR scans is not possible with this approach, because the bounding box of the data must be known in advance.

A solution to this problem was presented by Kocon [28] and later adapted for MNOs by us [16]: In a large regular grid, an arbitrary number of MNO trees are created. As soon as points fall into a new cell of the grid, a new MNO tree is created in that cell. The approach presented in this paper is based on this idea. We describe it in detail in Section 3.

2.2. Attribute indexing

A straightforward way to index a point cloud by attribute is to store it in a database and reuse the existing indexing capabilities. Dobos et al. for example, describe a concept for a database model, in which each row represents a single point [31]. The coordinates and attributes are all stored in different columns. A primary database index is then created on the coordinates and further indexes can be created on the attributes. Although the work of Dobos et al. is promising, it remains just a concept. They do not present an actual implementation.

Storing individual points as records in a database is not efficient given the enormous number of points and the large number of attributes. The PostgreSQL extension pgPointCloud provides a solution by grouping nearby points into patches [32]. Each patch stores the minimum, maximum, and average value for each coordinate and attribute. As Bormann et al. have shown, the performance of pgPointcloud in terms of indexing and querying is not competitive and in many cases even slower than working directly on the raw data [33], which makes it unsuitable for real-time indexing. Nevertheless, grouping points into chunks instead of saving each of them individually is a useful method to keep the size of the index small and to simplify compression [34].

Another way of indexing attributes is to use space-filling curves. Similar to spatial indexing (Section 2.1), they can map the multi-dimensional attribute space to one dimension. For example, HistSFC presented by Liu et al. [27] supports indexing of both spatial coordinates and attributes. However, Liu et al. discovered that space-filling curves do not scale well when applied to an arbitrary number of attributes. In their paper, they show that in tests on the AHN2 dataset, indexing with four dimensions already results in queries with a false positive rate of up to 164%, which increases rapidly with additional dimensions. To deal with this problem, some approaches automatically rank which attributes in a point cloud are most important to index and should, therefore, be included in the index structure [35]. Since our aim in this paper is to index arbitrary attributes (and not just a limited number), space-filling curves are not an option for attribute indexing.

Ladra et al. present a novel approach to represent point clouds as a so-called compact data structure [36]. Such a data structure can be completely kept in memory and obtains very fast query times, even though it is compressed. The approach is even able to index attributes. However, since everything must be kept in memory, it does not support arbitrarily large datasets. The biggest dataset we used for our evaluation was 365 GB in total (see Section 7.1), which does not fit into main memory, even compressed. Also, Ladra et al. do not focus on real-time indexing or visualization. Their approach is not optimized for fast point insertion and does not provide levels of detail, which are required to visualize large point clouds.

Nevertheless, an interesting aspect of the approach by Ladra et al. is the attribute index structure. They extend the Binned Min–Max Quadtree introduced by Zhang and You [37] and apply it to LiDAR point clouds. The points are sorted into an octree, and for each subtree, they are sorted by attribute. Since the algorithm has access to the whole dataset, the minimum and maximum values of all attribute values within each subtree are known. This allows entire subtrees to be skipped during querying if the value ranges do not match the query condition. The basic idea of combining a spatial index (i.e. an octree) with an attribute index and storing attribute value ranges for subtrees fits well with our goals and provides a basis for the data structure presented in this paper.

3. The M³NO data structure

The main building block of our approach is a data structure called the *Min–Max Modifiable Nested Octree (M³NO)*. It works in real time and supports both spatial indexing and indexing of arbitrary attributes. For this, it contains two different components that we describe in detail in the following (Sections 3.1 and 3.2). We also explain how points are represented in memory (Section 3.3) and on disk (Section 3.4).

3.1. Spatial data structure

The spatial data structure is based on the results of our previous work [16]. It consists of an arbitrary number of Modifiable Nested Octrees (MNO) [38] in a regular grid with a fixed cell size. In a real-time scenario, the total size of the point cloud is not known in advance, as new points are recorded continuously. Due to this, the indexing algorithm creates new root nodes each time points arrive in grid cells where no MNO tree has been created yet. This allows the data structure to dynamically grow as required.

Each MNO consists of octree nodes, which recursively divide the space into eight subspaces. This halves the size of the nodes along each axis with every additional level in the tree. All nodes contain a regular grid with a default of 256^3 cells to store the points. A maximum of one point can be stored in each cell. Note that, in most areas, the points lie on two-dimensional manifolds (the surfaces visible to the LiDAR scanner) and the regular grids are only sparsely filled.

By halving the node size at each level along all axes, the distances between the regular grid cells are also halved. This results in higher densities of points deeper in the tree, and thus a higher level of detail. Fig. 2 illustrates this for a two-dimensional point cloud.

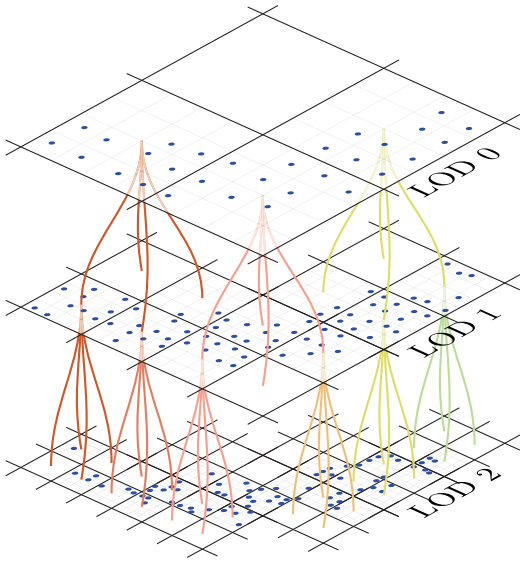


Fig. 2. Modifiable Nested Octree from our previous work [16]: In the upper region, three root nodes are generated, corresponding to the lowest level of detail (LOD). In the higher LODs, many smaller nodes contain more points in total and thus provide a higher level of detail. For illustration purposes, each node in the figure has a grid of 4×4 cells in which the points are stored.

The data structure saves all nodes with the corresponding points as individual files. When nodes need to be read or modified, they are loaded into a cache with a definable maximum capacity. If it is exceeded, the least recently used nodes are written back to disk. Using this out-of-core approach allows point clouds exceeding the size of main memory to be indexed.

3.2. Attribute data structure

Our approach allows points to be filtered based on the values of attributes. For this, users can define queries. Range queries such as *attr(intensity > 35)* or equality queries such as *attr(classification == 3)* are supported (see our query language in Section 5). The query result will then only contain the subset of the point cloud where the attributes match the filter conditions.

In order to accelerate the attribute based filtering of the point cloud, we have built a secondary index structure that is integrated with the spatial index. It is based on the idea by Ladra et al. [36] but is adapted for real-time capability. For every node of the spatial data structure, our index stores information about the distribution of attribute values in its subtree, which can be used during querying to determine if a subtree can be skipped. We implemented two representations of this distribution, the *Value Range Index* and the *Bin List Index* (see Sections 3.2.1 and 3.2.2).

Attributes are indexed completely independently of each other. The users can configure any number of attribute indexes. Each index is managing a single attribute.

Due to the small size of the information stored in the attribute indexes, they can completely operate in memory. They only need to be loaded from or stored to disk upon startup and shutdown.

3.2.1. Value range index

For each node, we store the minimum and maximum attribute value for all points in its subtree. Attributes with vector data types such as *color* use the component-wise minimum and maximum. The attribute

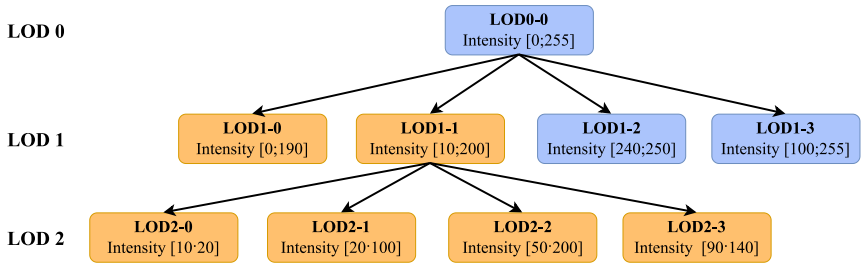


Fig. 3. Example of attribute index structure while filtering for intensities in the range [230; 255]. Orange nodes are discarded by the attribute index structure, blue nodes are loaded.

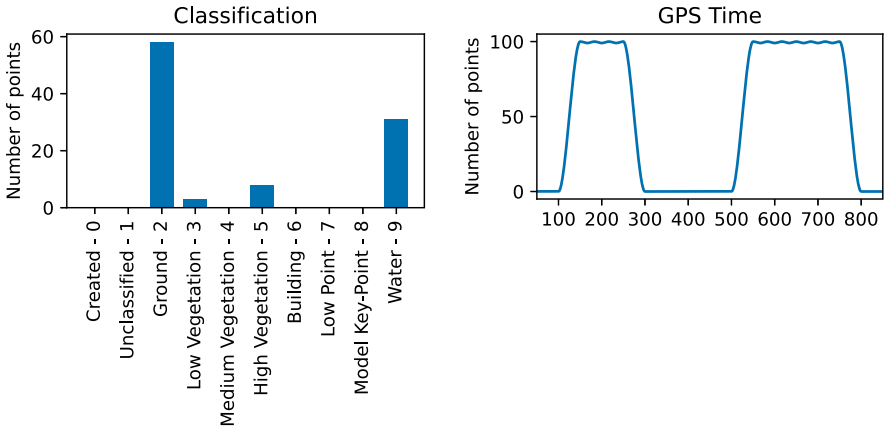


Fig. 4. Distribution of attribute values in a node.

value range is updated during indexing, whenever new points are added to the node.

An example of this attribute index structure for the *intensity* attribute can be seen in Fig. 3. For legibility, this example uses a two-dimensional quadtree instead of a three-dimensional octree. For each node, the index stores the value ranges of the *intensity* attribute values of the points in the underlying subtree (including the node itself).

In the example, the point cloud is filtered for *intensity* values in the range [230; 255]. The attribute index can accelerate range queries like this by eliminating entire subtrees, which in turn do not need to be loaded from disk. In the example, node LOD1-1 has a value range that does not overlap with the query range. Therefore, it does not contain a matching point and its entire subtree can be skipped. In contrast, node LOD1-3 has a value range, that overlaps with the query range. Here, it is necessary to load the node from disk and to sequentially filter all contained points. The attribute index also helps detect nodes where no sequential filtering needs to be performed, which further improves query speeds. The value range of node LOD1-2 is completely contained in the query range. This tells us that all contained points must be matching. The node needs to be loaded from disk, but filtering out non-matching points is not necessary.

3.2.2. Bin list index

The value range index works well for continuous numerical attributes such as *intensity*. However, some attributes are of a categorical nature. An example would be the *classification* attribute. In this case, the minimum and maximum are meaningless, as they do not imply that every value in between is actually used. The example shown in Fig. 4 (left) would lead to an attribute range of 2 to 9. However, there are several classification values between 2 and 9 that are not present in the node. If the user queried for *Building* (class 6), this would lead to a false positive, as there are no points classified as *Building* even though it is within the attribute value range.

There are similar cases for some continuous point attributes. The example in Fig. 4 (right) shows a possible distribution of the *GPS Time* attribute, when the sensor moves through the area of the node twice. Here, the attribute range is 100 to 800. If the user queried for anything between 300 and 500, this would lead to a false positive node, because there are no points within this value range despite being within the total value range.

Ladra et al. [36] solve this problem by calculating a histogram of the attribute value distribution. During querying, they can check the histogram bins that correspond to the query range. If these bins are empty, the node does not need to be loaded. Calculating histograms and defining their ranges does, however, require knowing the minimum and maximum of the attribute values beforehand. Since we are indexing points already during capturing, we do not know the final attribute value bounds.

Instead of histograms, we only store a set of non-empty bins. During indexing, we adapt the bin width so that the number of non-empty bins stays below a threshold k.

An empty node starts with the minimal bin size. Whenever points are added to the node, they are checked against the existing bins and any new bin is added to the bin list. When the number of bins reaches the threshold k, the bin size is doubled. Doubling the bin size means that neighboring bins are merged, which in turn reduces the number of bins again. Note, that this can be done without checking every point, because we know which of the old bins falls into which doubled bin.

Fig. 5 shows an example of this approach. Here, the number of bins should stay below $k = 6$. We want to add the values 10, 11, 42, 43, 23, 22, 20, 45, 46, 16, 18, 12, 19, 15 to the node in this order. At the beginning, the bin size is 1. We can add the first five values 10, 11, 42, 43, and 23 without exceeding k (first row). The next value 22 would add a sixth bin to the list and therefore exceed the threshold k, so the bin size is doubled before adding more values. This merges the bins 10..10 and 11..11 as well as 42..42 and 43..43, thereby reducing the number

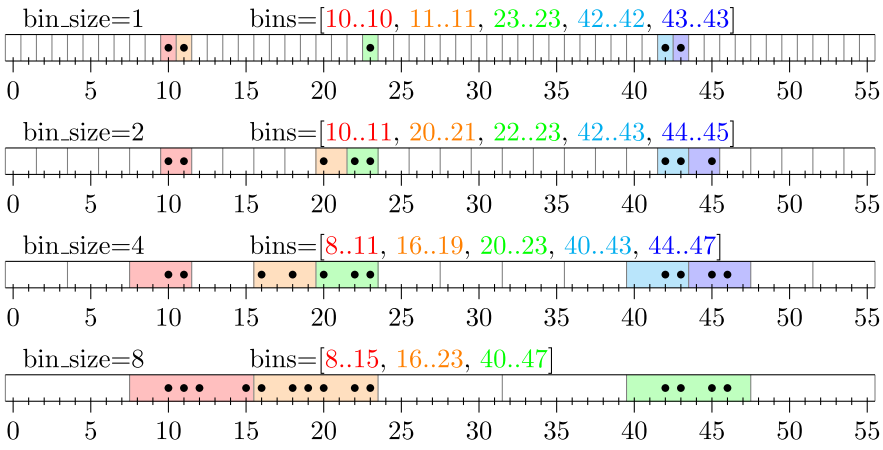


Fig. 5. Example of how the bin list index of a single node is constructed incrementally.

of bins again. We can now add the values 22, 20, and 45 (second row) without exceeding k. Note, that not all values necessarily add a new bin to the list. For example, value 22 fell into the existing bin 22..23, keeping the bin list unchanged. After doubling the bin size once again, we can add the values 46, 16, and 18 (third row). After that, the bin size is doubled one final time, and we add the remaining values 12, 19, and 15. This leaves us with the final bin list in row 4.

For attributes with floating point values, the bins are not regularly sized. Instead, they reflect the floating point accuracy, so that in the minimal bin size, each bin contains exactly one floating point value. For attributes with vector data types, the bins are calculated component-wise, so that each bin essentially becomes a bounding box inside the attribute vector space.

During querying, we check each bin against the queried value range. If we queried the value range [30; 35] in the example above, we could skip the node subtree, because the query range does not overlap with any of the three bins.

3.3. Representation of point data in memory

We use the open source library *pasture* [39] for the in-memory representation of the point data. The library comes with data structures for storing point data with arbitrary attributes, either in an *array-of-structures* (AoS) or *structure-of-arrays* (SoA) layout. While an SoA can be beneficial for cache efficiency when only accessing single attributes, we opted for the AoS layout. It massively simplifies operations that work on full points including all attributes, such as copying points from an input buffer into a node, as required during the LOD sampling. This layout stores points in a single byte buffer, where each point is the concatenation of its position and attributes. An additional piece of metadata, the *point layout*, is used to look up the offset of each attribute within the points, so individual attributes of a specific point can be accessed by reading or writing at the memory location $base\_address + point\_index \times point\_size + attribute\_offset$.

3.4. Representation of point data on disk

Our out-of-core data structure swaps nodes to disk once they are evicted from the least recently used cache. For representing point data on disk, we have developed a custom file format. It allows for storing point clouds with arbitrary attributes, is straight-forward to read or write, and comes with support for compression.

In the implementation presented in our previous paper [17], we relied on the LAS [11] file format, which is widely used for LiDAR point clouds. It can store common point attributes such as *intensity*, *return number*, or *classification*. Which attributes are used as well as their byte positions are determined by selecting one of eleven predefined point data record formats. With LAZ [40], there is also a compressed variant based on a specialized compression algorithm. It can achieve very good compression ratios of 5:1 or better.

Due to the inflexibility of the predefined point data record formats, LAS cannot be used for storing point clouds with arbitrary attributes. While it is possible to add extra attributes, the predefined ones cannot be altered. Setting unused attributes to zero would waste disk space and negatively impact the indexing performance because of the additional I/O overhead. Also, while compressed storage is desirable, the LAZ compression algorithm is too computationally expensive to be used during real-time indexing.

To overcome these drawbacks, we have developed a custom data format for encoding point data. Files consist of a header followed by the actual point data. The header contains the following information:

Magic number The magic number acts as an identifier for our file format.

Version number A version number allows for future backwards compatible changes to the file format.

Endianness The endianness of the point attributes. Point attributes are always stored in the native endianness of the platform the file is created on.

Compression Determines if the point data is compressed or not.

Number of points The size of the point cloud.

Point layout The list of point attributes. Each attribute is identified by a name and has a type. In addition to the type, the size of the attribute values in bytes is stored. This is needed for attribute types such as *ByteArray*, which do not have a statically known size.

The encoding of the point data depends on whether compression is enabled or not. Without compression, each point is stored as the concatenation of its attributes without any padding and in the order defined by the point layout in the header. This is identical to our in-memory representation. Therefore, encoding and decoding uncompressed point data is trivial to implement and very efficient.

For compression, we are using the LZ4 compression algorithm [41]. LZ4 is fast enough to run in real time while still achieving reasonable compression ratios. Before compression, the point data is transposed, as shown in Fig. 6. There are multiple compression contexts, where compression context i is responsible for the i'th byte of each point. This is beneficial for the compression rate, because it is very likely that similar data is stored together. This is very similar to the LAZER format

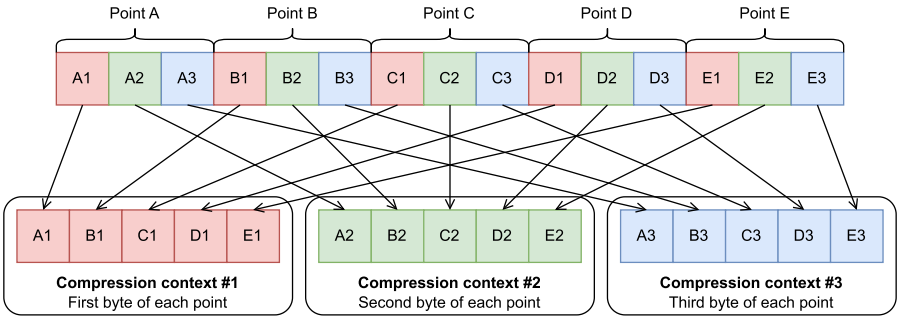


Fig. 6. Transposing point data for compression.

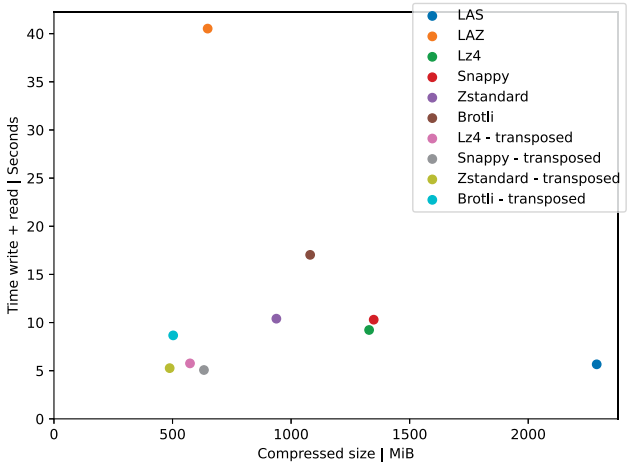


Fig. 7. Comparison of different compression algorithms.

proposed by Bormann et al. [42,43], although they are transposing the point data on a per-attribute basis, while we work on a per-byte basis. This has a positive impact on the compression rate as leading zeros, for example, end up one after the other and can be compressed quite efficiently.

The choice of LZ4 as a compression algorithm is based on tests that we have conducted, whose results can be seen in Fig. 7. We compressed a point cloud with LZ4, Snappy, Zstandard and Brotli, each as-is and with the data transposition described above. We have also given the size and timing for LAS and LAZ for reference. The result was, as long as the point data is transposed, all 4 compression algorithms perform similarly well, both in terms of compression ratio and read/write time. Since there is no clear winner, we chose LZ4 for our final implementation.

4. Indexing process

The indexer receives incoming points in real time to insert them into the data structure. For this, each node has an additional in-memory point buffer that collects points for the associated subtree. This is called the inbox of a node. For each incoming point, the indexer looks for an MNO tree in the regular grid of MNO trees. If there is one, it places the point in the inbox of the root node of that tree. If no tree exists at that location, a new one is created.

Multiple worker threads are responsible for inserting the points from the inboxes into the nodes in parallel. Each thread repeatedly performs the following steps:

1. A node with a non-empty inbox is chosen for processing. A priority is assigned to every node, of which the one with the highest priority is selected. In previous work, we have compared different ways of calculating the priority [16]. Here, we exclusively use the *NrPointsTaskAge* priority function that is shown in the following equation.

$$Priority = NrPoints \cdot 2^{TaskAge}$$

The variable *NrPoints* refers to the size of the inbox and *TaskAge* refers to the time for which the inbox has been non-empty.
2. The attribute indexes are updated with all points in the inbox. For each indexed attribute, the value range or bin list of the points in the inbox is calculated. This intermediate result is then merged into the existing value range or bin list of the node in the attribute index. We are using a mutex to synchronize access to each attribute index. Only the merge operation is in its critical path. It is very fast and does not depend on the number of points in the inbox, leading to low mutex contention.
3. The points in the node are either loaded from the cache, or from the corresponding file on disk. If it does not exist yet, a new one is created.
4. All points are removed from the inbox and inserted into the node. This is done using a grid center sampling approach [14, 29]. Among all points that fall into the same cell of the point grid of the node, the one that is closest to the center of the cell is accepted into the node. All remaining points are added to the inbox of the corresponding child node.
5. The updated node is stored in the cache. If the maximum cache size is reached, the least recently used node is evicted and stored to disk.
6. If there are any running real-time queries, they are notified of the changed node, so that they can update their query result accordingly.

5. Querying

Our implementation allows for flexible querying of the indexed point cloud based on spatial criteria, attribute values, or LOD, which combines all aspects of our index structure. We have developed a query language that allows users to define queries in an intuitive way.

Spatial queries filter the point cloud based on axis-aligned bounding boxes or a camera view-frustum. Bounding boxes are defined using their respective minimum and maximum coordinate values:

```
aabb([x1, y1, z1], [x2, y2, z2])
```

View-frustum queries are defined by a set of camera parameters. The query result will only contain points that are visible within the view frustum of this camera. Furthermore, the view-frustum query also performs an LOD selection, so that the projected point spacing is not larger than *max_distance* pixels apart. This leads to parts that are closer to the camera having a finer LOD and the LOD level becoming coarser further away from the camera.

```
view_frustum(
  camera_pos: [x1, y1, z1],
  camera_dir: [x2, y2, z2],
  camera_up: [x3, y3, z3],
  fov_y: f4,
  z_near: f5,
  z_far: f6,
  window_size: [x7, y7] ,
  max_distance: f8
)
```

The *level of detail* of the query result can be limited. This will only load nodes in the first n levels of the octree structure.

```
lod(n)
```

The point cloud can be filtered based on *attribute values*. Range and equality checks are supported.

```
attr(Classification == c)
attr(Classification != c)
attr(Intensity < i)
attr(Intensity > i)
attr(Intensity <= i)
attr(Intensity >= i)
attr(i1 < Intensity < i2)
attr(i1 < Intensity <= i2)
attr(i1 <= Intensity < i2)
attr(i1 <= Intensity <= i2)
```

Subqueries can be combined with the boolean operators *and*, *or*, and the unary ! operator for negation. This allows users to build complex custom queries out of simple building blocks. The following example query selects any point in LOD 3 or smaller that has a classification value other than 5 and that is within one of the two given bounding boxes:

```
lod(3)
  and !attr(Classification == 5)
  and (
    aabb([10, 15, 0], [20, 45, 60])
    or aabb([20, 30, 0], [30, 40, 60])
  )
```

Querying can either be performed on a fully indexed point cloud or while the point cloud is still being captured. *One-off queries* run once and then return the filtered query result. *Real-time queries*, on the other hand, can be executed during the capturing process. After returning the initial query result, the results of real-time queries will be kept up to date as more points are added to the index. In Section 6, we present a 3D-Viewer that makes use of this by translating its camera position into a real-time view frustum query.

To execute the query, the algorithm traverses the MNO tree. Each node is checked against the query. This can have one of three outcomes:

Negative No point in the current subtree matches the query. Spatial queries evaluate to *negative* if the bounding box of the subtree node is completely out of the query region. Attribute queries evaluate to *negative* if there is no overlap between the distribution of attribute values within the subtree and the queried value range.

Partial If no universal statement can be made about all points in the node, it is marked as *partial*. If the bounding box of a node overlaps with the query region of a spatial query but is not completely contained in it, then some points in the node will be positive and some will be negative, making the complete node only partially positive. Similarly, if it is known from the distribution of attribute values in the node that there will be both points inside and outside the queried attribute range, the node is partial.

Positive A node is marked as positive if it is known that all points in the node will match the query. For spatial queries this is the case if its bounding box is completely within the query area. Attribute queries evaluate to *positive* if the attribute values are completely contained in the queried value ranges.

The query is evaluated recursively along its parse tree. This means that spatial and attribute subqueries are evaluated first. The results of the subqueries are then combined according to the truth tables given in Fig. 8.

If the final outcome for the entire query is *Negative*, then the node is not loaded from disk and its subtree can be skipped from further traversal. Nodes marked as either *Partial* or *Positive* need to be loaded from disk or cache. While positive nodes can be added to the query result as they are, partial nodes require sequential point filtering:

For each subquery, a bitmap is calculated that contains, for every point, if that point matches the subquery. The bitmaps are combined according to the boolean operations used in the query. The final bitmap is used to filter the points in the node.

After the complete tree has been traversed, one-off queries can terminate. If the query is a real-time query, it keeps running. It receives a stream of IDs of nodes that have changed from the indexer. Each changed node is tested against the query again. If the outcome is *Negative*, the change is ignored. Otherwise, the query result is updated with the new contents of the node, possibly after another sequential filtering step (in case the outcome was *Partial*).

6. Real-time visualization architecture

We implemented the data structure in our open-source software package *Lidarserv*, which is available on GitHub [18]. In this section, we present its architecture (see Fig. 9). A central server is responsible for processing incoming points and queries. Clients can then send new points to the server or submit queries on the existing point cloud via network. This architecture is similar to the concept of the point cloud server presented by Cura et al. [34].

The central indexing server stores the indexed point cloud and processes incoming points and client queries. When new clients connect to the server, it transmits a set of data:

- The point cloud coordinate system
- The list of defined attributes
- The compression state
- The current bounding box of the indexed point cloud

Clients can then answer what type of client they are:

CaptureDevice: This type of client sends points to the server for indexing.

Viewer: This client can send queries to the server to get all points that match the query.

The architecture provides a high degree of flexibility, which makes it possible to cover a wide range of use cases. Depending on the application, clients and server can run on the same device, or a viewer client can be installed on a tablet with limited computing power (e.g. tablets). Clients can be implemented in any programming language. They just need to follow the Lidarserv network protocol. This allows for connecting any type of scanner without modifications to the server. We have already implemented a number of clients for common applications that are described in the following.

and	Neg	Par	Pos
Neg	Neg	Neg	Neg
Par	Neg	Par	Par
Pos	Neg	Par	Pos

or	Neg	Par	Pos
Neg	Neg	Par	Pos
Par	Par	Par	Pos
Pos	Pos	Pos	Pos

not	
Neg	Pos
Par	Par
Pos	Neg

Fig. 8. Truth tables for the boolean operators on the node query outcomes. The abbreviations Neg, Par and Pos refer to Negative, Partial and Positive node query outcomes, respectively.

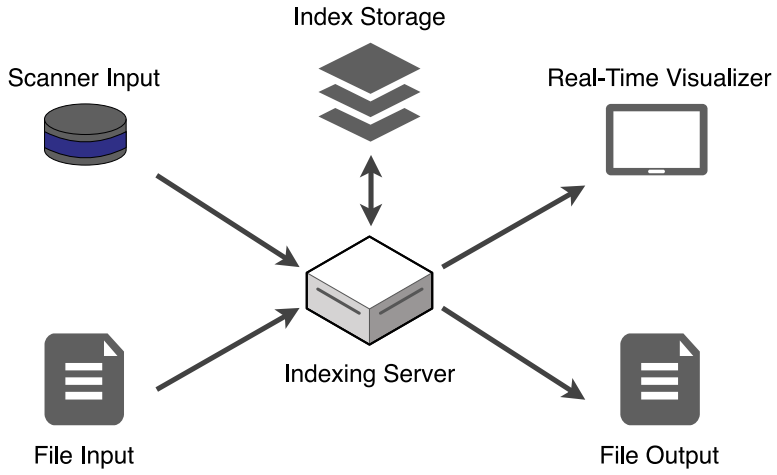


Fig. 9. Lidarserv architecture: The server processes incoming points and queries. Clients can send new points or queries via network.

The *scanner input client* is needed for the typical application of Lidarserv. This *CaptureDevice* client sends all the points in real time to the indexing server, using the binary point data representation described in Section 3.4. Further preprocessing steps, e.g. for real-time colorization or classification, can also be implemented here.

For testing purposes, the *file input client* is able to read LAS files. The data is then sorted by GPS time and transmitted to the server at the original speed (or faster). This client makes it possible to use our indexing approach to perform conventional batch indexing. Incremental indexing of multiple files is also possible.

The *viewer client* displays the existing point cloud in real time. The selected camera position determines which view-frustum queries are sent to the server. The server then processes these real-time queries using the index and sends the corresponding points back to the viewer. Furthermore, additional spatial and attribute filters can be defined and combined with the view-frustum queries. The queries can be configured to send entire octree nodes containing the searched points, or to filter the nodes point-wise on the server. When sending compressed octree node files, the query time can be reduced. When filtering points on the server, the data which has to be transferred over the network and stored in the RAM of the viewer clients can be minimized. A screenshot from the viewer client can be seen in Fig. 10. A video of the visualization during the indexing process was submitted together with the manuscript as supplementary material.

For testing and export purposes, there is also an *output file client* that sends queries and filter requests defined via CLI to the server and stores the results in a LAS file.

7. Evaluation

To evaluate our approach, we implemented the M^{3}NO data structure in Lidarserv and measured various properties using multiple exemplary test datasets. Some measurements were also carried out with pgPointCloud and PotreeConverter 2.0 for comparison. The test environment and dataset are described in Section 7.1. The results are presented in Section 7.2.

7.1. Test setup

The software package includes an evaluation program that indexes a point cloud as fast as possible to determine the maximum indexing speed. It also measures various metrics ranging from querying times to latency. All measurements were executed on a virtual Linux machine with a *Xeon Gold 6140* processor, 32 GB of RAM, and SSD storage with a write speed of 330 MB/s and a read speed of 340 MB/s.

We used three different publicly available datasets for the evaluation (see screenshots in Fig. 11). The subset of AHN4 is our largest test dataset with almost 11.5 billion points (365 GB). AHN4 covers the whole Netherlands and was acquired using state-of-the-art airborne laser scanning technology [12]. Our subset contains consecutive flight paths with continuous time to simulate real-time acquisition. It was acquired at an average rate of 1.22 million points per second (Mpts/s). The KITTI dataset from the KIT in Karlsruhe, Germany is a car-based scanning dataset with 805 million points (33 GB) [44]. The points do not have a GPS timestamp, so the recording speed cannot be specified. The points are ordered by their point source ID. In this case, this is the order in which they were recorded. The Paris-Lille-3D dataset is a French car-based mapping dataset with continuous time [45]. We only used the data from the city of Lille, which was recorded at a rate of about 0.41 Mpts/s. It contains 120 million points (3.36 GB). We have sorted the points of each dataset by GPS time so that we can later index them in their actual recording order to better reflect real-time indexing.

All attributes we filtered on were indexed using a Value Range Index (see Section 3.2.1). In addition, we used a Bin List Index (see Section 3.2.2) for the Classification and Color attributes. For each dataset, we configured the root node size of the index structure so that the nodes at the lowest level of detail cover most of the dataset to ensure smooth visualization even when viewing the entire point cloud. We then chose the number of levels of detail so that at the highest level of resolution, point grid cells are no more than one centimeter apart. We limited the least recently used cache size to 500 octree nodes for KITTI and Lille, and 10 000 nodes for AHN4. This way, we avoided having the majority of each dataset in RAM, which would have been advantageous for smaller datasets. This makes it easier to compare

Fig. 10. Screenshots of the viewer client while indexing the KITTI dataset.

(a) AHN4
Multiple flight lines

(b) KITTI
Multiple streets

(c) Lille
One long street

Fig. 11. Screenshots of the datasets used for evaluation.

the readings from the small datasets with larger datasets. We used 32 indexing threads for each dataset. For the pgPointCloud measurements, we chose a PDAL pipeline [46] and a PDAL chipper with a capacity of 400 to divide the points into patches for the insertion process [46].

7.2. Results

We took measurements for each of the four goals of the paper. The goal of real-time capability (*G1*) was checked by measuring the maximum possible indexing speed (see Section 7.2.1). The data reduction goal (*G2*) was measured by running several sample queries and recording the number of nodes and points returned (Section 7.2.2). The time reduction goal (*G3*) was evaluated by measuring the execution time of sample queries (Section 7.2.3). The real-time visualization goal (*G4*) was checked by measuring the latency of individual points from insertion to visualization (Section 7.2.4).

7.2.1. Real-time indexing

In order to reach the goal of real-time indexing, our data structure had to be capable of processing more points per second than received from the LiDAR sensor. To measure the maximum possible indexing speed of Lidarserv, we monitored the inboxes of all nodes and made sure that a constant load of new points was maintained. These measurements were performed with both compression enabled and disabled. For Lidarserv, we used LZ4 compression as described in Section 3.4. PotreeConverer 2.0 uses Brotli and pgPointCloud its own dimensional compression. The measured indexing speeds are listed in Fig. 12.

The measurements show that our approach can easily outperform the point rates of state-of-the-art airborne and car-based LiDAR scanners. The AHN4 dataset, which was acquired at an average of 1.22 Mpts/s, was indexed with 2.29 Mpts/s. On the Lille dataset, we even achieved an indexing speed of 5.24 Mpts/s which was 13 times faster than its acquisition speed. With these results, our indexing is significantly faster than the acquisition speeds, fulfilling the goal of real-time indexing (*G1*). The uncompressed test runs of AHN4 and Lille were faster than the compressed runs due to the additional computation required for compression. It is noteworthy that the overhead caused by attribute indexing on top of the spatial index is quite small.

The PotreeConverter measurements were 20% to 211% faster compared to Lidarserv. This is because PotreeConverter does not perform real-time or incremental indexing and therefore has more optimization options, such as splitting the cloud into equal sized chunks of points. This is impossible to do in real time. pgPointCloud on the other hand achieved only about 3–5% of the speed of Lidarserv. It also crashed when inserting larger point clouds due to excessive memory usage, which is why it is not included in the charts for AHN4 and KITTI.

7.2.2. Query data reduction

The second goal addresses the quality of the index in its ability to reduce the amount of data that has to be loaded. This is measured in the number of nodes and points that cannot be eliminated by the index despite not matching the query. For this, we defined a set of sample queries (see Fig. 13). Tables 1, 2, and 3 list all queries and the percentage of points returned for each dataset. We focus on attribute queries in order to better show the effect of the spatial distribution of the attribute values on the data reduction.

Fig. 14 provides an overview of how many points are searched for each query and how many were returned. This allows the efficiency of the attribute index structures to be assessed. Fewer points returned by both range filters indicate a higher efficiency and a faster query. The various queries have very different selectivities, which is why the results also differ between the queries.

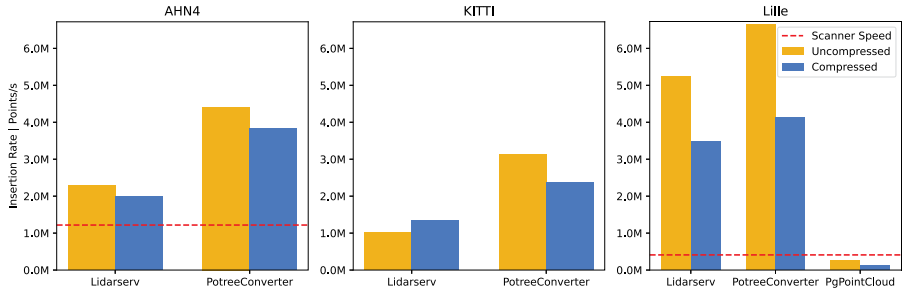


Fig. 12. Insertion speed comparison: Lidarserv is able to exceed the scanner speed, enabling real-time indexing. PotreeConverter is faster than Lidarserv, but does not allow incremental indexing. pgPointCloud ran out of memory on the larger datasets and is significantly slower than Lidarserv on the Lille dataset.

Table 1
Sample queries with the percentage and absolute number of points returned (AHN4 dataset).

Query	Percentage	Points
Classification Bridges **attr**(Classification == 26)	0.12 %	13 556 318
Classification Buildings **attr**(Classification == 6)	4.29 %	493 893 685
Classification Ground **attr**(Classification == 2)	77.87 %	8 974 515 216
Classification Vegetation **attr**(Classification == 1)	10.34 %	1 191 705 807
Intensity High **attr**(Intensity > 1400)	48.99 %	5 646 198 411
Intensity Low **attr**(Intensity <= 20)	0.00 %	182
Time Big Slice **attr**(GpsTime < 270521185)	100.00 %	11 524 770 195
Time Small Slice **attr**(270204590 < GpsTime < 270204900)	0.00 %	984
Time Medium Slice **attr**(269521185 < GpsTime < 269522000)	4.36 %	502 720 135

Table 2
Sample queries with the percentage and absolute number of points returned (KITTI dataset).

Query	Percentage	Points
Classification Buildings **attr**(semantic == 11)	12.14 %	97 802 750
Classification Ground **attr**(semantic <= 12)	54.35 %	437 793 886
Pointsource1 **attr**(35 <= PointSourceID <= 64)	11.43 %	92 079 676
Pointsource2 **attr**(208 <= PointSourceID <= 248)	15.18 %	122 307 911
RGB **attr**(ColorRGB <= [10,10,10])	0.02 %	139 878
Time Big Slice **attr**(199083995 <= GpsTime <= 466372692)	33.18 %	267 288 697
Time Medium Slice **attr**(687577131 <= GpsTime <= 805552832)	14.65 %	117 975 739

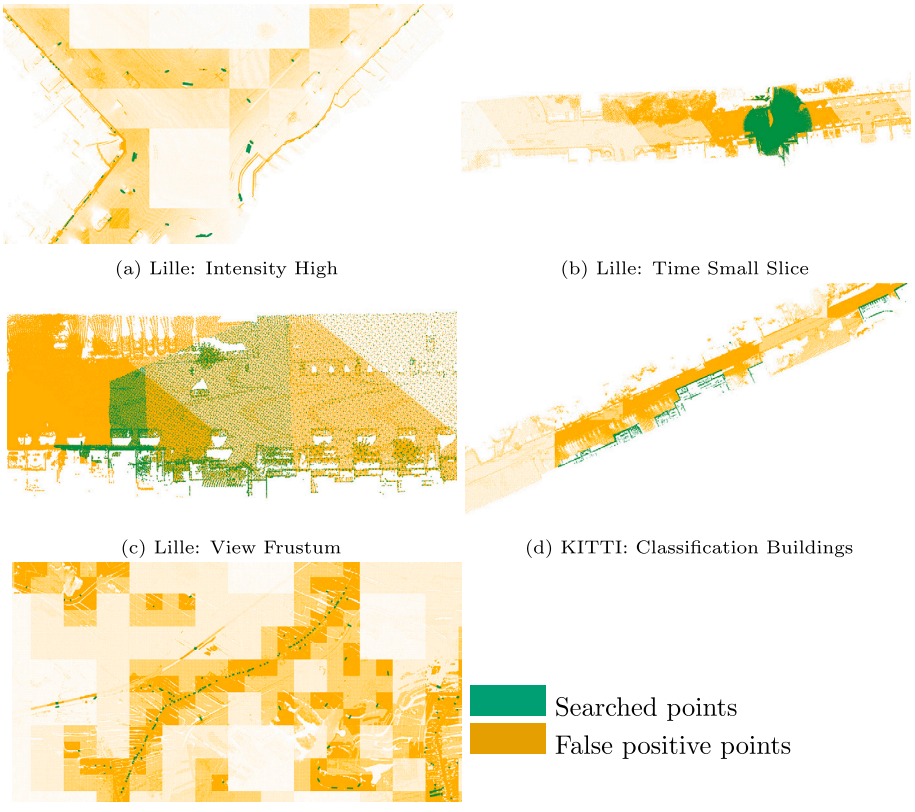
Lidarserv allows a large number of points to be removed before sequential filtering. A greater range of points can be filtered out when filtering for attributes, where points spatially close to each other have similar values (e.g. *time* or *classification*).

Fig. 15 compares the percentage of nodes/patches returned from the attribute index. This shows that proportionally many more nodes are filtered out than points. This is because the search criteria do not apply to a large range of small nodes, while large nodes often contain the points being searched for.

Node reduction significantly reduces the number of disk accesses of the query, even if point reduction is not optimal. This is especially true for cases where the index eliminates many nodes with only a few points.

Table 3
Sample queries with the percentage and absolute number of points returned (Lille dataset).

Query	Percentage	Points
Intensity High **attr**(Intensity > 128)	0.11 %	131 345
Intensity Low **attr**(Intensity <= 2)	5.59 %	6 702 129
Time Big Slice **attr**(GpsTime < 20000)	96.56 %	115 872 130
Time Small Slice **attr**(GpsTime > 23000)	3.44 %	4 127 870
PointsourceID >= 10 **attr**(PointSourceID >= 10)	2.84 %	3 404 478
PointSourceID >= 5 **attr**(PointSourceID >= 5)	14.51 %	17 414 734
ScanAngleRank <= 45 **attr**(ScanAngleRank <= 45)	1.93 %	2 321 714
ScanAngleRank <= 90 **attr**(ScanAngleRank <= 90)	13.03 %	15 638 173
View Frustum **view_frustum**(camera_pos: [−560.45, −584.87, 47.29], camera_dir: [0.75, 0.65, −0.12], camera_up: [0.0, 0.0, 1.0], fov_y: 0.78, z_near: 3.9, z_far: 3994169.6, window_size: [500.0, 500.0], max_distance: 10)	0.04 %	51 959



(a) Lille: Intensity High

(b) Lille: Time Small Slice

(c) Lille: View Frustum

(d) KITTI: Classification Buildings

(e) AHN4: Classification Bridges

Fig. 13. Sample queries: the searched points are displayed in green. The false positive points returned from Lidarserv are marked in yellow. It can be observed that only the high resolution LODs are loaded around the searched points. As the distance to the searched area increases, only the larger, lower resolution nodes are loaded.

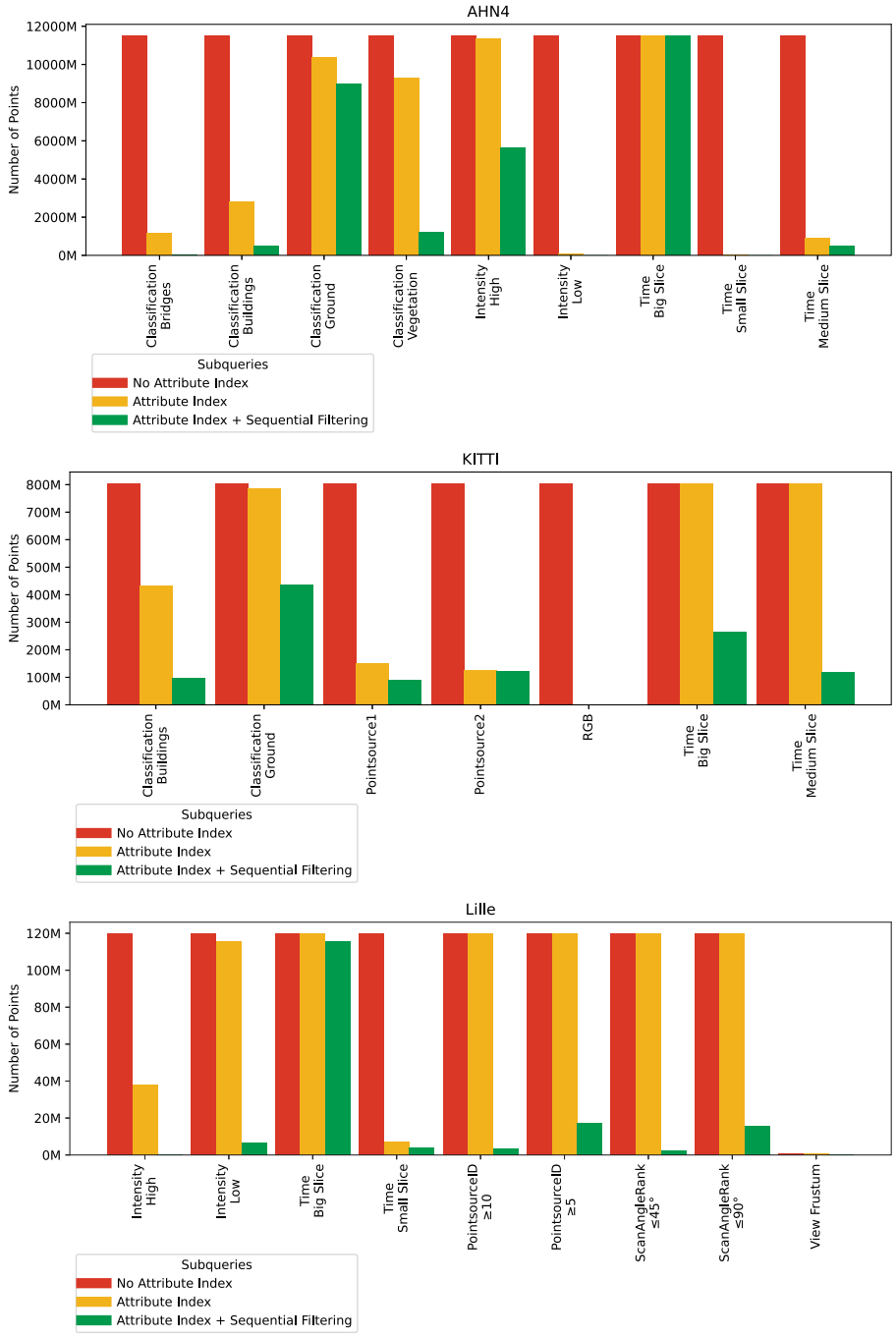


Fig. 14. Query comparison by number of points.

The results show that the attribute index is very effective in reducing the number of nodes to be loaded, which is why we consider the goal of data reduction to be achieved (*G2*).

7.2.3. Query time reduction

The index structure should decrease query execution times for end users or connected applications, enabling interactivity and short waiting times. In Fig. 16, we compared the Lidarserv query times of the sample queries.

Lidarserv demonstrated high querying speeds of 1.34 Mpts/s with compression enabled and 1.74 Mpts/s without. Compression had an impact on the querying speed, but can still be beneficial in a networked environment due to the reduced amount of data to be transferred. In Fig. 17, we compare the sizes of the indexed point clouds to evaluate the efficiency of the compression. Lidarserv achieves a compression rate of about 63% with the given data sets, while PotreeConverter 2.0 achieves 81%. For comparison, the LAZ Compression achieves approximately the same compression rate as the Brotli Compression of PotreeConverter 2.0.

The results demonstrate, that in all test queries, the attribute index significantly reduces query time in comparison to sequential point filtering. Especially for the attributes and queries that provide a high data reduction as shown in the previous section, the query time can be reduced by using the attribute index. In the worst case, the query takes as long as sequential filtering, because the overhead of checking the attribute index per node is very small. In the best case, we can achieve query times in the range of milliseconds even on massive datasets,

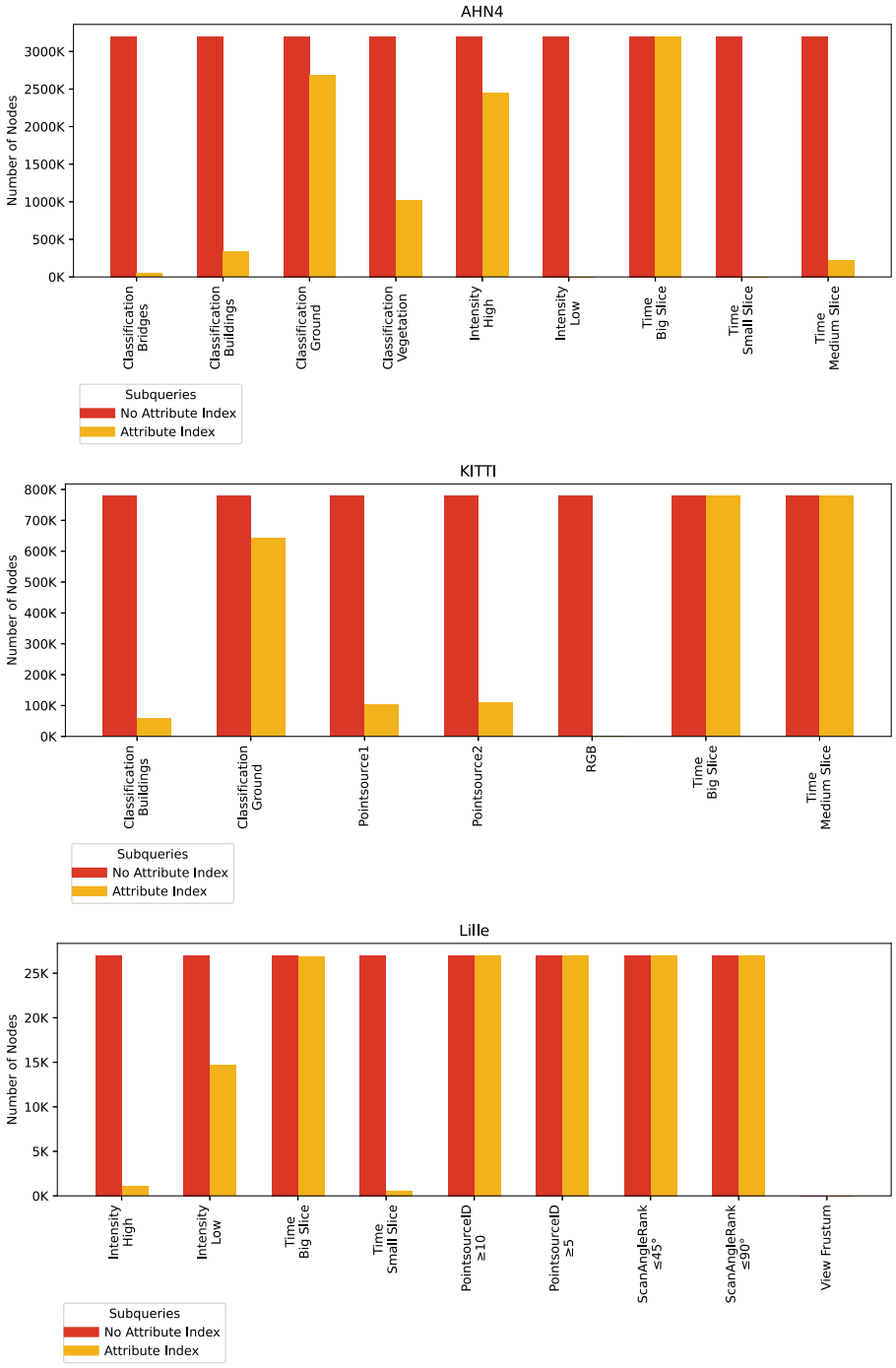


Fig. 15. Query comparison by number of nodes.

e.g. with the small time slice query on the AHN4 dataset (0.03 s). With these results, we meet the goal of reducing query time (*G3*).

7.2.4. Real-time visualization

To achieve the goal of real-time visualization, points must arrive at the visualizer with a low delay after being captured. For this, we measured the time it took the points to be rendered on screen in a real-time query. We replayed the Lille dataset with constant 1 Mpts/s multiple times. Simultaneously, we ran different real-time queries and measured the time it took to return newly inserted points. Fig. 18 shows the distribution of the latencies per query.

Queries with a large number of returned points have a higher latency than those with few points. This is due to the fact that more nodes need to be updated and transferred. In terms of visualization, the frustum query on the right-hand side of the diagram is particularly important. Here, half of the points are visible after only 4.2 ms, and the entire area is visible after 31 ms. To evaluate the latencies, there are a number of different latency guidelines for real-time applications, such as those summarized by Attig et al. [47]. To verify the real-time visualization goal, the threshold of 100–200 ms from Seow [48] for instantaneous interactions is applicable in our case. We measured an average latency of less than 160 ms for 95% of all queried points. The frustum query relevant for visualization is much lower. This means that our visualization can be classified as interactive and real-time capable, which meets our goal of real-time visualization (*G4*).

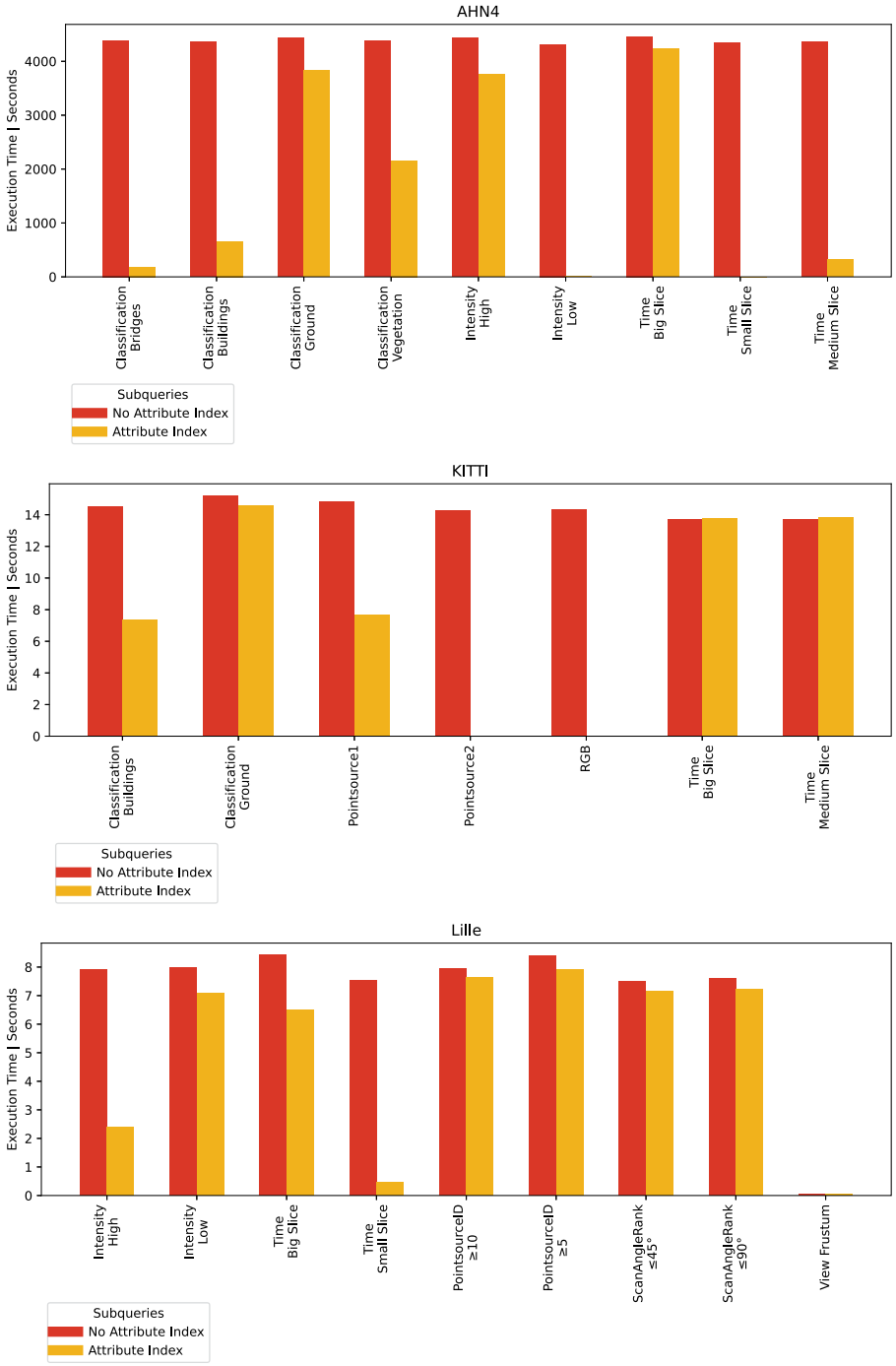


Fig. 16. Query comparison by number of time.

Looking at the breakdown of latencies per LOD in Fig. 19, our measurements show a slight increase with each LOD. This is due to the fact that points have to go through an additional tree level for each LOD. The root nodes are displayed very quickly, with 95% of the points arriving in less than 98 ms. Each level of detail then takes a little longer, with a 95% confidence interval of 221 ms for the highest LOD.

7.3. Scalability and limitations

Point cloud datasets can become massively large, with many terabytes and hundreds of billions of points. To show that our approach works for these datasets and does not slow down over time, we also measured the maximum possible indexing speed of the airborne AHN dataset as indexing progressed, shown in Fig. 20.

The insertion rate achieved during indexing is shown in blue in the graph. The actual recording speed of the scanner is shown in red.

Up to about 7 billion points, the points were recorded in evenly spaced, slightly overlapping flight lines, resulting in a relatively constant insertion rate. After that, there are crossing lines over existing areas (the airplane apparently turned around), which slows down the indexing speed considerably as areas that have already been recorded have to be reloaded from disk. In the last two billion points, there are smaller flight lines that overlap with the previously recorded lines for a small section. This is why there are so many variations here.

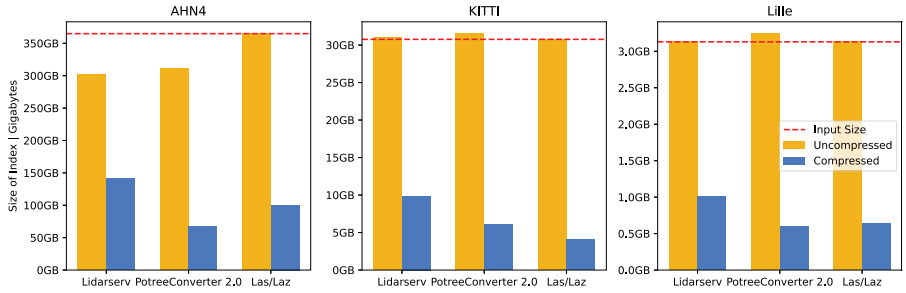


Fig. 17. Index sizes of Lidarserv compared to PotreeConverter 2.0.

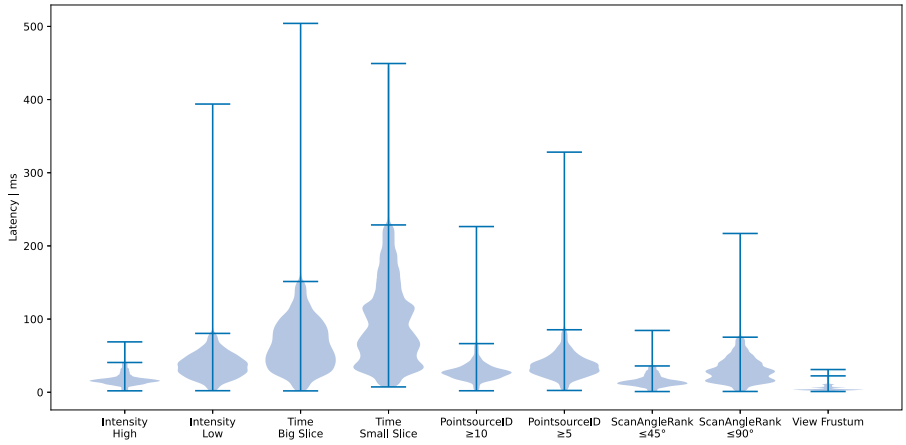


Fig. 18. Latency comparison for different queries. The horizontal lines show minimum, 95%-percentile and maximum.

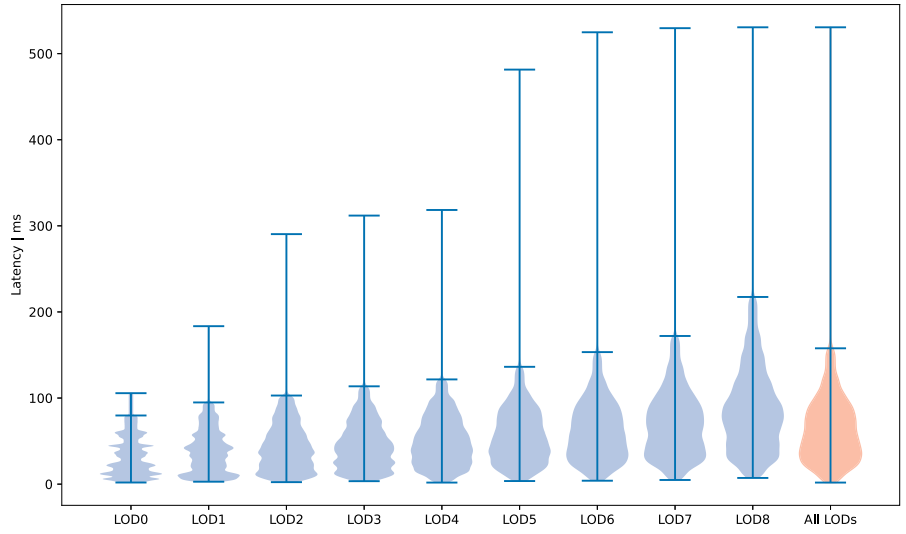


Fig. 19. Latency while querying the full point cloud, listed by LODs. The horizontal lines show minimum, 95%-percentile and maximum.

In summary, it can be said that the indexing speed depends on the amount of data recorded in the same area and whether the airplane has flown over an area several times, but if only new areas are recorded, the speed remains constant over time. This means that the indexing is scalable to larger datasets linear to the input size.

However, since the attribute index is stored entirely in RAM, there are some limitations. The exact size our system can handle depends on the point cloud and the number of nodes generated, as attribute information is stored for each node. Nevertheless, the amount is very small, especially for the area index. The worst case index sizes (in bytes) can be calculated as follows:

Range Index: $\#\text{nodes} \times (\text{sizeof}(\text{attr\_type}) \times 2 + 13\text{byte})$

Bin List Index: $\#\text{nodes} \times (\text{sizeof}(\text{attr\_type}) \times \#\text{bins} + 30\text{byte})$

For the AHN dataset, a total of 3 200 134 nodes were created. This results in the following index sizes:

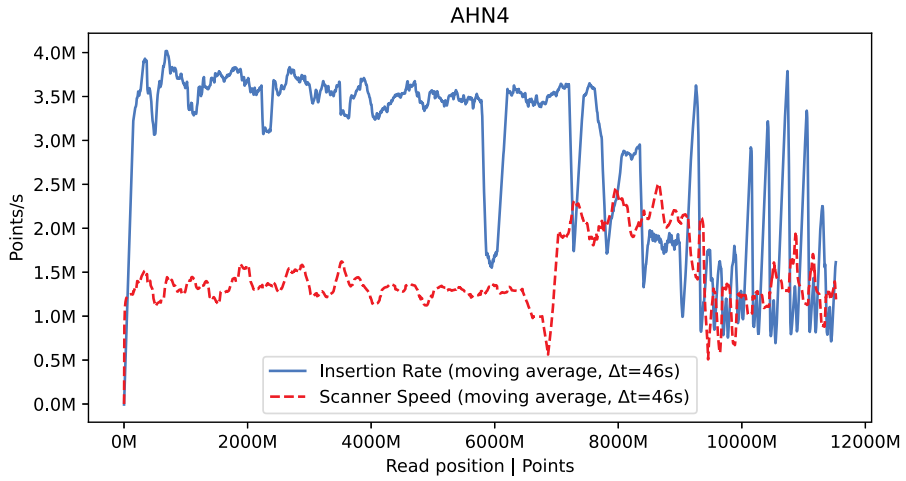


Fig. 20. Indexing speed over time for the AHN4 dataset.

- Range Index - Classification (1 byte): 46 MiB
- Range Index - Intensity (2 bytes): 52 MiB
- Range Index - GpsTime (8 bytes): 89 MiB
- Bin List Index - Classification (1 byte): 140 MiB

The total size of the attribute indexes is about 327 MiB. This is a very small amount of RAM compared to the size of the point cloud itself, which is several hundred gigabytes. The attribute index can be stored in RAM without any problems.

If the indexing speed is insufficient, the points will accumulate in the inboxes of the nodes. The inboxes are only stored in RAM, which means that a lack of memory could eventually cause a crash. In this case the points would be lost. To avoid this, a fallback to disk would have to be implemented in the future.

7.3.1. Bin list index

In Section 3.2.2, we introduced two possible index structures to accelerate attribute range queries. In this section, we compare the performance of the Value Range Index and the Bin List Index and discuss the advantages and disadvantages in detail.

To compare both attribute index structures, we took measurements with no attribute index, the Value Range Index and the Bin List Index (16 bins). For the KITTI dataset, we indexed the *classification* attribute. Fig. 21 shows that the attribute indexes have only a small impact on the indexing speed. The Value Range Index had no negative impact on the point rate at all. We even observed a slightly higher point rate when using the Value Range Index compared to using no attribute index. This is a surprising result, which we think is caused by slight performance fluctuations of the virtual machine that we used for our tests. With the Bin List Index, we observed a slowdown of 4.7% compared to the Value Range Index.

We tested the query performance of each index structure. We used queries of the form **attr**(semantic == C), where C is one of the classes that are present in the KITTI dataset. Fig. 22 shows the number of nodes each index returned.

The results demonstrate that both attribute indexes provide better performance than without attribute indexing. For the classes with very small and very large values (classes 6, 7, 44), both indexes perform similar. However, for the classes closer to the middle (especially classes 10 to 20), the Bin List Index clearly outperforms the Value Range Index.

The same effect also shows in the query times that we observed. For queries on the classification attribute, the Bin List Index is almost always faster than the Value Range Index. Only if Value Range Index and Bin List Index return close to the same number of nodes, the Value Range Index is slightly faster.

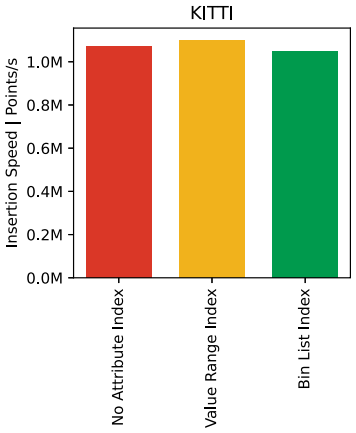


Fig. 21. Comparison of the impact of the different attribute indexes types on the indexing speed.

The Bin List Index was especially designed for attributes where the values are not uniformly distributed, like *Classification* or *GpsTime*. When we tested it with other attributes, we found that it offers no benefits over the Value Range Index.

In general, we can name four major differences between the two index structures:

- The Bin List Index is superior at filtering the nodes for queries on the *Classification* or *GpsTime* attributes. This leads to a better query performance for these attributes. The Value Range Index does not perform well for these non-uniformly distributed attributes, because it only stores the minimum and maximum of the values in each node but no details about the distribution of values between these bounds.
- The Bin List Index is more complex to compute than the Value Range Index. In our measurements, the effect on the point insertion rate was small, but we only indexed a single attribute for these tests. When inserting n new points into a node, updating the Value Range Index takes $\mathcal{O}(n)$ time. However, our implementation for updating the Bin List Index involves sorting the new attribute values, which makes our implementation only $\mathcal{O}(n \log n)$.
- The Value Range Index is faster when checking a node against a query. While the Value Range Index only has to check the queries against the value range, the Bin List Index has to check the query

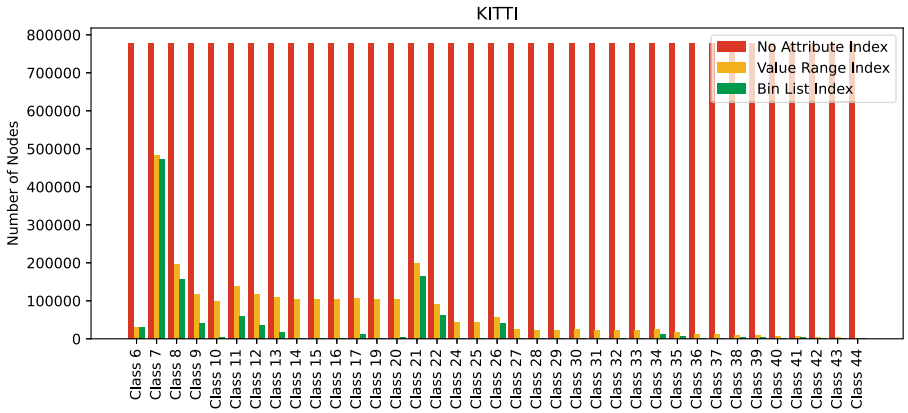


Fig. 22. Comparison of the node filtering capabilities of each attribute index type.

against each bin. For *Classification* and *GpsTime*, the improved node filtering conceals this effect, but for other attributes, this added complexity shows in slightly slower query times.
- As shown in Section 7.3, the Value Range Index has a lower memory footprint than the Bin List Index.

Therefore, we recommend using the Value Range Index for most attributes. The Bin List Index should be used for selected attributes with a favorable distribution of attribute values.

8. Conclusion

In this paper, we presented an integrated approach for indexing, querying, and visualizing 3D LiDAR point clouds with arbitrary attributes. This process works in real-time, i.e. while the data is being recorded. We described our data structure M^{3}NO and its individual components. This included details about how points are represented in memory as well as on disk with our custom file format. Further, we described the indexing process and how querying is performed. We also introduced an intuitive query language that can be used to interactively filter the indexed data. After presenting the architecture for real-time visualization, we evaluated our approach using various real-world datasets. This included a comparison of our implementation to state-of-the art solutions.

The work presented here was based on previous publications where we covered real-time spatial indexing and real-time indexing of arbitrary attributes. The integrated approach, the advancements we have achieved in our research, and the optimizations we have made in our implementation now enable live visualization for the first time. To the best of our knowledge, there is no other approach in literature that addresses this challenge. Existing works either focus on spatial or attribute indexing, only support a limited set of attributes, or do not support real-time visualization.

In terms of the research goals we defined in Section 1.1, we were able to meet all of them. Our evaluation results show that our approach can index points at least as fast as they are recorded by a laser scanner (*G1*). Our data structure M^{3}NO is organized in a way that allows whole subtrees of octree nodes to be eliminated early in the query process, which reduces the amount of data that needs to be loaded from disk (*G2*). The evaluation results also show that queries on point clouds are very fast, even if the dataset is massive. The AHN4 dataset is several hundred gigabytes in size and can be filtered based on attributes in a few milliseconds (*G3*). The same applies to view-frustum and LOD queries, which are particularly required to be fast to enable live visualization. This is evident from the latency measurements, where view frustum queries received incoming points after no more than 31 ms. They prove that our implementation is indeed able to visualize large point clouds in real-time and that it can be classified as interactive (*G4*).

We want to specifically highlight that our approach works out of core and is therefore able to index arbitrarily large datasets and is not limited by the available main memory, as we have shown in Section 7.3. Furthermore, the fact that an LOD structure is implicitly created enables visualization of arbitrarily large point clouds even on devices with limited computing capacity. Only a necessary portion of the point cloud needs to be loaded and rendered on screen.

In the future, this will make it possible that operators of mobile mapping systems use a device such as a tablet to visualize and query the data while it is being recorded. This will enable use cases such as live quality assurance, which in turn is expected to save time and money as issues in the point clouds can be fixed right away and repeated scan jobs can be avoided.

Nevertheless, the evaluation results also show that the performance of our approach highly depends on the spatial distribution of the attribute values. In Section 7.2.2, differences in the evaluated datasets become apparent. Queries for the attributes *GPS time* or *classification* work very well, but the performance varies largely for the *intensity* attribute, for example. Our approach works best if the searched points lie in a limited number of octree nodes and many subtrees can be eliminated as quickly as possible. However, a uniform attribute value distribution where almost all octree nodes contain points has a negative impact on performance. Designing the M^{3}NO this way was a deliberate decision. It can be considered a tradeoff we were prepared to accept to be able to implement real-time visualization. Other offline indexing approaches are able to achieve better query times and smaller index sizes as they have access to the whole dataset and can find an optimized indexing strategy, but, because of this, they are of course not real-time capable.

In the future, we want to further work on our data structure and make it more flexible, especially with regard to real-time modifications. At the moment, the index is static and only allows new points to be added. Already indexed data cannot be modified without recreating the whole index. Live modifications would be beneficial for various use cases. For example, spatial transformations would enable real-time simultaneous localization and mapping (SLAM) loop closure. Furthermore, downstream processing steps such as colorization or classification could make use of the index structure and work in parallel to the indexing process. The index could be updated in real time as soon as their results arrive.

Finally, we will explore the possibilities of graphics processing units (GPUs) and how they can be used to further improve the performance of our implementation. We expect that making use of massive parallelization will even out the differences between our implementation and other offline approaches or even surpass their performance.

In summary, we consider our work a major step with a high impact on practical applications and the scientific community. For the first time, real-time visualizations of 3D LiDAR point clouds while they are being recorded are possible. At the same time, our results open up new possibilities for further research.

CRediT authorship contribution statement

Paul Hermann: Writing – review & editing, Writing – original draft, Visualization, Software, Methodology, Investigation, Conceptualization. **Michel Krämer:** Writing – review & editing, Writing – original draft, Supervision, Conceptualization. **Tobias Dorra:** Writing – review & editing, Writing – original draft, Visualization, Software, Methodology, Investigation, Conceptualization. **Arjan Kuijper:** Supervision.

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.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.cag.2025.104254.

Data availability

All datasets are publicly available.

References

[1] Valero E, Bosché F, Bueno M. Laser scanning for BIM. J Inf Technol Constr 2022;27:486–95. http://dx.doi.org/10.36680/j.itcon.2022.023.

[2] Pyszny K, Sojka M, Wróżyński R. LiDAR based urban vegetation mapping as a basis of green infrastructure planning. In: Jóźwiakowski K, Ciesielczuk T, editors. E3S Web Conf 2020;171:02008. http://dx.doi.org/10.1051/e3sconf/202017102008.

[3] Krämer M, Bormann P, Würz HM, Kocon K, Frechen T, Schmid J. A cloud-based data processing and visualization pipeline for the fibre roll-out in Germany. J Syst Softw 2024. http://dx.doi.org/10.1016/j.jss.2024.112008.

[4] Lumbreras M, Diarce G, Martin-Escudero K, Campos-Celador A, Larrinaga P. Design of district heating networks in built environments using GIS: A case study in Vitoria-Gasteiz, Spain. J Clean Prod 2022;349:131491. http://dx.doi.org/10.1016/j.jclepro.2022.131491.

[5] Li B, Hou J, Li D, Yang D, Han H, Bi X, Wang X, Hinkelmann R, Xia J. Application of LiDAR UAV for high-resolution flood modelling. Water Resour Manag 2021;35(5):1433–47. http://dx.doi.org/10.1007/s11269-021-02783-w.

[6] Pfeifer N, Briese C. Geometrical aspects of airborne laser scanning and terrestrial laser scanning. In: International archives of photogrammetry, remote sensing and spatial information sciences. vol. 36, 2007, p. 311–9.

[7] Alvites C, Marchetti M, Lasserre B, Santopuoli G. LiDAR as a tool for assessing timber assortments: A systematic literature review. Remote Sens 2022;14(18). http://dx.doi.org/10.3390/rs14184466.

[8] Ton B, Ahmed F, Linssen J. Semantic segmentation of terrestrial laser scans of railway catenary arches: A use case perspective. Sensors 2022;23(1):222. http://dx.doi.org/10.3390/s23010222.

[9] Wu C, Yuan Y, Tang Y, Tian B. Application of terrestrial laser scanning (TLS) in the architecture, engineering and construction (AEC) industry. Sensors 2021;22(1):265. http://dx.doi.org/10.3390/s22010265.

[10] Li X, Liu C, Wang Z, Xie X, Li D, Xu L. Airborne LiDAR: State-of-the-art of system design, technology and application. Meas Sci Technol 2021;32(3):032002. http://dx.doi.org/10.1088/1361-6501/abc867.

[11] American society for photogrammetry and remote sensing (asprs), LAS specification 1.4—R14. 2019, https://www.asprs.org/wp-content/uploads/2019/03/LAS_1_4_r14.pdf. [Accessed 15 January 2025].

[12] AHN4 dataset, AHN4 landsdekkende dataset 2020. 2020, https://www.ahn.nl/ahn-4. [Accessed 15 January 2025].

[13] Bormann P, Krämer M. A system for fast and scalable point cloud indexing using task parallelism. In: Smart tools and apps for graphics - eurographics Italian chapter conference. The Eurographics Association; 2020, http://dx.doi.org/10.2312/stag.20201250.

[14] Schütz M, Ohrhallinger S, Wimmer M. Fast out-of-core octree generation for massive point clouds. Comput Graph Forum 2020;39(7):155–67. http://dx.doi.org/10.1111/cgf.14134.

[15] Hobu. Entwine. 2024, https://entwine.io/. [Accessed 15 January 2025].

[16] Bormann P, Dorra T, Stahl B, Fellner DW. Real-time indexing of point cloud data during LiDAR capture. In: 40th Computer Graphics & Visual Computing Conference: CGVC 2022. Eurographics-European Association for Computer Graphics; 2022, p. 65–73. http://dx.doi.org/10.2312/cgvc.20221173.

[17] Hermann P, Krämer M, Dorra T, Kuijper A. Min-Max modifiable nested octrees (M3NO): indexing point clouds with arbitrary attributes in real time. In: Hunter D, Slingsby A, editors. Computer Graphics and Visual Computing (CGVC). The Eurographics Association; 2024, http://dx.doi.org/10.2312/cgvc.20241235.

[18] Dorra T, Hermann P. Lidarserv GitHub repository. 2024, https://github.com/igd-geo/lidarserv/releases/tag/v2.1.0. [Accessed 17 January 2025].

[19] Bentley JL. Multidimensional binary search trees used for associative searching. Commun ACM 1975-09;18(9):509–17. http://dx.doi.org/10.1145/361002.361007.

[20] Guttman A. R-Trees: A dynamic index structure for spatial searching. In: Proceedings of the 1984 ACM SIGMOD international conference on management of data - SIGMOD '84. ACM Press; 1984, p. 47. http://dx.doi.org/10.1145/602259.602266.

[21] Meagher D. Geometric modeling using octree encoding. Comput Graph Image Process 1982;19(2):129–47. http://dx.doi.org/10.1016/0146-664X(82)90104-6.

[22] Finkel RA, Bentley JL. Quad trees a data structure for retrieval on composite keys. Acta Inform 1974;4(1):1–9. http://dx.doi.org/10.1007/BF00288933.

[23] Morton GM. A computer oriented geodetic data base and a new technique in file sequencing. Technical Report, International Business Machines Company; 1966, International Business Machines Company New York.

[24] Hilbert D. Über die stetige Abbildung einer Linie auf ein Flächenstück. Math Ann 1891;38:459. http://dx.doi.org/10.1007/978-3-662-38452-7_1.

[25] Guan X, Van Oosterom P, Cheng B. A parallel N-dimensional space-filling curve library and its application in massive point cloud management. ISPRS Int J Geo-Inf 2018;7(8):327. http://dx.doi.org/10.3390/ijgi7080327.

[26] Liu K, Boehm J, Alis C. Change detection of mobile LIDAR data using cloud computing. Int Arch Photogramm Remote Sens Spat Inf Sci 2016;41:309–13. http://dx.doi.org/10.5194/isprsarchives-XLI-B3-309-2016.

[27] Liu H, Van Oosterom P, Meijers M, Verbree E. An optimized SFC approach for nD window querying on point clouds. ISPRS Ann Photogramm Remote Sens Spat Inf Sci 2020;VI-4/W1-2020:119–28. http://dx.doi.org/10.5194/isprs-annals-VI-4-W1-2020-119-2020.

[28] Kocon K, Bormann P. Point cloud indexing using big data technologies. In: 2021 IEEE international conference on big data (big data). IEEE; 2021, p. 109–19. http://dx.doi.org/10.1109/BigData52589.2021.9671659.

[29] Schütz M. Potree: Rendering large point clouds in web browsers. 2016, Technische Universität Wien, Wiedeń.

[30] Schütz M, Herzberger L, Wimmer M. SimLOD: Simultaneous LOD generation and rendering for point clouds. Proc ACM Comput Graph Interact Tech 2024;7(1). http://dx.doi.org/10.1145/3651287.

[31] Dobos L, Csabai I, Szalai-Gindl JM, Budavári T, Szalay AS. Point cloud databases. In: Proceedings of the 26th international conference on scientific and statistical database management. ACM; 2014, p. 1–4. http://dx.doi.org/10.1145/2618243.2618275.

[32] Ramsey P, Blottiere P, Brédif M, Lemoine E. pgPointcloud - a PostgreSQL extension for storing point cloud (LIDAR) data. 2023, https://pgpointcloud.github.io/pointcloud/index.html. [Accessed 15 January 2025].

[33] Bormann P, Krämer M, Würz H. Working efficiently with large geodata files using ad-hoc queries. In: Proceedings of the 11th international conference on data science, technology and applications. SCITEPRESS - Science and Technology Publications; 2022, p. 438–45. http://dx.doi.org/10.5220/0011291200003269.

[34] Cura R, Perret J, Paparoditis N. A scalable and multi-purpose point cloud server (PCS) for easier and faster point cloud data management and processing. ISPRS J Photogramm Remote Sens 2017;127:39–56. http://dx.doi.org/10.1016/j.isprsjprs.2016.06.012.

[35] Nathan V, Ding J, Alizadeh M, Kraska T. Learning multi-dimensional indexes. In: Proceedings of the 2020 ACM SIGMOD international conference on management of data. ACM; 2020, p. 985–1000. http://dx.doi.org/10.1145/3318464.3380579.

[36] Ladra S, Luaces MR, Paramá JR, Silva-Coira F. Compact and indexed representation for LiDAR point clouds. Geo- Spat Inf Sci 2024;27(4):1035–70. http://dx.doi.org/10.1080/10095020.2022.2121664.

[37] Zhang J, You S. Supporting web-based visual exploration of large-scale raster geospatial data using binned min-max quadtree. In: Gertz M, Ludäscher B, editors. Scientific and statistical database management. Berlin, Heidelberg: Springer Berlin Heidelberg; 2010, p. 379–96.

[38] Scheiblauer C. Interactions with gigantic point clouds [PhD thesis], Institute of Computer Graphics and Algorithms, Vienna University of Technology.; 2014.

[39] Bormann P, contributors. Pasture. 2024, https://github.com/igd-geo/pasture/. [Accessed 06 January 2025].

[40] Isenburg M. LASzip: Lossless compression of LiDAR data. Photogramm Eng Remote Sens 2013;79. http://dx.doi.org/10.14358/PERS.79.2.209.

[41] Collet Y. LZ4. 2025, https://lz4.org/. [Accessed 10 January 2025].

[42] Bormann P. Improving the efficiency of point cloud data management PhD thesis, TU Darmstadt; 2024, http://dx.doi.org/10.26083/tuprints-00027526.
[43] Bormann P, Krämer M, Würz HM, Göhringer P. Executing ad-hoc queries on large geospatial data sets without acceleration structures. SN Comput Sci 2024;5(5). http://dx.doi.org/10.1007/s42979-024-02986-z.
[44] Geiger A, Lenz P, Stiller C, Urtasun R. Vision meets robotics: The KITTI dataset. Int J Robot Res (IJRR) 2013.
[45] Roynard X, Deschaud J-E, Goulette F. Paris-Lille-3D: A large and high-quality ground-truth urban point cloud dataset for automatic segmentation and classification. Int J Robot Res 2018;37(6):545–57. http://dx.doi.org/10.1177/0278364918767506.
[46] PDAL - point data abstraction library, PDAL - point data abstraction library. 2024, https://pdal.io/. [Accessed 15 January 2025].
[47] Attig C, Rauh N, Franke T, Krems JF. System latency guidelines then and now – is zero latency really considered necessary? In: Harris D, editor. Engineering psychology and cognitive ergonomics: cognition and design. Cham: Springer International Publishing; 2017, p. 3–14. http://dx.doi.org/10.1007/978-3-319-58475-1_1.
[48] Seow SC. Designing and engineering time: The psychology of time perception in software. Addison-Wesley Professional; 2008.