AGILE: GIScience Series, 5, 15, 2024. https://doi.org/10.5194/agile-giss-5-15-2024
Proceedings of the 27th AGILE Conference on Geographic Information Science, 4–7 Sept. 2024.
Editors: Alison Heppenstall, Mingshu Wang, Urska Demsar, Rob Lemmens, and Jing Yao.
This contribution underwent peer review based on a full paper submission.

Random Data Distribution for Efficient Parallel Point Cloud Processing

Balthasar Teuscher [1] and Martin Werner[1]

[1]Professorship Big Geospatial Data Management, School of Engineering and Design, Technical University of Munich, Munich, Germany

Correspondence: Balthasar Teuscher (balthasar.teuscher@tum.de)

Abstract. Current point cloud data management systems and formats are heavily specialized and targeted solely towards visualization purposes and fail to address the diverse needs of progressive point cloud workflows like for example semantic segmentation using machine learning. We therefore propose a distributed data infrastructure for dynamic point cloud data management that can support interactive real-time visualization at scale while simultaneously serving as a platform for analytical tasks. By introducing random data distribution, we show that simple query fragmentation and efficient and effective parallelism at scale are possible. At the same time, arbitrary queries in space and time can be efficiently run over the infrastructure including query semantics which returns only a random sample of the query results or preferred points based on an importance dimension calculated, for example, from a local point density information as commonly done in point cloud visualization. To cope with the unknown amount of user-specific attributes and to support even multiple ways of deciding the importance of a given point (ground point removal, coverage of space, random subset) the system is designed to support all of them transparently as multidimensional range queries backed by spatial indices.

Keywords. point cloud, data management, distributed

1 Introduction

Point clouds play a vital role in the broader field of geospatial information science. They represent a flexible representation of geospatial data at scale with few semantics constraints. Point clouds occur naturally in laser scanning, where the distance between a sensor and an object is estimated from the time of flight of a laser ray. Furthermore, they appear in photogrammetry contexts, representing the results of simultaneous localization and mapping (SLAM) and photogrammetric 3D reconstruction (Wehr and Lohr, 1999; Shan and Toth, 2018; Gamba, 2020).

In recent years, sensors capable of generating point cloud data have become ubiquitous, featured most prominently in mobile phones, cars and drones. This widespread dissemination of sensors deployed on different platforms results in an increasing amount of point cloud data to manage and process. New use cases arise with these platforms, and the ever-increasing spatio-temporal resolution of such data displays novel requirements and challenges existing solutions to store, process and visualize such data.

Point clouds can contain from a few thousand points for example indoor scenes from ScanNet up to trillions of points like for example the airborne laser scans of the Netherlands (Dai et al., 2017). Furthermore, point clouds can be sampled from complex geometric data, including triangle meshes and polytopes, and - in contrast to simple feature geometry - can even represent faithfully the interior of polytopes, such as in 3D scans of the human body with CT.

Point clouds are increasing in **volume**, due to the recent advancements in sensor availability, resolution, and sampling rate implying an ever-increasing **velocity** at which points are generated. In addition, the missing semantic information makes point clouds flexible and useful in observation and data integration but at the cost of a huge **variety** towards data interpretation. These characteristics reveal that point cloud data can be viewed as big data and as such shares common associated challenges (Lee and Kang, 2015).

Surprisingly, point cloud support in current geographic information systems remains surprisingly limited, and scalability is a major difficulty for existing solutions. We want to propose a reliable, scalable and simple architecture to overcome this limitation as we see a demand for a point cloud data management system that can ingest, fuse, update, analyze and visualize data in real-time.

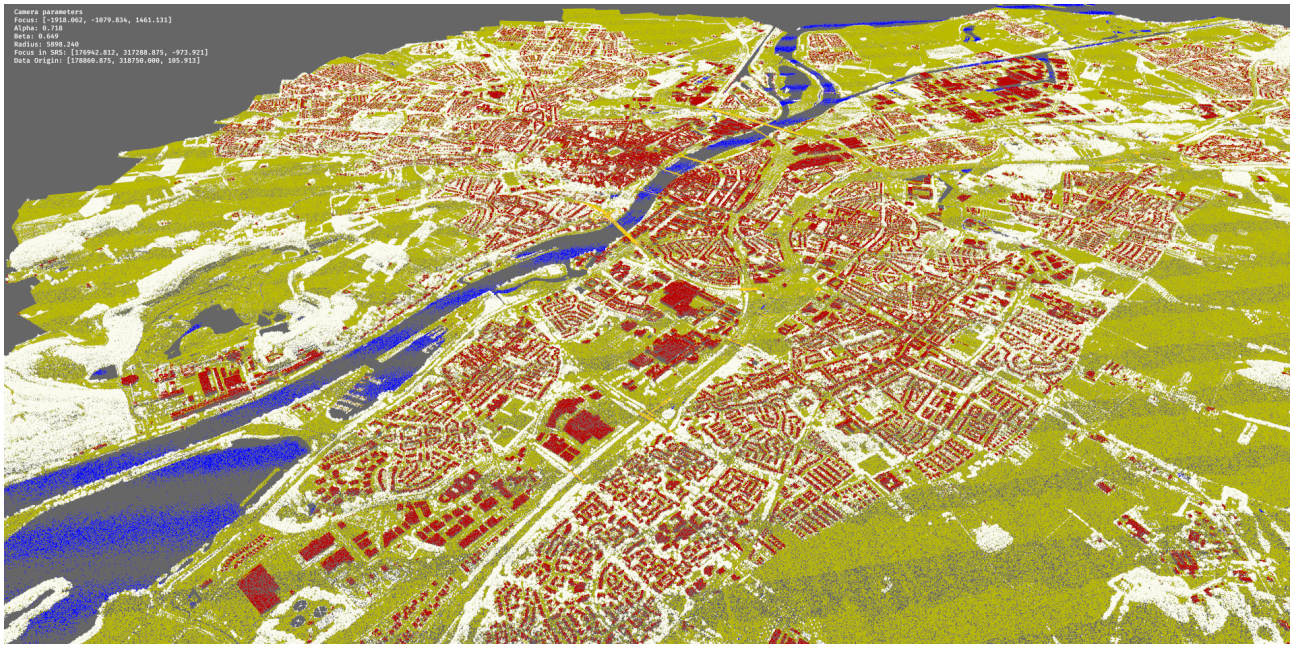


Figure 1. Visualization of an extract of a point cloud from terrestrial laser scanning in Maastricht (AHN3), colored based on the LAS classification attribute.

This paper presents a distributed point cloud data management system that offers high-performance parallel range query capabilities over spatial, temporal, and thematic dimensions. This opens up the possibility for interactive visualization while simultaneously serving as a source for long-running analytical tasks that operate on random samples, like for example, point cloud segmentation or classification.

The main contributions of this paper are

- a *definition of a simple and flexible distributed middleware* for point cloud processing
- a *data distribution and organization scheme* for effective range query execution
- a *thorough evaluation of data distribution schemes* for point cloud data and their impact on typical queries from visualization and point cloud analysis
- a *definition of a randomized query semantics* in which query results are expected to be uniform samples of the entire result set
- extensive *experiments on real-world datasets* for national-scale interactive point cloud computing.

The remainder of the paper is structured as follows. Section 2 presents the state-of-the-art from relevant literature. Section 3 outlines the conceptual foundations. Section 4 introduces the architecture of the point cloud management system. Section 5 evaluates selected use cases and applications of the system. Finally, Section 6 concludes the paper by addressing open issues and future work.

2 Related Work

Current geographic information systems use data models in which features with geometries are commonly represented as individual records. These models fail to scale well to billions of records stored for example in a database table or key-value store (van Oosterom et al., 2015). Hence the peculiarities of point cloud data brought forth specialized solutions to manage, process, analyze and visualize such data (Schütz, 2016; Butler et al., 2021).

Point cloud data resides commonly in text-based data formats, suitable for initial processing and transmission (Vo et al., 2016). Widely used storage formats are the LAS file format and the compressed LASzip incarnation (Isenburg, 2013). They build upon a standardized point and file format specification[1], benefitting exchanging and merging data from different sources. Unfortunately, the format has many drawbacks. For example, it is not straightforward to extend the point format with custom attributes. Further, having the points laid out as a continuous record in memory fragments the attribute values across the whole file storage space, leading to inefficient lookup of the entire file to access only one attribute. Additionally, in the compressed case, one needs to decompress from the beginning of a chunk to the point of interest to extract its information, rendering it rather unusable for random lookup.

While these shortcomings might not be relevant for time-insensitive offline algorithms and batch processing, they certainly matter when querying the data repeatedly with high frequency, a common characteristic of neighborhood analysis or visualization queries. For this, the data is of-

[1] https://www.ogc.org/standard/las

ten reorganized and converted to data structures that use spatial indexes like space partitioning trees (e.g., octree, k-d-tree, R-tree), space-filling curves (e.g., Hilbert curve, Z-index), locality-sensitive hashing (LSH) to accelerate spatial range queries (Meijers, 2022; van Oosterom et al., 2022).

Current state-of-the-art file-based solutions for rendering point cloud data are internally organized utilizing an octree, like for example, Cloud Optimized Point Cloud (COPC)[2], Entwine Point Tile (EPT)[3], implicit tiling in 3D Tiles[4] or Potree (Schütz et al., 2020). These structures all incorporate some level of detail (LOD/cLOD) or level of importance (LOI/cLOI), generally in discrete form, which is a widely recognized necessity to facilitate the visualization of extensive point cloud data given rendering and data transmission constraints (Butler et al., 2021).

Relational spatial database management systems (RDBMS) provide indexing and decent query performance for small result sets but fail to scale well, even when using specialized point cloud data extensions (Lokugam Hewage et al., 2022). One reason is that central components are commonly coordinating data organization in RDBMS. Another is the resource use and maintenance cost for indices, which can require up to 30% of additional data space (Yu and Sarwat, 2017; Wang et al., 2023). In terms of performance and scalability, however, a completely decoupled operation of data nodes would be preferable, which is called a shared-nothing architecture, even more so if data size and traffic volume changes are expected (Lokugam Hewage et al., 2022).

Preparing point cloud data for visualization purposes generally involves extensive preprocessing and finally results in a custom dedicated data representation unsuitable for general-purpose analysis. A recent attempt to serve large point clouds over the internet for interactive visualization resulted in a 25-fold storage amplification and a throughput of about 65,000 points per second in the preprocessing, respectively 100,000 points per second in consecutive HTTP queries (Meijers, 2022). For the whole Netherlands (AHN3) with about 700 billion points, this would mean a preprocessing time of more than 120 days.

2.1 Visualization of Point Clouds

Visualizing a set of 3D points on the screen is a comparably simple task with current hardware-accelerated 3D graphics. Given the 3D coordinates of the points, camera projection information, and the pose of the camera in the 3D world, it boils down to projecting all points to the screen plane, ordering the visible points on the screen plane by Z coordinate and drawing those points (e.g., in a color if the point cloud has attributes). In addition, Eye-Dome Lighting can be applied to avoid a 3D space inference of first-order geometry like surface normals limiting computation to exactly the visible points.

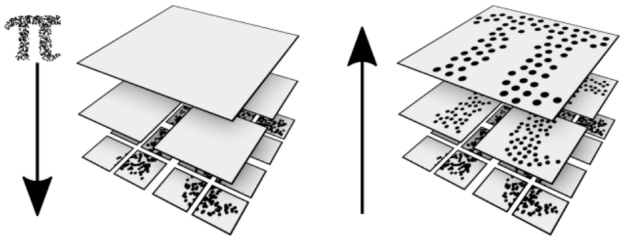
Despite hardware acceleration, however, the amount of points that can be rendered in real-time remains limited and the data management system is required to provide a representation of the visible point cloud data within a specified point budget of a few million points. Classical techniques to achieve this include clipping and view frustum culling, where data organization is used to materialize points within the field of view of the camera, importance sampling in which points are ordered by importance for visualization to further reduce the number of points and algorithms to retrieve lower density point cloud representations farther away from the camera.

In sampling, one tries to combat the overplotting problem that for most reasonable camera poses for huge point clouds, a multitude of points will be rendered per screen pixel to drop the information of all but the front pixel. Sampling has been extensively studied in computer graphics and multiple efficient strategies can be applied (Bloom, 1970; Schütz et al., 2020).

In view frustum culling, one tries to efficiently avoid the consideration of the majority of the points during the rendering process. Therefore, spatial partitioning and indexing, such as quadtrees, octrees, or bounding volume hierarchies, are often applied. In this case, a certain amount of points is represented by a 3D box. In the rendering stage, the system first determines, which boxes could be visible in the view frustum in order to download the points that these boxes represent to the drawing pipeline.

2.2 Potree: A State-of-the-Art Massive Point Cloud Visualization System

One of the most influential point cloud visualization systems is Potree. It provides a nice overview of a possible implementation of a visualization system for massive point clouds by combining a multilayer partitioning approach (in order not to load the whole of the partitioning information over the Internet) with a variant of Poisson-disk sampling to ensure real-time visualization of massive point clouds with limited point budgets (Schütz et al., 2020).



(a) Chunking **(b)** Bottom-Up Subsampling

Figure 2. Potree data structure generation (Schütz et al., 2020)

Poetry employs an octree-inspired layered data structure with an additive scheme (Figure 2). It is generated by first chunking the data into small patches that are subsequently

[2] https://copc.io/
[3] https://entwine.io/en/latest/entwine-point-tile.html
[4] https://docs.ogc.org/cs/22-025r4/22-025r4.html

used as tree-leaf nodes in the initial creation of the index structure. In the second step, inner tree nodes are recursively populated with points sampled from their children.

In order to get visually good results from spatially or temporally ordered 3D point cloud acquisitions, a certain amount of randomization to the order is favorable at the expense of increasing the computational burden. In their work Schütz et al. (2020) show that Poison Sampling creates superior visualization results at a higher computational expense, while random sampling offers a good tradeoff between visual quality and computational resource use.

The resulting custom hierarchical spatial data organization is optimized to download chunks in varying (discrete) levels of detail to become efficient. As this approach generates huge hierarchies, the hierarchies themselves are decomposed to avoid the requirement to download the whole hierarchy before visualization starts.

It is worth noting that the backend implementation of the Potree system is passive. That is, data is organized into many small files or chunks in a file that can be retrieved over the internet by clients if and only if they are needed as individual HTTP requests. In this way, a valid backend is just a web server providing files from the hierarchy generated by the Potree converter. For massive point clouds, classical caching mechanisms can provide very high speed for boxes queried multiple times (e.g., when the field of view is only slightly changing) while unused data resides on disk. On the contrary, this system's performance is also based on the ability of the storage infrastructure to provide the required data quickly, which can become a challenge.

3 Concept

The current state-of-the-art solutions display several drawbacks and limitations from the viewpoint of a general-purpose analysis ready point cloud data management system. For one they are rather slow due to the exhaustive preprocessing. Further, this preprocessing is designed solely for visualization purposes only and relies on specialized data structures, indices and formats which can generally only be used with dedicated software libraries. Querying these data sources often requires knowledge about the data structure and format in order to efficiently access only the data parts of interest for a certain query to resolve.

To address these shortcomings we propose a system that provides a simple RESTful API for generic range queries over the spatial, temporal, and continuous hashing dimensions and materialized continuous importance dimensions, a self-describing transmission format and a simple data distribution scheme to leverage parallel computing and cloud computing. The proposed system shall close a gap in current geospatial data management systems such as PostgreSQL with PostGIS or Apache Sedona, which are difficult to use for point cloud management. Key differences with existing systems are

- the linear scalability of the system with the data,
- the support of incremental query resolution
- a general support for sampling from the result set
- a database infrastructure not limited to efficient visualization

3.1 Continuous Level of Importance

The problem of sampling reasonable amounts of points from a geospatial dataset for visualization purposes is similar to a ubiquitous concept in spatial data modeling known as Level of Detail (LoD). In traditional maps, the amount of information visible (e.g., labels, types of features) depends on a discrete zoom level. For raster maps served over the internet, one generally creates rasters for each zoom level where higher levels are downsampled versions. In Potree, for example, the points are distributed among the hierarchically organized index tree nodes such that each tree node holds a similar amount of points constituting a representative spatial sample within the tree node's spatial bounds (Schütz, 2016). This makes it suitable for visualizing the point cloud on a discrete level of details corresponding to the depth in the spatial indexing graph employed.

However, this discrete nature is not at all required or welcome as it can provide artifacts and flickering interaction effects when suddenly the number of visible points and the fidelity of the geometry increase or decrease during the interaction. Hence, the concept of Continuous Level of Details (CLoD) has been introduced in which every point is associated with an importance number such that the set of all points with an importance number smaller than a given threshold τ can be considered a good sample of the data. Due to the interpretation of the newly associated value as importance, this approach is sometimes known as Level of Importance (LoI) as well (van Oosterom et al., 2022).

If an architecture for point cloud management and computing aims at CLoD support, it is interesting to note that the traditional discrete Levels of Detail can be seen as a natural instance of CLoD in which only integer values for the (continuous) importance value are utilized. Further, it is important to realize that the visualization of point clouds can then be interpreted as a range query in 4D space, where three geometric representations are augmented with an artificial dimension representing the spatial importance of a point during sampling.

3.2 Range Queries with Importance

To demonstrate the system we consider range queries, one of the most fundamental query types of database systems. In range queries, a query is formulated as a range object

which is used to find all intersecting objects. Given for example a 3D axis-aligned box, the goal is to compute the set of all points within this box. In this case, the box, respectively the ranges of the query object, are orthogonal to the data dimensions and therefore constitute an orthogonal range searching problem. With importance added as a dimension, a range query can be formulated as a function of the range bounds like

$$\mathrm{Range}(x_{\min}, y_{\min}, z_{\min}, i_{\min}, x_{\max}, y_{\max}, z_{\max}, i_{\max})$$

For datatypes that support infinity, this naturally encompasses range queries with infinite axes such as $[-\infty, \infty]$. Based on this we defined three semantic variations of these queries:

- a **full** variant, representative of the classical range query getting all information
- a p-**sampling** variant, where the query result is a uniform sample of the full result
- and a **facet-sampling** variant, where the query result is a pseudo-random sample in a way such that multiple facet queries can retrieve the full dataset

In order to implement p-sampling, we decided to add an artificial floating point attribute i with values from [0,1] such that a p-sampling query can be expressed as a full query with a limited sub-interval of this novel attribute by translating it to $i_{\min} = 0$ and $i_{\max} = p$. We call this novel attribute **continuous hash**. This approach does not only provide p-sampling queries but can be used as well to implement refinement queries through facet-sampling. As a means to facilitate these query semantics, we design our system with a generic backend implementation optimized for range queries over the spatial, temporal and importance dimensions.

Databases traditionally implement spatial indexing based on R-Trees or Quad-Trees among others (Wang et al., 2023). Such space partitioning trees or bounding volume hierarchies, in general, recursively partition space into smaller and smaller bounding boxes following certain optimization criteria. However, what is common to all of them is that the processing of range queries is simple as all trees fulfill the condition that the bounds of parent tree nodes fully contain all its children's bounds. The indexed objects, in our case points or point batches, are typically kept in the leaf nodes or alternatively just referenced by pointers from there.

In such structures, a tree traversal provides a highly performant way of finding relevant candidates intersecting given bounds. For these candidates, a final check against the query bounds provides exactness. An interesting implication of this approach is that a parallel implementation of range query resolution is possible by first discovering relevant candidates and then distributing the final checks against the query bounds.

3.3 Distributed and Parallel Query Processing Through Query Partitioning

In traditional relational databases, parallel query processing is not directly possible as most queries are translated into a sequential scan over one of the involved columns which can use an index to avoid scanning useless parts of the keyspace based on knowledge derived from query predicates.

One principled way of introducing parallelism to such systems is the approach in which a specified main query is not being executed by a single database instance. Instead, a middleware translates a single query into a family of subqueries, which are then executed by a set of database instances in a distributed computing setting. Each subquery provides a partial result and the middleware can resemble the accurate and complete query result by aggregating the subquery information in the context of the given main query (Kossmann, 2000).

Note that the subqueries need not be a faithful decomposition of the original query in this setting. One can design systems in which the subqueries can have partial overlap with each other (e.g., identical results are returned from two different subqueries) or in general provide query results that serve as a candidate set for which the original query predicates still need to be applied in a filter and refine pattern. Overlap can be especially powerful to achieve query-level fault tolerance. In this setting, multiple different nodes are asked for the same data. This can give a wall-clock performance boost because the fastest results can be utilized and query runtime does not depend on the slowest contributor. At the same time, it can help avoid incomplete query results if parts of the infrastructure are unavailable.

Since a coordinator component is often responsible for instantiating and merging micro queries, a caching mechanism (in-memory or on-disk) should be implemented on the level of the coordinator component to facilitate cache-aware query routing. As the cache can be essential to the interactivity of the overall system, this induces an open question of how to extend this architecture towards distributed caching in which the same mini-query results can be used for different client requests handled by different coordinator components. We envision a gossip-style cache information update as a good choice in this direction but leave this area for future research due to its complexity.

4 System Architecture

The main goal is to have a data infrastructure for dynamic point cloud data that can support interactive real-time visualization at scale while simultaneously serving as a platform for analytical tasks. To facilitate this, we propose a distributed system where the points are augmented with a random continuous hash value that is utilized to create a higher dimensional R*-tree. The architecture broadly follows client-server and microservices principles where the

service can run as a coordinator, worker, or both simultaneously (Figure 3). An open-source implementation written in the Rust programming language[5] relying on Apache Arrow[6] for portable in-memory data representation is publicly available.

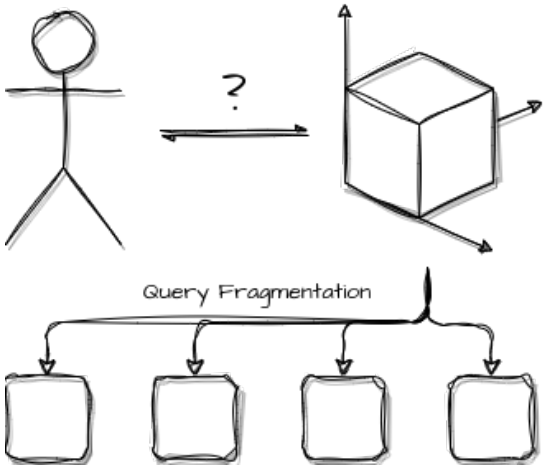


Figure 3. Overview of the distributed point cloud system architecture depicting a user's interest in a range object which is resolved through query fragmentation on multiple nodes.

4.1 Middleware Design

The middleware basically exposes a simple API to orchestrate and manage point cloud data and is also responsible for query routing. Query routing is the procedure of sending query requests to worker components. The system can be instantiated with exhaustive routing in which queries are decomposed into slices handed to the worker nodes of the system. Gossip query routing in which a query is emitted to every worker with a fixed probability p only, or knowledge-driven routing in which queries are routed incorporating knowledge about the data partitioning and distribution scheme of the system are envisioned.

For orchestration purposes, an endpoint exists where instances listen to requests from other instances to register themselves with their URL. Like this, a distributed system can be created for example by setting up an instance as a coordinator with a fixed address and then starting several instances as workers who make a request to the coordinator for registration. Such a simple topology depicted is further explored in Section 5 as a means to evaluate the system. In this setup, the coordinator is prone to become the bottleneck and single point of failure, though this can be easily mitigated by using several coordinator nodes behind a load balancer analoguous RAID 1. This analogy is as well applicable to the worker nodes which naturally represent a RAID 0. Through these redundancies, our system offers high flexibility, scalability and robustness against node failures. Things we like to explore further in the future are handling the coordination client side, assessing different topologies like peer-to-peer, and other distributed systems approaches.

A node setup as a coordinator is basically responsible for keeping track of existing worker nodes, distributing requests among them and joining their returned results. Upon a load request, containing for example several LAZ files, the coordinator adds a range to each request and forwards it to the workers. These ranges specify equally sized fractions for each node. Like this, the points are distributed randomly amongst the workers based on evaluating a random number generated for each point for inclusion in the range. A query request issued by the user specifies interest in data and associated query semantics as requested by the client application. Based on this information, the coordinator might decompose the query into multiple smaller queries by slicing the query. With query slicing, we mean a query segmentation strategy in which constraints on the query are formulated in order to have a smaller result set per query while the original query is then computed from the overall result set of all query slices. This computation can be a simple concatenation, a set union, or a union followed by a refinement stage if the subqueries return more data than the query actually requires.

A worker node, on the other hand, is responsible for actually executing the load, index and query request by processing the data. Upon a load request, the workers load a fraction of points provided their associated randomly generated number $r \in [0,1)$ is in the interval of the range $[from_k, to_k)$ specified in the request for the given worker k. Like this, the entire data is split among the workers as equally sized, nonoverlapping, normally distributed samples. Subsequently, each worker can independently construct an in-memory R*-tree over the point cloud data fraction. While other spatial indices would be possible, the R*-tree is a widely accepted method for range query processing. For static datasets, this can be done efficiently by bulk loading using the Sort-Tile-Recursive (STR) or Overlap Minimizing Top-Down (OTM) algorithm (Leutenegger et al., 1997; Lee and Lee, 2003). Bulk loading ensures high index quality for analytics workloads in which the point cloud and the associated spatial indices are not updated.

4.2 Importance Augmentation and Partitioning

The baseline level of importance is given by generating a uniform random number between 0 and 1 for each and every point as a continuous hash. Then, a query interval $[a,b]$ on the importance translates to a uniform sample containing a point with a probability of $b-a$. More elaborated ways of assigning levels of importance, like for example blue noise sampling, Poisson sampling or dart throwing can improve visual quality though they are not elaborated in this work since the improvements are too marginal with respect to added complexity (Schütz et al., 2020).

[5] https://www.rust-lang.org/
[6] https://arrow.apache.org/

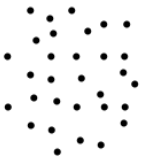
Adding a materialized continuous hash as a level of importance enables us to process 4D range queries, which essentially represent a random sample of the points in the given spatial bounds. Compared to other solutions where the importance lives implicitly in the hierarchy level of the tree, this gives us some advantages. For example, it is possible to query multiple equal samples by shifting the importance query range across the whole importance range. Through this, we can conveniently and efficiently support probabilistic analytical approaches or create multiple distinct samples for deep learning approaches like PointNet. Another advantage is the opportunity for iterative query refinement, as we can query for a consecutive importance range without getting the same points.

As a baseline data distribution strategy, we consider uniform random assignment of points to workers. While other data distribution schemes for example such modeled after a spatial ring structure based on a space-filling curve are potentially interesting to investigate, the random distribution strategy displays some pleasant characteristics for distributed range queries. For once it is not necessary to worry about query decomposition as one can push down the query as is on multiple workers and it behaves equivalent to only having a single node. Additionally, the data and result set are similar on each node, hence all nodes have a similar workload which results in effective and efficient parallelization.

4.3 Data Format

Point clouds can be conceptualized as a set of points P where p_i is in $\mathbb{R}^3$ or more generic, each point consists of a set of attribute values as shown in Figure 4. From this perspective, using a tabular representation for point cloud data follows naturally.

$$P = \{p_1, \ldots, p_N \mid p_i \in \mathbb{R}^3\} \qquad p_i = \{a_{1i}, \ldots, a_{mi}\}$$



a_1	...	a_m
a_{11}	...	a_{m1}
...	...	...
a_{1i}	...	a_{mn}

Figure 4. Conceptualization of point clouds: A set of points where each point is defined by a set of attribute values.

To represent a point cloud in our system we rely on Apache Arrow[7], a modern data format optimized for analytical workloads on two-dimensional data. The columnar data layout offers efficient processing of range queries over multiple dimensions, respectively columns. Further benefits are the low serialization and deserialization overhead when transmitting data and interoperability facilitated by its self-describing format and memory layout specification, which is supported by a growing ecosystem of software and applications.

5 Applications and Experiments

In this section, we present the results of selected experiments to exemplify the performance characteristics of the system in terms of query performance and scalability, for which several different configurations are evaluated and compared against similar experiments from existing related work where applicable. A well-researched field for this represents the visualization of governmental aerial laser scanning (ALS). Many more fields of application, like urban digital twins, medicine (replacement of 2D images), autonomous driving and robotics, are considered for future evaluation. Yet, the performance characteristics are predicted to be similar for these use cases given the underlying system is working in the same fashion.

5.1 Dataset

The Actueel Hoogtebestand Nederland (AHN) represents an ALS point cloud of the whole of the Netherlands with a little more than 800 billion points in its third version [8]. The dataset is delivered in 1374 LAZ files in which each point is modeled in a local integral coordinate system whose relation to the global scene is modeled by a scale vector and an offset vector. This dataset is essentially two-dimensional in the sense that the range of the z coordinate is very small compared to the other coordinates. For each experiment, we extracted a continuous subset of the whole dataset.

5.2 Loading and Indexing

Preprocessing entails all the steps necessary to make point cloud data available for querying. The major tasks involved in this are loading the data from its source, applying some partitioning and building up the index. In our case, we read the LAZ files, partition them into a fixed-size grid and then create an R*-tree. In Table 1 we measure the preprocessing performance in terms of throughput (points per time) as well as storage and memory footprint. The input for this experiment consists of a single LAZ file of about 743 megabytes containing about 150 million points and was run on a workstation with 16 cores.

The results show that our system has quite a high throughput while keeping the memory footprint at an acceptable level compared to Potree and PCServe. The performance for the in-memory case is unsurprisingly noticeably higher, loading the whole AHN3 with about 700 billion points would take about 6.7 hours. Even though our solution in its current state does not offer true out-of-core processing, the distributed nature of our system

[7] https://arrow.apache.org/

[8] https://www.ahn.nl/

Table 1. Load and index creation performance metrics comparison.

File	C_69AZ1.LAZ (743MB, 149 676 342 Points)						
Preprocessing	Load		Index		Total		
	Time (s)	Size (MB)	Time (s)	Size (MB)	Time (s)	Size (MB)	Throughput (Points/s)
Potree	6.2		17.9		24.0	4 141	6 230 284
PCServe	792.0	15 138	318.0	3 362	1110.0	18 500	134 844
Ours (in-memory)	4.1	8 558	1.0	1 267	5.1	9 715	29 114 246
Ours (on-disk)	11.5	7 442	1.9	1 267	13.3	8 709	11 235 275

can make up for it by aggregating the memory capacity of many nodes. Compared to benchmark results of point cloud data management systems, our system outperforms all existing systems in terms of loading and indexing except such based on Oracle Exadata (van Oosterom et al., 2015; Lokugam Hewage et al., 2022). Taking the difference of the core count 16 vs 48 on Oracle Exadata into account and our system demonstrating efficient use of parallelism we expect it to outperform these solutions as well on comparable hardware.

5.3 Query Performance

To evaluate our approach's effectiveness we conducted several experiments in various system configurations to give justifications for the system's behavior for real-world use cases and under scaling loads. First off we look at range queries for visualization purposes where small result sets are expected. Two query modalities are chosen to test the two extreme cases for visualization, on one hand, get an overview at a low zoom level on the other hand a detailed small extent when zoomed in. The results in table 2 are from two setups, once a single node and a cluster with eight workers on which queries of both modalities, highly sampled queries over the whole data extent and queries solely bounded by a spatial extent were executed. As a baseline PCServe is included.

In a single node setup with about 150 million points, our system outperforms the baseline, even in the case we spill the data to disk and only hold the index in memory. Even in a distributed setup with about 4.5 billion points, a small sample of the data representative of the whole dataset is queried within 600 ms over HTTP.

5.4 Scaleability

The scalability is evaluated on a cluster of personal computers connected over ethernet in a local area network. Each node is configured with an AMD Ryzen 7 5800X 8-Core CPU, 32GB RAM and 250GB SSD. With Docker Swarm, the services are managed and distributed among the nodes. The load time remains constant as each node reads the data from the disk, parses the content and then discards points not matching the fraction assigned. Loading by message passing to only decode the points once and assign them to each node was tested though much slower in this setup due to the interconnection bandwidth.

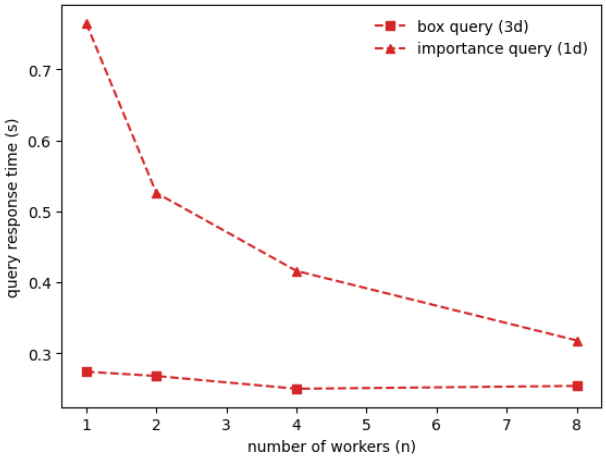


Figure 5. Worker scalability and query performance of selected queries executed on a system with 1, 2, 4 and 8 workers.

The results in Figure 5 show the query execution times on a setup with 1, 2, 4 and 8 workers. The importance query displays improvements from scaling. This is due to the fact that the importance is randomly distributed across the data and the query boils down to a scan hence on multiple nodes the amount of data to scan becomes less. The box query returns the points within spatial bounds. This is rather performant even on a single node since the spatial locality in the data from the sensor is preserved. This further shows that the system does not display performance regressions when scaling, indicating its low overhead cost.

In terms of query scalability, Figure 6 illustrates how queries scale with regard to the result size. For this queries with different result set sizes were executed against 8 workers with a total of about 2.5 billion points. The measurements are done on three levels, the worker service node, the coordinator service node and the client API respectively. Loading takes about 138 seconds and indexing takes about 4 seconds which equates to a throughput of

Table 2. Range query performance comparison.

	Dataset size (M Points)	Query type (box,sampling,mixed)	Request duration (ms)	Result size (Points)
PCServe			598	38 981
Ours (in-memory, single node)	150	box	60	58 616
	150	sampling	166	74 343
Ours (on-disk, single node)	150	mixed	196	74 926
Ours (in-memory, 8 workers)	4 500	box	166	59 259
	4 500	sampling	599	45 304

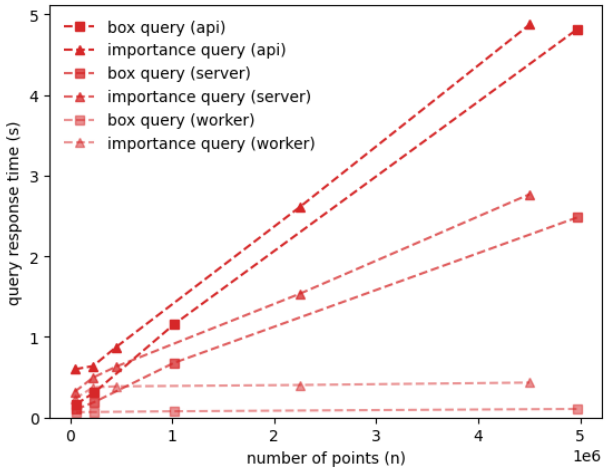


Figure 6. Query scalability in terms of the size of the result set returned.

about 17 million points per second. The results show that the query performance scales linearly to the result set with a throughput of over 1 million points per second. The box queries represent a two-dimensional spatial extent whereas the importance queries filter by importance over the whole extent of the dataset loaded. The formers tend to be faster as the resulting query extent is more compact in terms of chunks to process to resolve it.

From the measurements at different stages in the system, one can see that the major bottleneck is the data transmission over ethernet. At the worker level, the query execution time only slightly increases for the result sets up to 5 million points. At the server level, the increase is substantial, even more, measured at the client side. This indicates that distributing large queries leads to degraded performance compared to a local single-node setup. Nevertheless, for queries with a small result set, commonly used for visualization purposes, the overhead of a distributed setup is much lower and can to some extent be compensated by the faster query performance.

6 Conclusions

With this paper, we have proposed a simple, fault-tolerant, linearly scalable in-memory infrastructure based on microservices that can be used to interactively visualize and query massive point cloud data. To the best of our knowledge, this is the first distributed data infrastructure designed around the nature of point clouds and comparisons with established systems demonstrate the usefulness of the approach.

We showed that this architecture offers several benefits like even data and query distribution in terms of runtime, resource usage and expected result set. Further, the simple and effective architecture is equally suitable for single-node, multiple instances as well as distributed scenarios, since it is possible to set up various topologies as instances can be simultaneously coordinator and/or worker. Like this, we can create a robust and fault-tolerant setup for querying arbitrary range queries over spatial and importance dimensions through a simple API.

For future work, we want to investigate a point cloud processing mechanism in which we further exploit one of the key advantages of the proposed architecture, namely, that the data representation in main memory is simple and that data access can be immediate and not through an API like a database cursor. We aim to show that integration with numpy, tensorflow, or pytorch becomes easy and we want to develop further the given ideas into a more mature point cloud data infrastructure by analyzing the domain demand, designing new queries and query semantics, and finally providing the tool to use to the point cloud research community.

Data and Software Availability

Research code supporting this publication is available in the GitHub repository https://github.com/tum-bgd/2024-AGILE-RandomDataDistribution. Please follow the instructions in the file README.md in the repository to set up the system.

References

Bloom, B. H.: Space/time trade-offs in hash coding with allowable errors, Communications of the ACM, 13, 422–426, https://doi.org/10.1145/362686.362692, 1970.

Butler, H., Chambers, B., Hartzell, P., and Glennie, C.: PDAL: An open source library for the processing and analysis of point clouds, 148, 104 680, https://doi.org/10.1016/j.cageo.2020.104680, 2021.

Dai, A., Chang, A. X., Savva, M., Halber, M., Funkhouser, T., and Niessner, M.: ScanNet: Richly-Annotated 3D Reconstructions of Indoor Scenes, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2017.

Gamba, J.: Radar Signal Processing for Autonomous Driving, Signals and Communication Technology, Springer Singapore, https://doi.org/10.1007/978-981-13-9193-4, 2020.

Isenburg, M.: LASzip: lossless compression of LiDAR data, Photogrammetric engineering and remote sensing, 79, 209–217, 2013.

Kossmann, D.: The State of the Art in Distributed Query Processing, ACM Comput. Surv., 32, 422–469, https://doi.org/10.1145/371578.371598, 2000.

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

Lee, T. and Lee, S.: OMT: Overlap Minimizing Top-down Bulk Loading Algorithm for R-tree., in: CAISE Short paper proceedings, vol. 74, pp. 69–72, 2003.

Leutenegger, S., Lopez, M., and Edgington, J.: STR: a simple and efficient algorithm for R-tree packing, in: Proceedings 13th International Conference on Data Engineering, pp. 497–506, IEEE Comput. Soc. Press, https://doi.org/10.1109/ICDE.1997.582015, 1997.

Lokugam Hewage, C. N., Laefer, D. F., Vo, A.-V., Le-Khac, N.-A., and Bertolotto, M.: Scalability and Performance of LiDAR Point Cloud Data Management Systems: A State-of-the-Art Review, Remote Sensing, 14, 5277, https://doi.org/10.3390/rs14205277, 2022.

Meijers, M.: PCSERVE – ND-POINTCLOUDS RETRIEVAL OVER THE WEB, ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences, X-4/W2-2022, 193–200, https://doi.org/10.5194/isprs-annals-X-4-W2-2022-193-2022, 2022.

Schütz, M.: Potree: Rendering large point clouds in web browsers, https://publik.tuwien.ac.at/files/publik_252607.pdf, 2016.

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

Shan, J. and Toth, C. K., eds.: Topographic Laser Ranging and Scanning: Principles and Processing, CRC Press, 2 edn., https://doi.org/10.1201/9781315154381, 2018.

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

van Oosterom, P., van Oosterom, S., Liu, H., Thompson, R., Meijers, M., and Verbree, E.: Organizing and visualizing point clouds with continuous levels of detail, 194, 119–131, https://doi.org/10.1016/j.isprsjprs.2022.10.004, 2022.

Vo, A.-V., Laefer, D. F., and Bertolotto, M.: Airborne laser scanning data storage and indexing: state-of-the-art review, 37, 6187–6204, https://doi.org/10.1080/01431161.2016.1256511, 2016.

Wang, C., Yu, J., and Zhao, Z.: GLIN: A (G)eneric (L)earned (In)dexing Mechanism for Complex Geometries, in: Proceedings of the 11th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data, pp. 1–12, ACM, https://doi.org/10.1145/3615833.3628590, 2023.

Wehr, A. and Lohr, U.: Airborne laser scanning—an introduction and overview, 54, 68–82, https://doi.org/10.1016/S0924-2716(99)00011-8, 1999.

Yu, J. and Sarwat, M.: Indexing the pickup and drop-off locations of NYC taxi trips in PostgreSQL–lessons from the road, in: Advances in Spatial and Temporal Databases: 15th International Symposium, SSTD 2017, Arlington, VA, USA, August 21–23, 2017, Proceedings 15, pp. 145–162, Springer, 2017.