PCSERVE-ND-Pointclouds Retrieval over the Web

Meijers, M.

DOI
10.5194/isprs-annals-X-4-W2-2022-193-2022

Publication date
2022

Document Version
Final published version

Published in
ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences

ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences, Volume X-4/W2-2022
17th 3D GeoInfo Conference, 19–21 October 2022, Sydney, Australia

PCSERVE – ND-POINTCLOUDS RETRIEVAL OVER THE WEB

Martijn Meijers

Delft University of Technology, Faculty of Architecture and the Built Environment, GIS-technology – b.m.meijers@tudelft.nl

Commission IV, WG IV/9

KEY WORDS: Point Clouds, Space Filling Curve, Web Based Visualisation, Web Service, Database Management System

ABSTRACT:

We investigate how PCServe, a web service for disseminating massive point clouds, performs for read-only access (i.e. a visualization application). PCServe is backed by a database model based on Space Filling Curves. By adding a virtual hierarchy of blocks to the database, we can support different visualization applications for retrieval of point cloud data over the web without having to store the data multiple times. This makes expressive access to point clouds over the web possible. We investigate the amount of processing that is needed to create the database model and how well PCServe handles requests from the visualization application. Some suggestions are provided how the current approach can be improved.

1. INTRODUCTION

Point clouds are rich representations of the real world, measured accurately, and stemming from different sources, like LiDAR, photogrammetry or bathymetric echo sounding techniques. Due to their massive sizes, interactive visualization of point cloud data remains a challenge. Point clouds are preprocessed in specific data structures for web-based point cloud visualization.

In recent years, a variety of protocols have been created for point cloud retrieval, each serving different applications (e.g. 3D Tiles by Cesium, Indexed 3D Scene Layers, I3S, by Esri, Cloud Optimized Point Cloud, COPC, by Hobu, Inc., Potree by the Potree developers). Different visualization clients (like Potree and Cesium) expect different file formats (protocols). When using a file based organisation and multiple clients that need to be served, for each application a different file organisation is necessary, requiring the point cloud to be stored multiple times server side, which can be impractical given the volume of the point cloud data. Hence, we propose PCServe, a web service for retrieval of point clouds supporting different existing protocols (Potree, 3D Tiles), to be able to serve point data to different interactive visualization applications. PCServe relies on how we organise point clouds inside a database system with the help of a Space Filling Curve (SFC). Within the database system, we have an efficient mechanism for determining arbitrary subsets of the massive point clouds (based on nD-polytopes, i.e. we can express generic nD query geometry and obtain the result using the SFC based organisation).

In this paper we show that adding one additional value that encodes the level of detail per point allows us to adjust the density of the requested subset targetted at an application. We also give an algorithm for how we can map these continous level of detail values to a virtual discrete block based organisation (octree). Moreover, we demonstrate using PCServe how a visualisation application, like PoTree, can subsequently fetch point cloud data over the web from our database organisation based on SFCs. Figure 1 shows Potree with PCServe having delivered the points from the database.

The remainder of this paper is organized as follows: Section 2 reviews related work on managing and disseminating point clouds via web services. Section 3 gives details on our database organisation, how we obtain the virtual block based structure and how it will be used for PCServe, the nD-point cloud web service. Section 4 reports on results we obtained with our implementation. In Section 5, a discussion follows on the obtained results. Section 6 gives a conclusion and future work is given in Section 7.

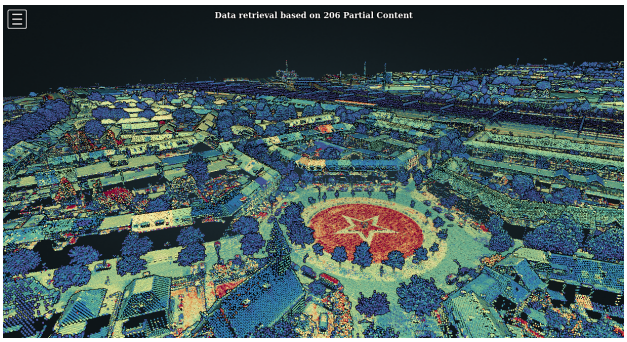

Figure 1. Points from the Actual Height model of the Netherlands (AHN3) data set colored by 'intensity gradient' visualized by Potree, retrieved from PCServe using Range requests

2. RELATED WORK

Butler (2019) gives an overview of web services for point cloud data handling. In particular they investigate: Indexed 3D Scene Layers (I3S), PotreeConverter, Greyhound, Entwine and 3D Tiles. They discriminate a variety of requirements for these services, of which spatial partitioning, and level of detail as well as attribute filtering are important ones to be able to handle for massive point cloud data sets.

Cura et al. (2015) also organise point clouds in a database management system. Although their approach is fully based on using a database, and not a web service, they face similar challenges for organising the points. Their organisation is based on grouping points spatially. Cura et al. (2016) achieve an implicit

This contribution has been peer-reviewed. The double-blind peer-review was conducted on the basis of the full paper.
https://doi.org/10.5194/isprs-annals-X-4-W2-2022-193-2022

level of detail structure by storing the points per group in sorted order.

Van Oosterom et al. (2015), Psomadaki et al. (2016) and Liu et al. (2021) developed an approach to address nD window queries on massive point clouds stored in a database management system. Considering characteristics of spatial data and applications, the approach utilizes a Space Filling Curve (SFC) to encode each nD point into a one-dimensional key. Then, a B+-tree organized table is built to store and index these keys. When querying, the approach transforms the nD query window into a set of non-overlapping one-dimensional SFC ranges for selection. In this transformation, an nD histogram is used to alleviate problems originating from non-uniform data distribution. It is this database organisation that we base our work on.

We note that the majority of approaches first group points into groups of points (also termed patches or tiles), and then use these groups directly for level of detail access. In this work we aim to test how well the reverse works: First we add a mechanism for level of detail on the individual point level (which is independent from a spatial access method), and based on this we make groups of points, making our approach in principle more flexible (supporting different types of block based access to the points).

3. BRINGING ND POINTS STORED IN A DATABASE TO THE WEB

The main task of PCServe is to translate the web service protocol, which a client side visualization application expects for point cloud data retrieval, to database queries and retrieve the points. Once the points are retrieved from the database, the points should be serialized in the correct encoding, which the client is expecting in the response.

Next to dealing with the individual points and their attributes, another important task is thus to make available a spatial access method to access groups (blocks) of points with resolution guarantees. This makes interaction possible at both the overview as well as the most detailed level.

In this section we first review (briefly) the database organisation we use as basis for PCServe. Subsequently we explain how we add a continuous Level of Importance (cLoI) value to each and any point in the data set and finally we show the algorithm we apply to make a virtual block based organisation available in our database.

3.1 Database organisation

State-of-the-art database point cloud organisation is that blocks of points are grouped and stored together (Cura et al., 2015). The SFC based approach pioneered by van Oosterom et al. (2015) stores individual point records inside a table. The SFC key makes it possible to include multiple dimensions (also non-spatial dimensions) in selection criteria. This allows users to be expressive in querying the point clouds, specifying exactly which points to retrieve.

Before using the data in the organization, we decide which dimensions we place inside the SFC key. This depends on the application, but should involve these dimensions on which we will query frequently. For this paper, we consider the position and the level of detail dimensions of the points (but others, like time, could also be used).

The SFC key can either encode the original values to the full wanted resolution (no need to store the data as additional column), or can approximate the original values encoded in the key, which requires that also the original values are stored (or expressed and stored as delta values against the approximation encoded in the SFC key).

We store the points in the database sorted based on the SFC key values. Subsequently, we can efficiently express which points we want to retrieve for a nD convex query geometry: The nD query geometry is translated into 1D SFC ranges (by applying the algorithm of Orenstein and Merrett, 1984). Moreover, this query framework allows to make a trade off between accurate approximation, where we need to spend more time preparing the discretised version of the query geometry, or a faster, but coarser approximation. As points are indexed by SFC key and are stored on disk in the same order as the SFC traverses the hypercube, the database can perform this join between SFC keys and SFC ranges efficiently. Liu et al. (2020) explains in more detail the query mechanism and how a nD histogram helps generating 1D SFC ranges where in the nD hypercube there exists data.

3.2 Obtaining the octree layout

3.2.1 Adding cLoI values for the individual point records

The points we store in the database do not have a level of detail value associated once they are collected. Hence, we add per point the cLoI attribute as floating point number.

Van Oosterom (2019) explains the reasoning behind the cLoI values. In short, the cLoI values follow 'level organisation thinking': Not so many points assigned to the most important level, then (in 2D) four times more points on next level and so on. We use a random generator to get to a cLoI value l for individual points:

$$l = \frac{1}{n} \log_2(U(2^{n(L+1)} - 1) + 1) \tag{1}$$

Equation 1, shows that each point is assigned a floating point importance value l, between 0 and largest level L, where n is the dimensional nature of the data ($n = 2$ for surface scan, $n = 3$ for volumetric data) and U is the value from the random generator. It depends on the data set size what will be the largest needed level L. The uniform distribution given by U is transformed into the wanted distribution of l (with limited number of points on overview level 0, and very many points on most detailed level L).

As the cLoI value has a linear relationship with the density of points, this permits us to request data at a required density, by querying for those points with cLoI value between 0 and wanted level. Note that we add the cLoI value per point, before we encode the x, y, z and cLoI into the SFC key value.

3.2.2 Getting an octree out of the cLoI points

Existing protocols (in use in applications, like Potree and Cesium) require block based organisation, making information available at the server side on which blocks of point data are available, and how these are related to level of detail. Often these strategies employ a 3D octree structure where nodes closer to the root node contain the more important points, and with an additive scheme points residing in deeper levels of the tree are added to the scene once needed, e.g. because a user moves closer to the terrain. As such, an octree will not be a full tree, but can

```
create table ndpc__points_with_sfckey_imp (
 sfc_key             bigint,
 x                   double precision,
 y                   double precision,
 z                   double precision,
 gps_time            double precision,
 importance          double precision,
 intensity           integer,
 return_number       smallint,
 number_of_returns   smallint,
 classification      smallint,
 scan_angle_rank     smallint,
 user_data           smallint,
 point_source_id     smallint,
 scan_direction_flag boolean,
 edge_of_flight_line boolean
);
```

Figure 2. SQL create table statement for the points table. Note, that the SFC key we used is a partial resolution key, containing an approximation for the position and importance level of the points.

have empty nodes from a certain level on-wards (i.e subtrees will be empty). Therefore, we make a table with which nodes in the octree actually contain points by a pre-processing step. For this, we have implemented the following algorithm:

We first determine the 3D spatial extent around all points, i.e. the cubic Axis Aligned Bounding Box (AABB), for which we make the integer side lengths the next multiple of 2 (resulting splits will be integer coordinates again). Then we can start to construct the octree as follows:

- The root node will contain the important points, i.e. those points with their cLoI value between 0 and 1, for the whole spatial extent. We translate the 4D extent that is spanned by this node into a series of 1D SFC ranges, by means of which we query the point table for how many points exist. The SFC ranges generated for the node will be stored for later use. The added cLoI value allows us to select the important points.
- Once, we have obtained this count, and if it is non-zero, then we split spatially the cubic AABB into 8 child nodes. These child nodes will contain the spatially overlapping points for which their cLoI value lie between 1 and 2.
- This splitting in 3D space and incrementing the cLoI level and checking if a node is empty continues recursively, until we reach the largest cLoI value available in the point cloud.

Throughtout this process, we store per node the materialized path to the root (r, r0, r01, r017, . . .) as node identifier, the point count, the SFC ranges and the cubic AABB geometry of the node (to be able to a. visualize easily the nodes their extent and b. apply exact filter after retrieving coarse approximation based on SFC key). After pre-processing we have per pointcloud data set two tables in the database:

- points table, indexed by SFC key value, with all surveyed attributes per point, and extra cLoI attribute (Figure 2).
- hierarchy table, containing per node an identifier and an array of SFC ranges that represents the nD extent (Figure 3). This array of SFC ranges permits us to retrieve per octree node the corresponding points from the points table.

```
create table ndpc__hierarchy_ahn3_points_with_imp_sfc
(
 oct_id            character varying not null,
 level             integer,
 node              geometry(LineStringZM, 28992),
 point_count       bigint,
 sfc_ranges        bigint[]
);
```

Figure 3. Blocks table, storing nodes of the octree

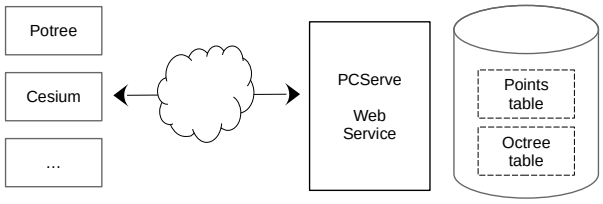

Figure 4. PCServe, the web service component, retrieves points from the database by joining the octree and points table based on the SFC keys (points) and SFC ranges (octree).

3.3 Using PCServe – Encoding the octree and points for clients

Figure 4 illustrates the overall architecture of PCServe. Retrieving a block of points is performing a join between the SFC ranges from the hierarchy table for one block and the point table (a similar query as we used while counting the number of points present in one of the octree nodes). Figure 5 shows that the SFC ranges are retrieved from the hierarchy table indexed by the identifier of the node, and are subsequently joined with the point table on the SFC key value. After retrieval of the point records from the database, PCServe takes care of serialization as demanded by the protocol of the point cloud visualization application.

Next, as the encoding of the points depends on the visualization application used, we survey the existing file formats in use by Potree, a web application that can visualize large point clouds. The rendering is implemented in Javascript, using WebGL and the popular three.js Javascript library. We show how the files are used in the request-response cycle for retrieving point clouds from a web server, forming a protocol on how the point cloud data should be retrieved. Next we describe how this protocol was translated in so-called *routes* of our PCServe web service. A route corresponds to a (Python) function, that is executed in the web service and generates a response, when the visualization client is making a request.

3.3.1 Potree version 1.6 The data structure, which Potree version 1.6 is based on, makes use of separate `.laz` files (American Society for Photogrammetry and Remote Sensing, 2019) storing blocks of points. The blocks are clustered and indexed by a 3D octree (hierarchy), where this tree (its nodes, their extents and point counts) is stored and requested before and separately from the points.

Hierarchy When Potree starts, it first requests the hierarchy file `r.hrc`. This is a binary file that contains the layout of the 3D octree hierarchy, starting at the root node of the octree. This hierarchy is stored as linearized version (breadth first traversal) of its nodes. Per node a one byte value indicates which nodes are and are not present as children. For each node also the point count is stored. For massive

```
with j as (
 select
  sfc_key, x, y, z,
  intensity, classification,
  gps_time, importance
 from
  ndpc__points_with_sfckey_imp p,
  -- join with
  (select
    unnest(sfc_ranges[1:1]) as low,
    unnest(sfc_ranges[2:2]) as high,
    level,
    node
   from
    ndpc__hierarchy_ahn3_points_with_imp_sfc
   where
    oct_id = '<oct_id>'
  ) r
 where p.sfc_key between r.low and r.high
   and p.x between st_xmin(r.node) and st_xmax(r.node)
   and p.y between st_ymin(r.node) and st_ymax(r.node)
   and p.z between st_zmin(r.node) and st_zmax(r.node)
   and p.importance between r.level and r.level + 1
)
select
  x, y, z,
  intensity,
  classification,
  gps_time,
  importance
from
  j
order by
  sfc_key
```

Figure 5. Querying points associated with one octree node. The unnest function takes care of unpacking the SFC ranges stored as array to two columns (named low and high). The first filter (sfc key between low and high) uses the B-tree index to execute the join, while the second filter filters the points exactly on space and importance.

data sets the hierarchy file can get too large to be retrieved at the start of the visualization application in a reasonable amount of time. Therefore, the full octree can be split in multiple subtrees (again `.hrc` files placed in a subfolder), at fixed depth levels (e.g. every fifth level node will start a new subtree).

For PCServe, we have defined a route that allows to retrieve the hierarchy information in the binary `.hrc` format, which Potree expects. When handling this route, the application server connects to the database, and serializes the hierachy table in the requested linearized binary form for the octree. Sorting ascendingly the octree nodes by the length of their identifier, and in case of ties, lexicographically gives us this order.

Points Points belonging to an octree node and their attributes are stored as compressed `.laz` files. Inside the header of the `.laz` file the cubic AABB of the block is stored.

We have defined a route that receives an identifier, queries the database for the right block (and relevant SFC ranges) and then performs the join between hierarchy table and point table for the requested block.

After retrieval the points are serialized in the compressed `.laz` file format.

3.3.2 Potree version 2.0 The newer version of the Potree protocol expects to find 3 files on the server: 1. `metadata.json`, 2. `hierarchy.bin`, 3. `octree.bin`. The metadata file, contains which attributes are present in the point cloud, what is its spatial extent and what is the byte size of the subtree belonging to the root node of the octree. This file is always requested first when the Potree viewer starts.

Subsequently, when a client is requesting both the hierarchy and octree files, not full files are retrieved by the Potree client, but requests are made to retrieve *parts* of the two other files stored at the server side. Note, that in the description we will focus on retrieval of the blocks of points, but the same principles apply to retrieving parts (subtrees) of the hierarchy file as well.

The file format for the hierarchy file is quite similar to version 1.6, but for each node in addition an offset and a size value are stored, so that the client can request the right parts of the file that contain the points.

Hierarchy Fielding et al. (2014) define range requests around which version 2.0 of the file format for Potree is modelled. The client sends in the header field of a GET request the so-called 'Range' header. The response subsequently contains the part of the file that was indicated by the byte range in that header.

Instead of having explicit numeric identifiers which correspond to file names, the offset in bytes in the file where a block starts and how many bytes a block occupies thus becomes the mechanism to identify and retrieve a block of points by. This mechanism is also used for subtree retrieval of the hierarchy file itself.

Points The new format allows the developer to choose from multiple compression types (for which 2 variants have been implemented at the time of writing: uncompressed and Brotli compression). Points belonging to a block are stored together, and are serialized as a stream of bytes inside the `octree.bin` file (for retrieval, range requests are made to this file as well).

In uncompressed form, attributes are stored interleaved: x, y, z, intensity, ..., user attribute$_n$ and this repeats for all point records. In compressed form (where the Brotli compression is used), the point attributes are stored column-wise: x,y,z, ..., x,y,z, intensity, ..., intensity, user attribute$_n$, ..., user attribute$_n$, and per block compression is applied.

Note that in compressed form instead of the real position values, per point a Morton code is stored and all columns are sorted along it (allowing the compression algorithm to compress more efficiently).

In PCServe we have opted to support two request and response types:

1. Partial request (when the 'Range' header is present) with a response by means of the 206 response code allows downloading parts of either the hierarchy (hierarchy.bin) or the points (octree.bin) file;
2. A request for the full dataset (when the 'Range' header is absent). A response, by means of a 200 response code, then allows the client to download the whole pointcloud data set at once.

File	C_69AZ1	C_69AZ2	Both
Point count	149 676 342	501 804 679	651 481 021
.las (MB)	4 191	14 051	18 242
.laz (MB)	743	2 476	3 219
To text (min)	9.9	32.9	–

Table 1. Size of AHN3 input files. As reference also the time to convert the `.laz` file to ASCII text representation (with some attributes per point) is given.

phase	input files C_69AZ1	C_69AZ2	Both (to 1 table)
Add SFC + cLoI and Copy (min)	13.2	44.7	58.8
Index (min)	1.7	5.8	7.6
Cluster (min)	3.5	12.9	18.0
Analyze (min)	0.1	0.1	0.7
Total (min)	18.5	63.6	85.1
nD Histogram (min)	–	–	18.7

Table 2. Preparing the database structure (table and index) with SFC and cLoI added per point.

To make this first type of request / response cycle work with our database backed solution:

1. We have determined for each block, given the linearized version (breadth first traversal sort order) of the octree nodes, the starting point in bytes for each point block. This offset value has been added as extra column to the hierarchy table, which stores the layout of the octree inside the database.
2. We defined a route that from the HTTP request header retrieves the offset. This value then functions as unique key of the blocks, and we can determine which block to request from the database, after which we can join the points and the hierarchy table.

4. RESULTS

We have implemented PCServe and installed it on our server, a HP DL380p Gen8 server with 2× 8-core Intel Xeon processors, E5-2690 at 2.9 GHz, 128 GB of main memory, a RedHat Enterprise Linux 6 operating system. The attached disk storage is a set of 7,200 rpm harddisks (in RAID6 configuration) making available 41 TB of storage capacity. The database management system used is PostgreSQL version 10.1.

We downloaded two files of LiDAR data of AHN3 (the Actual Height data set of the Netherlands) in the `.laz` file format from the national geo-portal. Table 1 shows characteristics of the point cloud files.

Adding the SFC key and importance value for each point, converting the `.laz` files to PostgreSQL `COPY FROM` statements has been implemented with a program coded in the Rust programming language. As Space Filling Curve we have used the Hilbert curve. To determine the cLoI value, we determined beforehand some settings (to be able to express the largest available importance level), such as: total point count and how many points we want to store at the overview level (which will become the root block of the octree in our virtual block hierarchy). We settled on a point count of 100 000.

	Input files C_69AZ1	C_69AZ2	Both (to 1 table)
Table (MB)	15 138	50 750	65 888
B-tree index (MB)	3 362	11 271	14 633
Total (MB)	18 500	62 022	80 521

Table 3. Resulting sizes of the database structure (table and index) with SFC key and cLoI attribute added per point.

Timing SFC ranges generation Hilbert (min)	16.7
Timing queries first and second filter (min)	43.3
Unglued SFC ranges	10 869 967
Glued SFC ranges	2 748 063
Points first filter	2 080 181 654
Points exact (where clause applied)	651 442 607

Table 4. Constructing the octree.

Tables 2 and 3 show the time needed for the load, index and cluster steps and the amount of disk space required by the individual records stored in PostgreSQL. Comparing against uncompressed `.las` files the increase in size (individual records together with added columns SFC key and cLoI and B-tree index) is a factor 4.4×, against compressed `.laz` a factor of 25×.

Deriving the nD histogram after loading the data into the database, was made with a Python script using psycopg2 as connection library to the PostgreSQL database system. This script queries how many points exist for a subpart on the SFC, giving an indication for which parts of the hypercube traversed by the SFC are filled with data. Histogram creation took 18.7 minutes. The histogram was serialized as Python dictionary to disk, consuming 95MB of space (but could have been stored as extra database table as well).

Querying for the octree layout was also implemented as a Python script, connecting to the same database. Table 4 gives some statistics about this process. It shows that the time is divided over: (a) mapping the 4D extent (3D octree geometry and cLoI level) to 1D SFC ranges (16.7 mins) and (b) querying for the point count per octree node (43.3 mins). Figure 6 gives an histogram of how many points are stored per octree node. Table 5 shows that 13 501 octree nodes are present in the resulting octree. Ranges that are directly adjacent on the curve were glued together (but were produced independently from the n-ary tree traversal of the query procedure). As an approximate SFC key has been used, the database retreives a factor 3 times more points initially in the query, that need to be processed by the where clause for exact checking.

We have implemented PCServe with Python and relevant li-

level	oct count	point count
0	1	30 119
1	7	116 422
2	24	450 523
3	74	1 909 593
4	226	7 631 203
5	745	30 542 745
6	2 615	122 157 155
7	9 809	488 604 847
total	13 501	651 442 607

Table 5. Statistics on the octree layout, which we obtained

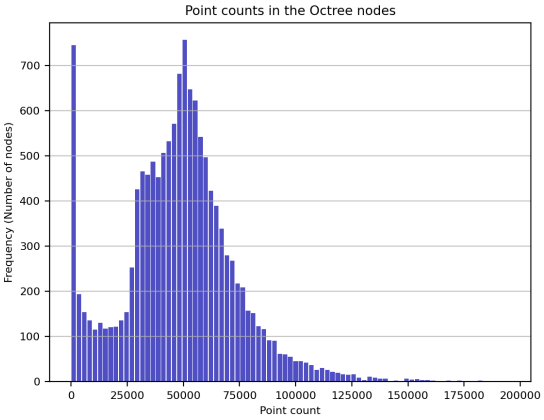

Figure 6. The histogram shows how many points are stored per octree node

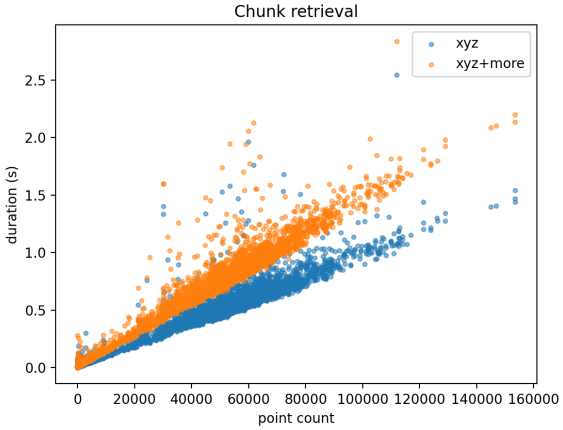

Figure 7. The scatter plot shows how much time is spent (in seconds) to answer a request, given the amount of points that are in the block. A clear trend can be observed between the amount of point data and the time needed to produce the response.

Table 6. Averages for responses made by PCServe

	xyz	xyz+more
Query duration (ms)	329 (72.7%)	416 (69.6%)
Serialize duration (ms)	123 (27.3%)	181 (30.4%)
Total duration (ms)	452	598
Point count	44 555	38 981

braries. The most notable libraries being Flask[1] (for handling the web requests), psycopg2 for connecting to PostgreSQL and laspy (for serialization as `.laz` file). Using the uWSGI[2] server (which is implemented in C and embeds the Python interpreter) we can serve multiple requests of multiple users at the same time.

For Potree v.2.0 uncompressed we have recorded during use of the web service the amount of time it takes to retrieve data for each octree node. We have made available two versions of the end point, one in which only the position of the points is returned and in the second version the position and some more attributes are returned (intensity, classification, importance, GPS time). We logged per request the amount of time that was needed for querying the database, as well as serializing the resultset into the binary format requested by Potree. Figure 7 shows the total time needed for retrieving each block. Table 6 shows average durations, split over time needed for query and serialization. Note, that the two variants (xyz / xyz+more) were used independently, thus showing slightly different average point counts. Based on around $1.1 \cdot 10^4$ logged requests for both variants, we determine the following average times needed for producing a block with points server side:

- xyz: on average 500ms is spent for retrieving 50 000 points
- xyz+more: on average 760ms for querying and serializing 50 000 points

5. DISCUSSION

The total amount of time spent on pre-processing the points was 2.75 hours, where all processing was run serially, which translates to a rate of $2.38 \cdot 10^8$ points per hour. Parts of this process are straight forward to parallellize with multiple threads or processes (e.g. obtaining the nD histogram, querying for the octree layout can be performed per subtree of the childs of the root node, or multiple input files could be simultaneously loaded to the database), but will require modifications to our code (and tuning needs to take place, to not get I/O contention).

Individual records storage is rather expensive, as many records are stored and how Postgres stores these records does also add quite some overhead (e.g. reserving space per record for transaction information). However, storage based on individual records is flexible, and it allows us to make use of different virtual hierarchical / blocking strategies, preventing duplicate storage.

It is clear that although an octree is used, for the AHN3 surface scan, which is 2.5D in nature, the octree therefore behaves more like a quadtree: every next level 4 times more nodes than on previous level (comparing to 3D octree where this would be 8 times more on every level). It also reveals that we missed some points (38 414 points) to place them in an octree node with our querying algorithm. This is due to the fact that even though a node at a certain importance level might be empty of points, child nodes with higher importance level might still contain points (but we already terminated the recursion due to the empty criterion).

The way points are distributed is based on their importance level and their geographic position. This results in blocks that are filled with on average around 48 250 points. However, also quite some nodes contain a very limited amount of points. This is not nice for retrieving them from the web service, as each request will carry some overhead due to the transmission over the network (delay). It may be better to associate these points with the parent node in the octree. We can accomplish this by a post processing step, where we find the underfull nodes and store their SFC ranges with their parents octree node. However, in the octree the level of the node then does not correspond exactly with the integer cLoI range of the points any more. An open question is whether we can also find these nodes when constructing the tree (we could count points for all higher levels for a node as well, but this might be expensive for levels closer to the root node).

The average access time of 500–760 ms (dependent on which

[1] `https://flask.palletsprojects.com`
[2] `https://github.com/unbit/uwsgi`

```
with tbl as (
  select
    sfc_key,
      row_number() over (order by sfc_key) as col
  from
    ndpc__points_table
  order by
    sfc_key
)
select min(sfc_key) from ndpc__points_table
  union all
select sfc_key from tbl where col % 100000 = 0
  union all
select max(sfc_key) from ndpc__points_table
```

Figure 8. Query for determing sfc key bounds, leading to blocks with exactly 100 000 points (except last block which might be underfull). The blocks may overlap each other slightly.

attributes are retrieved) per block of 50 000 points is not necessarily remarkable, but allows for interactive use and the time needed scales log-linearly with the number of points retrieved for a block. In case faster responses are required, we could a. store the blocks of points in the database (but this would be introducing redundancy), b. we could investigate if the database model could be based on blocks (as is current state-of-the art), e.g. by storing the SFC keys in a blob, and index each block by minimum and maximum SFC key in a block, or c. (the most often used blocks) can be cached in memory at the web service level, by means of a component, like MemCached (or even combination of these).

The often used octree strategy divides the points in blocks that are not overlapping for a specific level (distance) from the root node. This makes it efficient to transmit the hierarchy, as the extents of the blocks can be determined based on the materialized path to the root and the global extents, and thus do not have to be stored per node. If we would allow spatial overlap in the blocks, we can produce data blocks that are guaranteed to have n number of points (by following the SFC). Figure 8 shows how we can determine the block extents from the points table with a straightforward SQL query. However, a hierarchy organisation of the blocks will be necessary (e.g. Rtree on the block extents to be able to determine fastly which blocks are needed for overview) and this hierarchy will have to store per block the extents, so the index will grow somewhat larger than in the octree (where bounds are implictly represented). A huge advantage will be that underfull nodes will not be present (as all blocks will be maximally filled with points, allowing to balance request overhead better than in the case with a varying number of points per block).

From looking at the file formats of Potree, we found that the differences between the uncompressed and compressed format of Potree version 2.0 could offer interesting possibilities. A column wise organisation of the attributes could allow applications in principle to request specific columns only. Together with 'Range' requests this is a powerful combination, as in the protocol multiple byte-ranges can be requested, thus allowing an application to ask for certain columns and omit others (reducing request size by not asking for what one does not need). To allow this to function, metadata needs to be added in the hierarchy to where a column starts (in case of compressed columns, in uncompressed form this can be deduced from attribute size and point count, which are both already stored in the metadata file). However, if all columns are compressed together into a block based organisation (as is currently done), the compression may prevent this from working correctly.

6. CONCLUSION

In this paper we have introduced PCServe, our testbed for enabling nD-point cloud data access over the web. Our experiments have shown that Potree is being able to consume the points for rendering, with reasonable interactive request rate from the database, in which we organise the nD point cloud based on deep integration of space and level of detail into a SFC key.

7. FUTURE WORK

For the future, we expect PCServe to be useful to be able to retrieve arbitrary subsets of large point clouds based on the same database organisation. For spatial analyses, which require subsets of points at different resolutions and with arbitrary query geometry, thus not based on a spatial access method, like octree (e.g. for 3D modelling with point cloud data, PCServe could return the points present in a building footprint) could be well supported, allowing applications to retrieve input over the web.

Moreover, for supporting visualization applications, we like to further investigate:

- Supporting other visualization clients (their file format / protocol) accessing PCServe over the web, like Cesium and QGIS. As two different type of protocols are currently supported (request / response of individual files and range requests to one file), we hypothesize that multiple protocols for different viewing applications can indeed be supported. This is on-going implementation work (supporting 3D Tiles and COPC, which both do not differ conceptually much from the two types of protocols we have used for Potree). This then would allow to support multiple applications from the same data organisation (and it would allow to see differences between protocols, e.g. compare compression techniques and different grouping strategies).
- Use the continuous Level of Importance directly in the viewer to gain a more smooth visualization. The octree blocks are a useful mechanism for shipping the data, but a rendering application could use the importance values per individual point for deciding which point (not) to show (close to observer very many points, far away less, with smoother transition than with blocks alone).
- Caching of frequently used blocks. It should be possible to cache relevant blocks (e.g. often used) either in memory, in the database or on the file system.
- Compare PCServe against plain file storage, performance for access time, as well as needed file sizes (determine tipping point when PCServe (which has more 'moving parts', e.g. database server, which needs to be installed) would be more useful; e.g. if just one visualization application is to be supported static file storage on a cloud storage platform also has its use.
- Try non-octree based blocks (with slightly overlapping blocks and different type of spatial index, e.g. K-d tree) and see what advantages and disadvantages such an alternative blocking scheme brings.

- Another idea we could study is to use full resolution SFC keys all up to and including in the visualization client in the transfer pipeline, and unpack the values only when the data is put on the GPU (having a compact representation for network transfer).

- Further study available compression techniques and their relation with file organization of the points in blocks. We should compare what is available (Laz, Draco, Lerpcc, Brotli, Zstandard, ...), their trade off between compression ratio and resources needed for compression, availability of library for supporting compression technique, and possibility for including or omitting attributes from a block.

For organising the points in the database, the work on PCServe has revealed we could work on:

- Reducing time needed for pre-processing, by exploiting parallellism: Parallel load / creating octree instead of using serial algorithm, exploiting multi-core CPU system properties. Related is scaling out to processing a nation wide dataset. Maybe we need a cluster of database systems (nodes), to serve such massive point clouds. A question we should investigate then is how to partition the data over the nodes.

- Maybe we should use a block based organisation as well in the database by means of column store type table organisation, (e.g. Citus provides such a storage type for tables in Postgres, cf. Cubukcu et al., 2021). This could add compression in the database transparently for the application and thus solve the problems of the large amount of storage size needed for individual records.

- Integrate other dimensions in the organisation, like time and object identification (supporting visualization applications that involve temporal aspects, like change detection).

With PCServe we have focussed on visualization applications with read-only access. However, using point clouds entails more:

- For uploading and processing (organising) point clouds by others into a repository of point cloud data, PCServe could play a role (e.g. structuring the point cloud server-side – carrying out the pre-process as in this paper). This will be a long running processes. Alignment with the OGC API Processes standard could allow asynchronous handling of such tasks, allowing a user to check back on status of task, and be notified once such a task is finished.

- Some applications may want to modify the point clouds, e.g. store additional attributes per point, like adding a normal vector to the points (effectively changing the schema of the points table) or even remove points. How to handle these kind of modifications?

An open question remaining in this respects is: Could a unified protocol be standardised, that is suitable for all (read/write/update) scenario's and is well supported by applications that are handling point cloud data.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for providing constructive remarks. Moreover, we like to thank Haicheng Liu and Markus Schütz for a thorough proofreading of the draft version of this paper.

References

American Society for Photogrammetry and Remote Sensing, 2019. LAS Specification 1.4 - R15.

Butler, H., 2019. OGC Testbed-14: Point Cloud Data Handling Engineering Report. Technical report, Open Geospatial Consortium.

Cubukcu, U., Erdogan, O., Pathak, S., Sannakkayala, S., Slot, M., 2021. Citus: Distributed PostgreSQL for data-intensive applications. *Proceedings of the 2021 International Conference on Management of Data*, ACM.

Cura, R., Perret, J., Paparoditis, N., 2015. Point cloud server (PCS): Point clouds in-base management and processing. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, II-3/W5, 531–539.

Cura, R., Perret, J., Paparoditis, N., 2016. Implicit LOD using points ordering for processing and visualisation in point cloud servers.

Fielding, R., Lafon, Y., Reschke, J., 2014. Hypertext Transfer Protocol (HTTP/1.1): Range requests. Technical report, Internet Engineering Task Force (IETF).

Liu, H., Thompson, R., van Oosterom, P., Meijers, M., 2021. Executing convex polytope queries on nD point clouds. *International Journal of Applied Earth Observations and Geoinformation*, 105(102625), 1-11.

Liu, H., van Oosterom, P., Meijers, M., Guan, X., Verbree, E., Horhammer, M., 2020. HistSFC: Optimization for nD massive spatial points querying. *International Journal of Database Management Systems (IJDMS)*, 12(3), 7–28. https://aircconline.com/abstract/ijdms/v12n3/12320ijdms02.html.

Orenstein, J. A., Merrett, T. H., 1984. A class of data structures for associative searching. *Proceedings of the 3rd ACM SIGACT-SIGMOD symposium on Principles of database systems - PODS '84*, ACM Press.

Psomadaki, S., van Oosterom, P., Tijssen, T., Baart, F., 2016. Using a Space Filling Curve approach for the management of dynamic point clouds. E. Dimopoulou, P. van Oosterom (eds), *ISPRS Annals Volume IV-2/W1, 11th 3D Geoinfo Conference*, Athens, 107–118.

van Oosterom, P., 2019. From discrete to continuous levels of detail for managing nD-PointClouds. Keynote presentation at ISPRS geospatial week, June 13, 2019, Enschede, Netherlands.

van Oosterom, P., Martinez-Rubi, O., Ivanova, M., Horhammer, M., Geringer, D., Ravada, S., Tijssen, T., Kodde, M., Gonçalves, R., 2015. Massive point cloud data management: Design, implementation and execution of a point cloud benchmark. *Computers & Graphics*, 49, 92–125.