VOXEL DATA MANAGEMENT AND ANALYSIS IN POSTGRESQL/POSTGIS UNDER DIFFERENT DATA LAYOUTS

W. Li¹, S. Zlatanova¹, B. Gorte¹
¹ Faculty of the Built Environment, The University of New South Wales, NSW 2052, Australia
(wei.li, s.zlatanova, b.gorte)@unsw.edu.au

Commission IV

KEY WORDS: Voxel, Spatial Model, Data Management, PostgreSQL

ABSTRACT:

Three-dimensional (3D) raster data (also named voxel) is important sources for 3D geo-information applications, which have long been used for modelling continuous phenomena such as geological and medical objects. Our world can be represented in voxels by gridding the 3D space and specifying what each grid represents by attaching every voxel to a real-world object. Nature-triggered disasters can also be modelled in volumetric representation. Unlike point cloud, it is still a lack of wide research on how to efficiently store and manage such semantic 3D raster data. In this work, we would like to investigate four different data layouts for voxel management in open-source (spatial) DBMS - PostgreSQL/PostGIS, which is suitable for efficiently retrieving and quick querying. Besides, a benchmark has been developed to compare various voxel data management solutions concerning functionality and performance. The main test dataset is the groups of buildings of UNSW Kensington Campus, with 10cm resolution. The obtained storage and query results suggest that the presented approach can be successfully used to handle voxel management, semantic and range queries on large voxel dataset.

1. INTRODUCTION

Voxels are volumetric pixels, which are spaced in a regular grid in three-dimensional (3D) space and are perceived without gaps between them. In contrast to the similar concept of point clouds, voxels can only inhabit discrete positions in space dictated by the grid. The grid is regular and all possible voxel positions are equally spaced. Voxels are the quickest way to quickly model and visualize volumetric data (especially in natural or organic formations).

Applications for voxels include the visualization and analysis of medical and scientific data (Chmielewski, Tompalski, 2017), and representation of terrain in games and simulations. Medical researchers are now using volumetric imaging to view Magnetic Resonance Imaging (MRI) scans from different angles, effectively to see the inside of the body from outside. Minecraft (Duncan, 2011), a sandbox video game, uses voxels to store terrain data. Geologists often use voxel modeling techniques to model geological features like terrain and elevation. More broadly, scientists can use voxel-based modeling to visualize and measure the volume of anything from fluids to green spaces in urban centers (Anderson et al., 2018). Voxel are also fundamental in several recent approaches to semantic labeling. Hâne, C. et al. (Hane et al., 2013) use voxels to represent spatial constraints on scene labels. The regular voxel grid also provides an ideal basis for deep Convolutional Neural Networks (CNNs), as has been recently demonstrated by Zhou and Tuzel (Zhou, Tuzel, 2018).

Same as with the management of other geographic data, voxels have for many years now been managed in the traditional way of using file system with different data structures. These files can then be stored in a hierarchy of folders for sparse voxel models such as a 3D city model with different LODs and accessed by the third-party software. Generally, voxel can be characterized as being mostly static data sets, that means, once voxels are processed, there are few opportunities to modify or update them. During the last decade, Relational Database Management Systems (DBMSs) are the most widely used and the most mature database systems, and they have been applied in various industries. To enrich geospatial functions and geospatial processing capability, in 2010, Open Geospatial Consortium (OGC) issued “OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 2: SQL option” (Herring, 2010), which defines basic geometry types like Point, Curve, Surface and Polygon. Examples of object-relational DBMS for geographic information include PostGIS (Obe, Hsu, 2011), Oracle 11g (Kothuri et al., 2008), and Microsoft SQL Server (Fang et al., 2008). These relational databases can define geospatial objects, and adopt different indexes for fast spatial queries (Binary Tree in SQL Server, Binary Tree, R-Trees, and Generalized Search Tree in PostGIS) (Guo, Onstein, 2020).

To make use of voxel data in different scenarios, an efficient storage and retrieval system is therefore needed. In this paper, we investigate four data layouts for storing and managing the 3D raster data in PostgreSQL/PostGIS, namely - (1) a flat ARRAY table; (2) a POINT geometry table; (3) a MULTIPoint geometry table, and (4) a PCPATCH table with the help of Pointcloud. We present two different implementations of these four data layouts that use the INTEGER data type with scaling and offset or use the NUMERIC data type directly, depending on its use.

Our primary contributions are as follows:

1. A novel perspective on different data layouts on voxel models is proposed considering voxels with multiple offset coordinates for the sake of improving the ability of expression of the same built environment from multiple perspectives.

* Corresponding author

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-VI-3-W1-2020-35-2020 | © Authors 2020. CC BY 4.0 License.
2. A high-level conceptual design and relational database design have been developed for voxel models in a joint manner.

3. An exploration and comparison of semantic and spatial analysis on our proposed data layouts are developed to investigate suitable voxel data management solution in different application scenarios.

The rest of this paper is structured as follows: Section 2 gives a brief retrospect to the voxel and voxel data management. In addition, the essential aspects and approaches for voxel modelling using geo-database are discussed. Section 3 proposes our design and implementation of 3D raster data management in PostgreSQL/PostGIS with details about the four kinds of data layouts. Moreover, a case study on the UNSW campus to demonstrate the usability and efficiency of the proposed storage model is shown in Section 4. The last section draws the conclusions about the presented work and outlines the relevant aspects of our future research and developments tasks.

2. MANAGEMENT OF 3D RASTER DATA

(3D) Raster data have evolved to establish a major source of information for many topographic applications (Psomadaki, 2016), and representing 3D urban scenes by voxels brings a number of advantages (Zlatanova et al., 2016). However, due to their large volume and semantic complexity, management of voxels is a rather challenging topic. For many years voxels have been managed using file-based solutions. Recently, new ways to manage voxels can be achieved using RDBMS.

2.1 Voxels

Our world can be modelled in voxels by gridding the 3D space and specifying what each cell represents by semantically “attaching” every cell/voxel to a real-world object (Goncalves et al., 2016). Storing volumetric spaces such as buildings, air, water and terrain is possible. Because of the nature of the way that they are generated, voxels are considered structured data (found on a regular grid). On the one hand, voxels can be considered as vector-based structures (since they are a collection of grids, but on the other hand they present many differences with point clouds, given that point clouds are not a regular grid and unstructured data).

Each voxel \( V_i \), that is part of a voxel object has three coordinates, namely \( X_i, Y_i, Z_i \), and attached several attributes. These attributes are highly correlated to the way the voxel was generated in the first place, such as classification information (which corresponds to the type of real-world object the voxel belongs to, e.g., tree, building etc.). Every object is defined by a set of voxels, with set’s length depending on the level of detail (LOD) (Li et al., 2019). The storage unit base is a 3D voxel of a certain size and each voxel’s characteristics e.g. type (wall, glass, roof, door, etc.), color, density, etc. is then stored as a semantic property. Representing real-world objects by a single geometry type (3D cube) instead of a collection of polygons/polyhedron greatly simplifies a range of geometric operations: volumes and areas are calculated by simply counting the number of voxels that form an object; 3D bisections become simple selection operations; dynamic Levels-of-Detail (LOD) as objects can be resampled with larger cubes (Zlatanova et al., 2016).

2.2 Voxels Management

Same as with the management of other geographic data, a file-based approach is the most common and basic approach to manage 3D raster data. Traditionally, 3D raster data have been stored in XYZ-format or its compressed format – VX3. The two most important schemes to represent voxels are Octree (Laine, Karras, 2010) and vector implementation. For a realistic representation of highly complex shapes, the Octree grows closer to its full size (Patil, Ravi, 2005), and the vector implementation may be simpler as well as more efficient in terms of storage and further analysis or visualization. Storing the voxel data is mostly achieved using ASCII, which have been utilised mostly because they are human-readable. For instance, for a binary voxel model, we use a single bit (0 or 1) to represent the state of each voxel. For a general voxel model, the absolute or relative coordinates combining several attributes are used to represent each voxel.

However, the file-based approach is not suitable for data sharing, remote accessing and updating. Considering data sharing among different applications, concurrent access control is difficult to achieve. Besides, since the amount of voxels easily reaches millions, on-the-fly indexing is required in order to achieve efficiency while processing. This, together with the increasing availability of voxel information, makes files an inefficient method of organisation, as being aware of which datasets are where and when is not simple (Psomadaki, 2016).

Besides the file-based solution, voxel can also be stored in a Database Management Systems (DBMSs), relational or not, like Oracle, PostgreSQL (Momjian, 2001) and MonetDB (Zlatanova et al., 2016). The use of files to manage voxels is viable for limited space coverage, however, in the case of large-scale coverage on a precinct or regional scale, a database is more appropriate. The DBMS-based approach presents many advantages over file-based organisations. One of the motivating factors in using a DBMS to store voxels is their usefulness in providing access to the data via a declarative language, like SQL (Structured Query Language) (Chamberlin, Boyce, 1974) for Relational DBMS (RDBMS). Besides, DBMSs offer benefits in terms of security, concurrent access, scalability, management of updates, quick access through indexing and user management. At the same time, it is easier for the integration with other types of (geo-) spatial data already stored in databases.

3. DATA LAYOUT

In this section, we first describe the sample voxel dataset, then we test several different data layout for storage and queries.

3.1 The Sample Voxel Data

We select part of the University of New South Wales (UNSW) Kensington lower campus as case study. The raw voxel dataset includes six building objects with total 33,170,471 voxels as described in Table 1. We assign ID and name for each building object, such like the first building named “Built Environment” with \( objID = 1 \). It is worth noting that each data set has the same scale and different offset. Besides, since the data source for the voxel data came from the existing BIM model, each voxel itself carries some IFC attributes \(^1\) like IfcBeam, IfcDoor. After statistics, we list all the IFC attributes in Table 2.

\(^1\) http://standards.buildingsmart.org/IFC/RELEASE/IFC4/ADD2_TC1/HTML/
3.2 The Software and Hardware That Was Used

A high performance HP laptop with 16GB RAM and a 2.80GHZ Intel(R) Core(TM) i7-7600U CPU, running on a 64bit Windows 10 Enterprise and equipped with an internal 512GB SSD for disk storage was used for all the tests.

PostgreSQL 11.2 together with pgAdmin (version 4.3) is used as a RDBMS for storage and running spatial or semantic queries. For general query, we directly use pgAdmin to run the query and evaluate the running time. Moreover, so as to obtain an accurate time, for each query we issued the command ‘EXPALIN ANALYZE stmt’, where stmt is the query that was run. The query tie is the sum obtained from the planning and execution times and it is these values which are reported in the result tables.

Further, QGIS software can be used for visualising entities. It can open a model across internet connection to the database server, edit that model and save the amended model either to a model file on the local system or merge it back into the source model on the server.

3.3 ARRAY Table

The majority storage models can be adopted for voxel management is based upon the organisation of voxels in the flat ARRAY table, where each voxel is stored separately in a single row using common data types (INTEGER or NUMERIC). Each voxel attribute constitutes a separate field. The voxels are populated in a table and indexed using a B-tree index on the target scale.

Due to our subsequent spatial queries will involve cross-object retrieval, we integrate six buildings into one flat table with corresponding semantic object ID, named objID that represents the building ID in Table 1.

Concerning semantic labels, we set up other two tables: ifcclass and objclass, which stand for IFC semantic information in each voxel grid and object semantic information of voxels in each building, where scale and offset values are stored in objclass.

To enhance database performance, PostgreSQL provides several index types: B-tree, Hash, GiST, SP-GiST and GIN. Each index type uses a different algorithm that is best suited to different types of queries. In our case, for the flat PostgreSQL ARRAY table, we create B-tree indexed on (X, Y, Z) since equality and range queries on these columns are more often in daily use, and then populate with data from using SQL script via SQL Shell. An additional index was then added on columns objID and ifcID, which refer to the building semantic ID.

3.4 POINT Table

A spatial POINT represents a single location on the Earth. This point is represented by a single coordinate (including either 2-, 3- or 4-dimensions). Points are used to represent objects when the exact details, such as shape and size, are not important at the target scale.

For POINT table, we keep both objID and ifcID semantic information, and create one geometry column with 3D point in PostGIS. To achieve this data layout, it is essential to know the following relevant functions in PostGIS:

- geometry ST_MakePoint(float x, float y, float z);

The script creating POINT table is shown in List 1. Regarding to index, we create B-tree index on two semantic columns objID and ifcID, and GiST index on geom columns (see Figure 2)

3.5 MULTIPOLY Table

MULTIPOLY is another geometry that consists of a collection of POINT. In this kind of layout, we consider regarding each building object and each IFC object as one multipoint, that means, voxels in one MULTIPOLY geometry have same ifcID and objID. To achieve this data layout, it is essential to know the following relevant functions in PostGIS:
When using the multipoint geometry type, we need to consider how to cut the voxel data, that is, those voxels stored in a multipoint object. We propose a semantic-based voxel data partitioning strategy. Specifically, we want to store all voxels with the same semantic information in a multipoint object, including object semantics and IFC semantics. Moreover, similar to POINT, two indexes (i.e., B-tree and GiST) are built on (objID, ifcID) and geom, respectively. The following script shows the details (see List 2) and an example of a table created in PostgreSQL to store POINT layout.

```
CREATE EXTENSION IF NOT EXISTS POSTGIS;
DROP TABLE IF EXISTS voxelpt CASCADE;
CREATE TABLE voxelpt
(
    id serial PRIMARY KEY,  
    objID INTEGER,  
    ifcID INTEGER,  
    geom geometry  
);
INSERT INTO voxelpt(objID, ifcID, geom)
SELECT objID, ifcID, ST_MakePoint(x,y,z)
FROM voxel AS VALUES;
DROP INDEX IF EXISTS geom_voxelpt CASCADE;
DROP INDEX IF EXISTS idx_voxelpt CASCADE;
CREATE INDEX geom_voxelpt ON voxelpt USING GIST (geom);
CREATE INDEX idx_voxelpt ON voxelpt(objID);
DO $$
DECLARE
    f record;
BEGIN
    FOR f in SELECT DISTINCT objID, ifcID FROM voxelpt
    LOOP
        INSERT INTO voxelpt(objID, ifcID, geom)
VALUES (f.objID, f.ifcID, ST_Collect(ARRAY(SELECT geom
WHERE objID=f.objID AND ifcID=f.ifcID))));
    END LOOP;
END $$
;
DROP TABLE IF EXISTS voxelpt CASCADE;
CREATE TABLE voxelpt
(
    id serial PRIMARY KEY,  
    objID INTEGER,  
    ifcID INTEGER,  
    geom geometry  
);
```

Listing 1. Example of converting ARRAY to POINT layout.

### 3.6 PCPATCH Table

Pointcloud ² is a PostgreSQL extension for storing point cloud (LIDAR) data, where PcPatch can be regarded a potential structure used for management of voxel model in PostGIS. In our case, we collect a group of voxels with same semantic information into a PcPatch. Each patch should hopefully contain voxels that are near together. That is the same partition strategy as MULTIPOINT (see Section 3.5).

Following the Pointcloud schema ³ (PostgreSQL Pointcloud deals with all this variability by using a “schema document” to describe the contents of any particular LIDAR point.), we prepare a schema document to describe the contents of any particular voxel. Each voxel contains three dimensions, named X, Y, Z, and each dimension will be of INTEGER data type, with scaling 0.1. This schema document is stored in the pointcloud_formats table, along with a pcid (i.e., “pointcloud identifier”) (see List 3).

```
<table>
<thead>
<tr>
<th>id</th>
<th>objID</th>
<th>ifcID</th>
<th>geom</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
<td>19</td>
<td>MULTIPoint (515.494 37.516 493)</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>27</td>
<td>MULTIPoint (75 252 66.75 253 66)</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>8</td>
<td>MULTIPoint (77 293 114 77 293)</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>20</td>
<td>MULTIPoint (11 250 65.11 250 66)</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>2</td>
<td>15</td>
<td>MULTIPoint (75 186 59.79 186 60)</td>
</tr>
</tbody>
</table>
```

Listing 2. Example of converting POINT to MULTIPOINT layout.

The central role of the schema document in interpreting the contents of a point cloud object means that care must be taken to ensure that the right pcid reference is being used in objects, and that it references a valid schema document in the pointcloud_formats table. The goal of voxel storage is to try and keep this small because there’s so much data. So data are packed into a byte array, using as few bytes as possible to represent each value, as shown in Figure 4.

```
<pc:dimension>
    <pc:position>1</pc:position>
    <pc:size>4</pc:size>
    <pc:description>
        X coordinate as a long integer.
    </pc:description>
    <pc:name>X</pc:name>
    <pc:interpretation>int32_t</pc:interpretation>
    <pc:scale>0.1</pc:scale>
</pc:dimension>
```


Different from multipoint representation, GiST index is created based on 2D bounds of the patch because it cannot be indexed directly on the PcPatch type. Fortunately, Pointcloud provides `PC_EnvelopeGeometry(PcPatch)` functions that can directly obtain bounding box as a PostGIS Polygon 2D. Thus, we can index 2D Polygon. The following script shows the details (see List 4).

² https://github.com/pgpointcloud/pointcloud
³ https://github.com/pgpointcloud/pointcloud
ordinates in EPSG 28356 projection. Due to space limits, we tables with NUMERIC data type to directly storage real co-

Within this section, we evaluate and analyse the query per-

4. EXPERIMENTS

Within this section, we evaluate and analyse the query performance on voxels under different data layout in Postgr-
eSQL/PostGIS database. In order to evaluate the balance between disk space and query time. We create corresponding

table and one building object table including scale and offset in the database table.

Eval-I: Load all voxels for particular building. We are going to load all voxels associated with building “Built Environment” in UNSW campus. To achieve this loading, table objclass will be joined with main voxel table since it can provide building name information. The queries in Table 3 are conducted and results are demonstrated in Table 4, where “INTEGER” stands for queries happen on table with INTEGER data type and so do “NUMERIC”. Querying time of ARRAY and MULTIPOLY is steady for the discussed two data types - INTEGER and NUMERIC. POINT takes slightly more time for NUMERIC type and PCPATCH costs less on it. It is obvious that MULTIPOLY and PCPATCH runs much faster than the first two data layouts due to building ob-

Eval-II: Load all voxels for particular IFC object. Similar to the first experiment, this time we retrieve the IFC semantic information. There are 26 different IFC features and we pick up IfcDoor as the query condition. That is, our goal is to retrieve all door voxels in this dataset. It is straightforward that combining main voxel table with ifcclass, as depicted in Table 4 and Figure 7. The running time showed almost the same trend as Eval-I. That is, MULTIPOLY and PCPATCH are more efficient when doing such a semantic query.

4.2 Spatial Analysis

In this section, we investigate the spatial analysis on different data layouts.

Eval-III: Distance between two buildings. In our case, we consider to use 2D box to represent each building object and calculate the distance between two buildings as the distance between two centers of box2d, which is a spatial data type used to represent the two-dimensional enclosing box of a geometry or collection of geometries. The representation contains the values xmin, ymin, xmax, ymax. These are the minimum and maximum values of the X and Y extents. To achieve this query, it is essential to use the following relevant functions in PostGIS:


**Data Layout**

<table>
<thead>
<tr>
<th>SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY</td>
</tr>
<tr>
<td>POINT</td>
</tr>
<tr>
<td>MULTIPOINT</td>
</tr>
<tr>
<td>PCPATCH</td>
</tr>
</tbody>
</table>

- float ST_Distance(geometry g1, geometry g2);
- box2d ST_MakeBox2D(geometry pointLowLeft, geometry pointUpRight);
- geometry ST_Scale(geometry geomA, float XFactor, float YFactor, float ZFactor);
- geometry ST_Translate(geometry g1, float deltax, float deltay, float deltaz);

Next, in order to calculate the distance between any two buildings, scaling and translation are applied on “X” and “Y” values first to convert them into correct coordinate projection, and then using ST_Distance function to return the minimum 2D Cartesian (planar) distance between two geometries, in projected units (spatial ref units).

Considering the four kinds of data layout we have proposed so far, the flat ARRAY schema is a special one that does not contain geometric feature. According to box2d definition, we calculate a set of xmin, ymin, xmax, ymax for each building object to build a box2d object as follows:

In addition, PCPATCH is also a special one whose data type is specifically designed for point clouds. Fortunately, the pointcloud_postgis extension adds functions that allow to use PostgreSQL Pointcloud with PostGIS, converting PCPATCH to Geometry. In our case, following operation are utilized to return the 2D bounds of the patch as a PostGIS Polygon 2D:

```
CREATE VIEW box AS
SELECT ST_Translate(ST_Scale(ST_SetSRID(ST_Point(minX, minY), ST_Point(maxX, maxY)), 1.0), 0.0, 0.0), geom;
```

Table 3. Load all voxels for particular building.

<table>
<thead>
<tr>
<th>Semantic</th>
<th>ARRAY</th>
<th>POINT</th>
<th>MULTIPOINT</th>
<th>PCPATCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFC</td>
<td>INTEGER</td>
<td>4,203.041</td>
<td>13,665.43</td>
<td>0.741</td>
</tr>
</tbody>
</table>

Table 4. The query times for four queries using the dataset. All times are in milliseconds.

```
CREATE VIEW box AS
SELECT ST_Extent(ST_Translate(ST_Scale(ST_SetSRID(ST_Point(minX, minY), ST_Point(maxX, maxY)), 1.0), 0.0, 0.0))::geometry AS geom;
```

```
CREATE VIEW box AS
SELECT ST_Extent(ST_Translate(ST_Scale(ST_SetSRID(ST_Point(minX, minY), ST_Point(maxX, maxY)), 1.0), 0.0, 0.0))::geometry AS geom FROM voxelpatch GROUP BY objID AS tmp;
```

Listing 5. Example of generating bounding box for MULTIPOINT layout.

```
CREATE VIEW box AS
SELECT ST_Point(maxX, maxY) AS geom FROM voxelpatch;
```

Listing 6. Example of generating bounding box for PCPATCH layout.

Based on above analysis, We search all buildings that within 100 meters of “Built Environment”. Our query result is visualized in Figure 8, where the caption of Figure 8 gives detail of visualization. Besides, we compare the time for constructing bounding box for above query result (including “Built Environment” itself) on different data type under MULTIPOINT and PCPATCH layouts. From Table 5, it is easy to find that NUMERIC type is more efficient to get the minimum bounding box for the supplied geometry, particular in PCPATCH. That is because coordinates scaling and translation can be avoid during such query and PostgreSQL Pointcloud provides function PC_EnvelopeGeometry to return 2D bounds directly. In contrast, when using INTEGER type for PCPATCH, PcPatch object needs to be converted into PcPoint object and then casts PcPoint to the PostGIS geometry.

**Eval-IV: Analysis of internal objects in building.** In this experiment, we analyze some positional relations among rooms in UNSW campus based on voxel model, in which we are gen-

![Figure 7. Visualisation of Eval-II via QGIS.](image-url)
To achieve this query, we first should acquire the roof height called “Science Theatre”. In the voxel model, attribute value IfcSpace represents room space and we pick the minimum z value as the height of one room. Here, we adopt POINT layout since in MULTIPoint and PCPATCH particular IFC object only has one geometry, once you compare the geometry, you either return the whole geometry or nothing, unless the geometry is broken up into a single point.

To achieve this query, we first should acquire the roof height called “Science Theatre”. In the voxel model, attribute value IfcSpace represents room space and we pick the minimum z value as the height of one room. Here, we adopt POINT layout since in MULTIPoint and PCPATCH particular IFC object only has one geometry, once you compare the geometry, you either return the whole geometry or nothing, unless the geometry is broken up into a single point.

5. DISCUSSION AND CONCLUSIONS

In this paper, a relational 3D geo-database solution for the management and analysis of voxel model with multiple scaling and offsets were presented. Four different kinds of data layouts for voxel data management in PostgreSQL/PostGIS are investigated in this paper. A case study shows that different data layout can be applied to different situation to get better performance. For instance, when retrieving a particular building or IFC object in its entirety, it is better to consider MULTIPoint and PCPATCH. Moreover, these two kinds of layouts bring other benefits as well, such as small disk space and fast loading. On the contrary, ARRAY and POINT are more flexible layouts that allow a user to look for one or some voxels with special semantic information. Compared to ARRAY, POINT as a basic geometry, can take full advantage of spatial functions inherent in PostgreSQL. However, since only one voxel object is recorded per row, these two representations can take up a lot of disk space. In our test, four layouts take up 1.651MB, 2.672MB, 64KB, and 16KB of disk space, respectively.

There are several possible directions that can be explored for future works. First, how to decide the partitions for MULTIPoint and PCPATCH schema is an interesting topic, for the time being, we divide voxel objects with the same semantics directly into a group. In fact, depending on the different application scenarios we face and the frequent query operations, we can consider different ways of partitioning. Moreover, PCPATCH looks like a very promising data layout for voxel model in PostgreSQL/PostGIS.

With the arrival of big data, voxel applications are also required changeable data schemes, faster query response times, and more flexible scalability than traditional relational databases currently using (Li et al., 2020). To respond to these new challenges, NoSQL (Not only SQL) databases are now being adopted for geospatial data management. We would design and implement 3D voxel management in the most popular NoSQL databases in the next step.

REFERENCES


Psomadaki, S., 2016. Using a Space Filling Curve for the management of dynamic point cloud data in a Relational DBMS.
