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

W. Li[1]*, S. Zlatanova[1], B. Gorte[1]

[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). Voxels are also fundamental in several recent approaches to semantic labeling. Häne, 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

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 – `VOX`. 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/

Building ID	Name	Scale	Offset	Number of voxel
objID=1	Built Environment	10cm	(336300,6245507,25)	17,460,029
objID=2	Block House	10cm	(336042, 6245613, 27)	3,392,202
objID=3	Dalton	10cm	(336305, 6245569, 29)	1,887,512
objID=4	Quadrangle	10cm	(336409, 6245580, 31)	3,161,733
objID=5	Round House	10cm	(336047, 6245651, 25)	6,037,174
objID=6	Science Theatre	10cm	(336325, 6245582, 28)	1,231,821

Table 1. Object ID, Name, Scale and Offset in Dataset.

IfcBeam	IfcBuildingElementPart	IfcBuildingElementProxy	IfcColumn
IfcCovering	IfcDiscreteAccessory	IfcDistributionElement	IfcDoor
IfcFlowSegment	IfcFlowTerminal	IfcFurnishingElement	IfcMember
IfcOpeningElement	IfcPlate	IfcRailing	IfcRamp
IfcRampFlight	IfcSpace	IfcRoof	IfcSite
IfcSlab	IfcStair	IfcStairFlight	IfcWall
IfcWallStandardCase	IfcWindow		

Table 2. Code list of IFC attribute (26 in total).

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 in the `X`, `Y`, and `Z` coordinates respectively. In our case, the coordinates are saved into a 4 byte-integer and the scale and offset is stored as a table property.

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, as 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.

	id [PK] integer	x integer	y integer	z integer	objid integer	ifcid integer
1	1	16	312	118	1	3
2	2	16	312	119	1	3
3	3	16	312	120	1	3
4	4	16	312	121	1	3
5	5	16	312	122	1	3

Figure 1. Example of a table created in PostgreSQL to store `ARRAY` layout.

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 `MULTIPOINT` Table

`MULTIPOINT` 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 `MULTIPOINT` geometry have same `ifcID` and `objID`. To achieve this data layout, it is essential to know the following relevant functions in PostGIS:

```
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 idx_voxelpt CASCADE;
DROP INDEX IF EXISTS geom_voxelpt CASCADE;
CREATE INDEX idx_voxelpt ON voxelpt(objID);
CREATE INDEX geom_voxelpt ON voxelpt USING GIST (geom);
```

Listing 1. Example of converting `ARRAY` to `POINT` layout.

	id [PK] integer	objid integer	ifcid integer	st_astext text
1	24557297	4	24	POINT Z (484 347 181)
2	24557298	4	24	POINT Z (484 347 182)
3	24557299	4	24	POINT Z (484 347 183)
4	24557300	4	24	POINT Z (484 347 184)
5	24557301	4	24	POINT Z (484 347 185)

Figure 2. Example of a table created in PostgreSQL to store `POINT` layout.

- *geometry* ***ST_Collect****(geometry[] g1_array);*

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 `MULTIPOINT` layout is shown in Figure 3.

	id [PK] integer	objid integer	ifcid integer	st_astext text
1	1	5	19	MULTIPOINT Z (515 494 37,516 493 ...
2	2	4	27	MULTIPOINT Z (75 252 66,75 253 66,...
3	3	1	8	MULTIPOINT Z (77 293 114,77 293 1...
4	4	4	20	MULTIPOINT Z (11 250 65,11 250 66,...
5	5	2	13	MULTIPOINT Z (79 186 59,79 186 60,...

Figure 3. Example of a table created in PostgreSQL to store `MULTIPOINT` 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 PostgreSQL. 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"

[2] https://github.com/pgpointcloud/pointcloud
[3] https://github.com/pgpointcloud/pointcloud

```
DROP TABLE IF EXISTS voxelmpt CASCADE;
CREATE TABLE voxelmpt
(
        id serial PRIMARY KEY,
        objID INTEGER,
        ifcID INTEGER,
        geom geometry
);
DO $$
DECLARE
    f record;
BEGIN
    FOR f in SELECT DISTINCT objID, ifcID
                FROM voxelpt
    LOOP
        INSERT INTO voxelmpt(objID, ifcID, geom)
        VALUES (f.objID, f.ifcID,
        ST_Collect(ARRAY(SELECT geom
        FROM voxelpt
        WHERE objID=f.objID AND ifcID=f.ifcID)));
    END LOOP;
END;
$$
DROP INDEX IF EXISTS idx_voxelmpt CASCADE;
DROP INDEX IF EXISTS geom_voxelmpt CASCADE;
CREATE INDEX idx_voxelmpt ON
voxelmpt(objID, ifcID);
CREATE INDEX geom_voxelmpt ON
voxelmpt USING GIST (geom);
```

Listing 2. Example of converting `POINT` to `MULTIPOINT` layout.

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).

```
<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>
```

Listing 3. Example of schema document for `PCPATCH` 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 thins 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.

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).

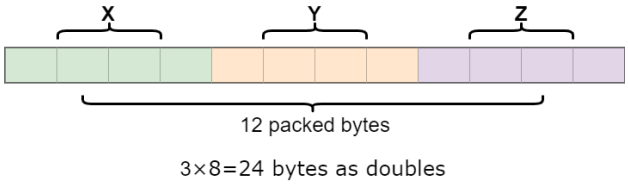

Figure 4. Example of packed bytes.

```
DROP TABLE IF EXISTS voxelpatch CASCADE;
CREATE TABLE voxelpatch (
  id SERIAL PRIMARY KEY,
  objID INTEGER,
  ifcID INTEGER,
  pa PCPATCH(1)
);
DO $$
DECLARE
    f record;
BEGIN
    FOR f in SELECT DISTINCT objID, ifcID
    FROM voxel
    LOOP
    INSERT INTO voxelpatch(objID, ifcID, pa)
    VALUES (f.objID, f.ifcID, PC_Patch(
    ARRAY(SELECT PC_MakePoint(1, ARRAY[x,y,z])
    as pt FROM voxel WHERE
    objID=f.objID AND ifcID=f.ifcID
)));
    END LOOP;
END;
$$
DROP INDEX IF EXISTS idx_voxelpatch CASCADE;
DROP INDEX IF EXISTS geom_voxelpatch CASCADE;
CREATE INDEX idx_voxelpatch ON
voxelpatch(objID, ifcID);
CREATE INDEX geom_voxelpatch ON
voxelpatch USING GIST(PC_EnvelopeGeometry(pa));
```

Listing 4. Example of generating PCPATCH layout representation.

	id [PK] integer	objid integer	ifcid integer	pa pcpatch
1	1	5	19	0101000000010000059DF0000...
2	2	4	3	0101000000010000000071A0000...
3	3	4	27	0101000000010000000F6000000...
4	4	1	8	0101000000010000020C20000...
5	5	4	20	0101000000010000074FB1200...

Figure 5. Example of a table created in PostgreSQL to store PAPATCH layout.

3.7 Relational Database Design

According to the above four kinds of data layout, we create a conceptual model including four voxel tables, one IFC semantic table and one building object table including scale and offset value for each object in Figure 6.

4. EXPERIMENTS

Within this section, we evaluate and analyse the query performance on voxels under different data layout in PostgreSQL/PostGIS database. In order to evaluate the balance between disk space and query time. We create corresponding tables with NUMERIC data type to directly storage real coordinates in EPSG 28356 projection. Due to space limits, we

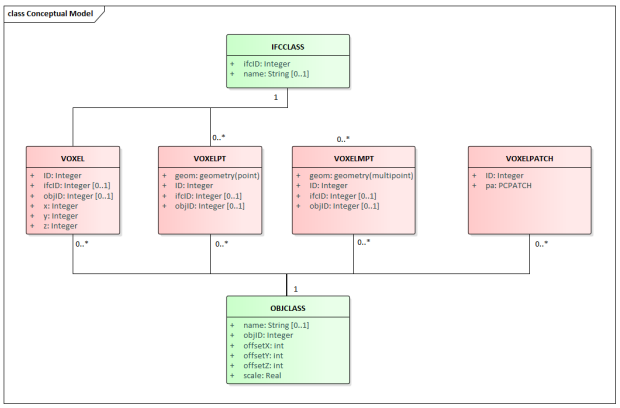

Figure 6. UML diagram for our conceptual design.

only show the operations on tables with INTEGER data type; the results on NUMERIC table will be compared.

4.1 Semantic Analysis

Firstly, the semantic analysis regarding to building object and IFC feature will be presented in this subsection. To make a fair comparison, the default query returns "*", which is all columns 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 `MULTIPOINT` 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 `MULTIPOINT` and `PCPATCH` runs much faster than the first two data layouts due to building object is segmented into only few partitions (i.e., few records in database).

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, `MULTIPOINT` and `PCPATCH` are more efficient when dong 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	SQL
ARRAY	`EXPLAIN ANALYZE SELECT * FROM voxel V JOIN objclass O ON V.objID=O.objID WHERE O.name='Built Environment';`
POINT	`EXPLAIN ANALYZE SELECT * FROM voxelpt V JOIN objclass O ON V.objID=O.objID WHERE O.name='Built Environment';`
MULTIPOINT	`EXPLAIN ANALYZE SELECT * FROM voxelmpt V JOIN objclass O ON V.objID=O.objID WHERE O.name='Built Environment';`
PCPATCH	`EXPLAIN ANALYZE SELECT * FROM voxelpatch V JOIN objclass O ON V.objID=O.objID WHERE O.name='Built Environment';`

Table 3. Load all voxels for particular building.

Semantic		ARRAY	POINT	MULTIPOINT	PCPATCH
Object	INTEGER	11,307.749	6,500.334	7.089	4.127
	NUMERIC	11,525.086	12,620.6	6.061	0.705
IFC	INTEGER	4,203.041	13,665.43	0.741	0.431
	NUMERIC	4,972.347	4414.142	0.24	0.428

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

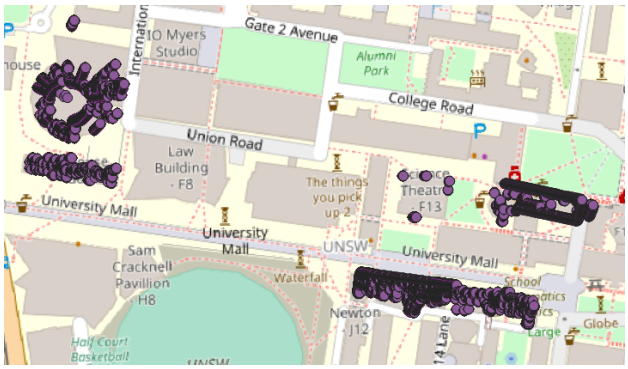

Figure 7. Visualisation of Eval-II via QGIS.

- *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:

```
CERATE VIEW box AS
SELECT ST_Translate(
    ST_Scale(ST_SetSRID(
    ST_MakeBox2D(
        ST_Point(minX, minY),
        ST_Point(maxX, maxY)
    ),28356), O.scale, O.scale),
    O.offsetX, o.offsetY) AS geom
FROM (SELECT objID,
    MIN(x) AS minX, MIN(y) AS minY,
    MAX(x) AS maxX, MAX(y) AS maxY
    FROM voxel GROUP BY objID) AS tmp
JOIN objclass O ON tmp.objID=O.objID;
```

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

```
CREATE VIEW box AS
SELECT ST_Extent(
    ST_Translate(ST_Scale(
        pa::geometry, O.scale, O.scale),
    O.offsetX, o.offsetY))::geometry AS geom
FROM voxelpatch V
JOIN objclass O ON V.objID=O.objID
GROUP BY V.objID;
```

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 gon-

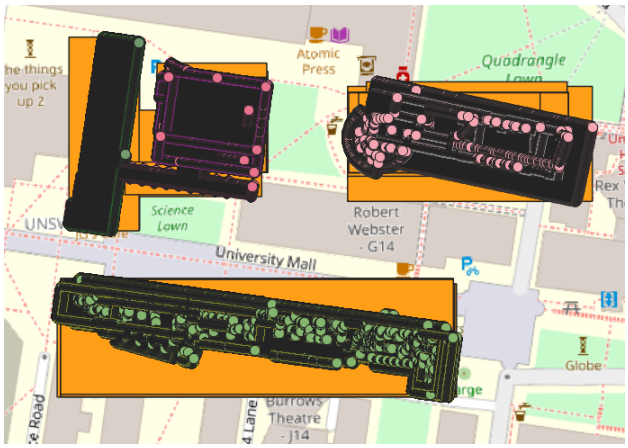

Figure 8. Visualisation of Eval-III via QGIS. Object painted in orange belong to the bounding box. The bottom green is "Built Environment".

Data Type	MULTIPOINT	PCPATCH
INTEGER	19,450.293	179,273.16
NUMERIC	5,443.022	36.529

Table 5. The query times based on two different data type using the dataset. All times are in milliseconds.

ing to search all rooms in "Red Cente" building whose height is larger than the roof "Science Theatr", the following query is conducted:

```
SELECT tmp1.z, tmp1.geom
FROM (
        SELECT ST_Z(geom) AS z, geom
        FROM voxelpt V, objclass O, ifcclass I
        WHERE V.objID=O.objID
        AND I.name='IfcSpace'
        AND V.ifcID=I.ifcID
        AND O.name='Built Environment'
) AS tmp1,
(
        SELECT ST_Z(geom) AS roof
        FROM voxelpt V, objclass O, ifcclass I
        WHERE V.objID=O.objID
        AND I.name='IfcRoof'
        AND V.ifcID=I.ifcID
        AND O.name='Science Theatre'
        ORDER BY roof DESC LIMIT 1
) AS tmp2
WHERE tmp1.z > tmp2.roof;
```

Listing 7. Script for Eval-IV.

To achieve this query, we first should acquire the roof height of "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 Postgis. 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 schemas, 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

Anderson, K., Hancock, S., Casalegno, S., Griffiths, A., Griffiths, D., Sargent, F., McCallum, J., Cox, D., Gaston, K., 2018. Visualising the urban green volume: Exploring LiDAR voxels with tangible technologies and virtual models. *Landscape and Urban Planning*, 178, 248–260.

Chamberlin, D. D., Boyce, R. F., 1974. Sequel: A structured english query language. *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, 249–264.

Chmielewski, S., Tompalski, P., 2017. Estimating outdoor advertising media visibility with voxel-based approach. *Applied Geography*, 87, 1–13.

Duncan, S. C., 2011. Minecraft, beyond construction and survival.

Fang, Y., Friedman, M., Nair, G., Rys, M., Schmid, A.-E., 2008. Spatial indexing in microsoft sql server 2008. *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 1207–1216.

Goncalves, R., Zlatanova, S., Kyzirakos, K., Nourian, P., Alvanaki, F., van Hage, W., 2016. A columnar architecture for modern risk management systems. *2016 IEEE 12th International Conference on e-Science (e-Science)*, IEEE, 424–429.

Guo, D., Onstein, E., 2020. State-of-the-Art Geospatial Information Processing in NoSQL Databases. *ISPRS International Journal of Geo-Information*, 9(5), 331.

Hane, C., Zach, C., Cohen, A., Angst, R., Pollefeys, M., 2013. Joint 3d scene reconstruction and class segmentation. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 97–104.

Herring, J. R., 2010. Opengis implementation standard for geographic information-simple feature access-part 2: Sql option. *Open Geospatial Consortium Inc*, 439.

Kothuri, R., Godfrind, A., Beinat, E., 2008. *Pro oracle spatial for oracle database 11g*. Dreamtech Press.

Laine, S., Karras, T., 2010. Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics*, 17(8), 1048–1059.

Li, W., Zlatanova, S., Diakite, A. A., Aleksandrov, M., Yan, J., 2020. Towards Integrating Heterogeneous Data: A Spatial DBMS Solution from a CRC-LCL Project in Australia. *ISPRS International Journal of Geo-Information*, 9(2), 63.

Li, W., Zlatanova, S., Yan, J., Diakite, A., Aleksandrov, M., 2019. A Geo-Database Solution for the Management and Analysis of Building Model With Multi-Source Data Fusion. *The International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 42, 55–63.

Momjian, B., 2001. *PostgreSQL: introduction and concepts*. 192, Addison-Wesley New York.

Obe, R., Hsu, L., 2011. PostGIS in action. *GEOInformatics*, 14(8), 30.

Patil, S., Ravi, B., 2005. Voxel-based representation, display and thickness analysis of intricate shapes. *Ninth International Conference on Computer Aided Design and Computer Graphics (CAD-CG'05)*, IEEE, 6–pp.

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

Zhou, Y., Tuzel, O., 2018. Voxelnet: End-to-end learning for point cloud based 3d object detection. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 4490–4499.

Zlatanova, S., Nourian Ghadikolaee, P., Gonçalves, R., Vo, A. V., 2016. Towards 3d raster gis: on developing a raster engine for spatial dbms. *ISPRS WG IV/2 Workshop" Global Geospatial Information and High Resolution Global Land Cover/Land Use Mapping" Novosibirsk, Russian Federation, 21 April 2016*.