A SCALABLE AND MULTI-PURPOSE POINT CLOUD SERVER (PCS) FOR EASIER AND FASTER POINT CLOUD DATA MANAGEMENT AND PROCESSING

Rémi Cura [AB], Julien Perret [A], Nicolas Paparoditis [A]

[A] Université Paris-Est, IGN, SRIG, COGIT & MATIS, 73 avenue de Paris, 94160 Saint Mandé, France
first_name.last_name@ign.fr
[B] Thales Training & Simulation SAS, 1 rue du Général de Gaulle 95523 Cergy-Pontoise, France

ABSTRACT:

In addition to more traditional geographical data such as images (rasters) and vectors, point cloud data are becoming increasingly available. Such data are appreciated for their precision and true three-Dimensional (3D) nature. However, managing point clouds can be difficult due to scaling problems and specificities of this data type. Several methods exist but are usually fairly specialised and solve only one aspect of the management problem. In this work, we propose a comprehensive and efficient point cloud management system based on a database server that works on groups of points (patches) rather than individual points. This system is specifically designed to cover the basic needs of point cloud users: fast loading, compressed storage, powerful patch and point filtering, easy data access and exporting, and integrated processing. Moreover, the proposed system fully integrates metadata (like sensor position) and can conjointly use point clouds with other geospatial data, such as images, vectors, topology and other point clouds. Point cloud (parallel) processing can be be done in-base with fast prototyping capabilities. Lastly, the system is built on open source technologies; therefore it can be easily extended and customised.
We test the proposed system with several *billion* points obtained from Lidar (aerial and terrestrial) and stereo-vision. We demonstrate loading speeds in the $\sim$*50 million pts*$/\mathrm{h}$ *per process* range, transparent-for-user and greater than *2 to 4:1* compression ratio, patch filtering in the *0.1 to 1* s range, and output in the *0.1 million pts*$/\mathrm{s}$ *per process* range, along with classical processing methods, such as object detection.

1. INTRODUCTION

The last decades have seen the rise of Geographical Information System (GIS) data availability, in particular through the open data movement. Along with traditional image (raster) and vector data, point clouds have recently gained increased availability (the site opentopography[1] is a good example) and usages (robotic, 3D, virtual reality). Sensors are increasingly cheap, precise and available. However, due to their massive un-organised nature (no neighbourhood information) and limited integration with other GIS data, the management of point clouds still remains challenging. This makes point cloud data barely accessible to non-expert users. Yet, many fields would benefit from point clouds, had they an easiest way to use them.

1.1 Problems

Point clouds data sets are commonly in the TeraByte (TByte) range and have very different usages; therefore, every aspect of their management is complex and has to scale.

Having such large data sets makes the compression an essential need. Not only is the compression necessary, but it also has to maintain a fast read and write access, and be transparent for the users. Indeed, we observe that, today, virtually all images and videos are compressed; most users not noticing it at all.

Similarly, so much data cannot (should not) be duplicated and must be shared, following a broader trend in the Information Technology world. Sharing data necessarily introduces concurrency issues (several users simultaneously reading/writing the same data).

Users usually need to access only a part of the data at once, thus efficiently extracting (filtering) a subset is important. With many varying usages, the criterae for choosing the subset may be volatile, and sometimes mixed.

Visualising something helps understanding it. In the case of multi-billion point clouds, a Level Of Detail

[1] www.opentopography.org

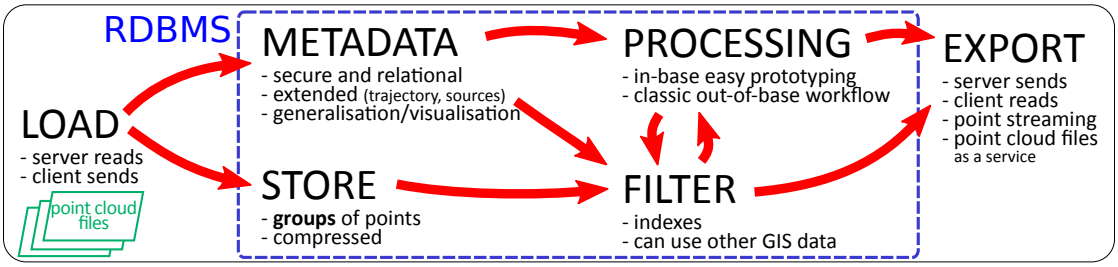

Figure 1: Graphical Abstract : In-base point cloud management pipeline in the Point Cloud Server (PCS).

(LOD) strategy is necessary, because the data set cannot be displayed in its entirety at once.

Features of point clouds can be very different depending on their source (Lidar, stereovision, medical, etc.), regarding the number and type of attribute, the geometric precision and noise, etc. Yet, point clouds usually are geospatial data, which makes them akin to vectors and rasters from the GIS world. Thus, point clouds may be used conjointly to other data types, either directly or by converting point clouds to images or vectors.

Lastly, point clouds are processed in many different ways suiting each user's needs. These methods must be fast and easy to design, scale well, and be robust.

Another important problem is related to point cloud management. For various reasons, point clouds are often handled as sets of points. Yet, a point cloud (data set) is much more than just points, as it also includes meta-data and other information such as sensor geometry, etc. Managing such data sets is difficult; like knowing which data sets are available and where. Because data sets are heterogeneous, managing extended meta-data such as point cloud coverage, date of acquisition and so on is also difficult, especially without a standard data format. Treating point clouds as only points is especially problematic, as is illustrated by a very recent benchmark release[2], which provides massive and very useful hand-labelled point clouds, yet does not provide any meta-data at all, neither extended meta-data nor contextual data.

In this article, we propose to use a point cloud server (PCS) to solve some of these problems. The proposed server architecture provides perspectives for meta-data, scalability, concurrency, standard interfaces, co-use with other GIS data, and fast design of processing methods. We create an abstraction layer over points by dealing with groups of points rather than individual points. This results in easy compression, filtering, LOD, coverage, and efficient processing and conversion.

[2] www.semantic3d.net

1.2 Related work

File system Historically, point clouds have been stored in files. In order to manage large volumes of these files, a common solution is to build a hierarchy of files (a tree structure, like a quad tree for instance) and access the data through a dedicated set of softwares. This approach is continuously improved (Hug et al., 2004; Otepka et al., 2012; Richter and Döllner, 2014) and a detailed survey of the features of such systems can be found in (Otepka et al., 2013). This approach is simple, and scaling is relatively easy (provided the Operating System (OS) maximum file number is not reached). However, using a file-based system has severe limitations. These systems are usually built around one file format, and are not necessarily compatible with other formats. Recent efforts have been made towards format conversion[3]. The features of such systems are very limited (limited meta-data handling, lack of integration with other GIS data, difficulty to use several point clouds together). Moreover, these systems are not adapted to sharing data and multi-users environments (concurrency).

DBMS for points Hofle (2007) proposed to use a Data Base Management System (DBMS) to cope with the concurrency issue. The DBMS creates a layer of abstraction over the file-system, with a dedicated data retrieval language (SQL), native concurrency capabilities (supporting several users reading/writing data at the same time), and the wrapping of user interactions into transactions that can be cancelled in case of errors. DBMSs have also been used with raster and vector data for a long time, and the possibility to define relations in the RDBMSs (Relational DBMS) offers a simple way to create robust data models, and deal with meta-data. Adding the capability to create point clouds as services, DBMSs solve almost all the problems we face when dealing with point clouds. Usually, the database stores a great number of tables, and each table storing a point per row (Lewis et al., 2012; Rieg et al., 2014). Such a database can easily reach billions of rows. Nevertheless, storing these many rows is problematic because DBMSs may have a non-negligible

[3] http://www.pdal.io/

overhead per row. Moreover, indexing such a number of row is slow and takes a lot of space, and the possibilities for compression are limited.

Column store database and No-SQL These limitations are more generic than point clouds, and apply to any massive amount of data which is weakly relational and does not get updated often. As such, they have been researched and inspired the concept of column-oriented databases, such as MonetDB[4]. This database system is used to store individual points (Martinez-Rubi et al., 2014, 2015; van Oosterom et al., 2015). This approach is effective to store large amounts of row without much overhead, and also solves most of the indexing issues. However, points are not compressed, integration with other GIS data is weak, and scaling to multiple computers is not straightforward. In parallel, stripped down column stores were proposed, having been specially tailored for massive and weakly relational data, forming the No-SQL databases. They scale extremely well to many computers and can deal with large amounts of data (Wang et al. (2014) and SpatialHadoop[5]). However, this comes at a price. Indeed, NoSQL databases must drop a few guarantees on data. At the moment, they are not integrated with other GIS data and have much less functionalities. Indeed, NoSQL databases are closer to being a file-system distributed on many computers (with efficient indexing) than being DBMSs. Thus, massive scaling still necessitates specialised hardware, and the people to maintain it.

Cloud Computing A recent possible workaround for this issue is to use Cloud Computing facilities [6] to store the points, like Amazon S3. In this solution, data storage is offered as a service and externalised. This may provide the ultimate scaling, but suffers from the same limitations as the NoSQL, with open issues on indexing.

DBMS for patch All the previous data management systems try to solve a very difficult problem: managing a massive quantity of individual points. Solutions that scale well must focus on data storage and retrieval, and drop the rest of the management problem (feature, meta-data, integration, processing). Another recent approach is being explored in pgPointCloud (2014) and other commercial RDBMS. The key idea is to manage groups of points (called patches), rather than points, in a RDBMS. Creating this abstraction layer over points allows retention of all the advantages of a RDBMS, but keeps the number of rows low, thus avoiding the associated scaling difficulties (index, compression). Moreover, the proposed abstraction offers new theoretical possibilities, because it creates a generalisation of the groups of points. The price is that, in order to access a point, its whole group has to be accessed first, and so the way points are grouped must be compatible with the intended point usages.

In this work, we present a point cloud management system fully based on pgPointCloud (2014) and other open source tools. We test the main aspects of this point cloud management system to prove that it answers the common needs of point cloud users, as illustrated by Figure 1.

1.3 Plan

Following the IMRAD format (Wu, 2011), the remainder of this article is divided into three sections (Method: § 2., Result: § 3. on page 10, Discussion: § 4. on page 19).

Each section has the same organisation covering the bases of point cloud usages (See Figure 1 on the previous page). First, we consider how points can be stored as groups in a Point Cloud Server (Storing). Then, we consider how to load point clouds in the PCS (Loading). Point clouds contain meta-data that can also be stored and used (Point Cloud Context). We study how to access only a part of the points using conditions (Filtering). Points can be outputted from the PCS (Exporting). Last, we consider various methods to exploit points (Processing).

Thus, each subsection is found in method, results, discussion with the same subsection numbers.

2. METHODS

2.1 Storing groups of points in a RDBMS

The proposed solution relies on a PostgreSQL (2014) RDBMS server using the PostGIS (2014) and pgPointCloud (2014) extensions. The key idea is to store a point cloud per server table, with one row storing a compressed group of points. Groups of points are called patches of points. The type of a point (attributes size, definition, nature) is defined in a global XML schema. See Figure 2 for an overview of how storage is organized in PCS.

The user can load data into the server using several common means (major programming languages, Bash, SQL, Python), from any format of point cloud that can be expressed as a list of values. Point clouds are stored without loss and are compressed. Sophisticated database indexes allow efficient filtering of the patches. Point clouds can be used with vector and rasters and other point clouds. Meta-data are integrated and exploited. Furthermore, point clouds can easily be converted into other GIS data (vector/raster). Processing methods are directly accessible within the database; additional methods can be added externally or internally. Accessing points from the database is also easy and can be done in several ways (whole files, specific points and streaming).

[4] www.monetdb.org
[5] http://spatialhadoop.cs.umn.edu/
[6] https://github.com/hobu/greyhound

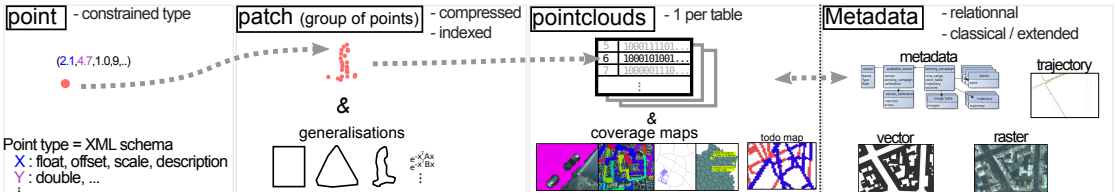

Figure 2: PgPointCloud storage illustration. Point attributes are described by an XML schema. Points are grouped in patches, indexed and compressed, which may have several generalisations. A point cloud is stored in a table, with one patch per row, along with other tables generalising the point cloud (like coverage map). The PCS also stores meta-data (date, place) in a relational way, extended meta-data like trajectories, and possibly other GIS data like vector and rasters.

2.1.1 Storing groups of points rather than points

Briefly, storing groups of points offers the advantages of generalisation (potentially more complex semantic objects), reduces the number of rows in the database by several orders of magnitude, reduces index size, allows efficient compression, and offers a common framework for different types of point clouds coming from different sources. Working on groups of points separates the filtering and retrieving of points. This allows to take decisions based on filtering results before retrieving points. For instance, based on the density, an optimal LOD can be automatically chosen. Groups can also easily be split or fused at any point after data loading.

It is important to note that storing groups of points rather than points also introduces a fundamental limitation: to obtain an individual point, we need to get the full group first. This means that the grouping points approach is only possible when points can be categorised into groups that are coherent for the intended applications. Incidentally, all intended applications must require the same grouping rules.

Generalisation Choosing to use groups of points instead of individual points amounts to use a generalisation of the data, that is an abstraction. Abstracting the data is very common in GIS. For instance, when making a map of a large city, representing all individual building footprints would diminish the user understanding of the map. Instead, building footprints may be aggregated to form urban blocks (see Mackaness et al. (2014) for a recent introduction to the generalisation topic).

Regarding point clouds, we may have a group of 10k points sampled along a small part of the road that is flat (10k 3D points). For some application, we could abstract the group with a plane (three 3D points). Geometrically representing this group of points by a plane reduces storage, but the change is more profound, because the plane is another representation of the underlying object that has been sensed.

The plane could be used as part of a facade reconstruction Lafarge et al. (2013), or even be the base for a further building generalisation Meng and Forberg (2007).

The generalisation does not have to be geometric. For instance, a group of points can be abstracted by statistical distributions (similarly to Preiner et al. (2014), although they use the distribution for surface reconstruction).

Ummenhofer and Brox (2015) illustrate both uses. They use an octree and thetrahedrals as support for their geometric generalisation, and aggregates as a statistical representation. Combining both, they can reconstruct surfaces without using the points but only their generalisations.

Such generalisation is by essence highly tailored to an usage, being a form of information-loosing modeling. In this article, we propose several generalisations adapted to a variety of usages (Figure 19 page 16). Those generalisations can be used conjointly in the Point Cloud Server. By doing so, we avoid the pitfall of duplicating the data for each specific usage.

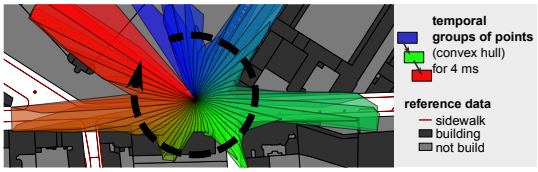

Figure 3: Rotating Lidar (Velodyn) with strong temporal dependency (200 ms acquisition).

2.1.2 Point grouping strategy

Points should be grouped with regard to how they will be retrieved afterwards. As points tend to be retrieved by their spatial position (spatial-grouping), grouping the points that are spatially close together makes sense. Some Lidar devices include a strong time dependency, and are commonly used to detect moving objects. In this case, time-grouping may be interesting, in order to easily differentiate between points roughly at the same position, but acquired at varying time (See Fig. 3, and Section 3.1 and Fig. 14 p. 12). Grouping

rules may also mix spatial and temporal rules, as well as other rules like semantic grouping if this information is available (for instance, points pertaining to buildings would be in separate groups than points pertaining to the ground).

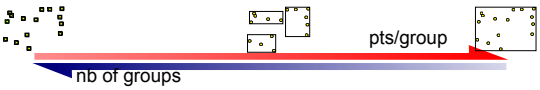

Figure 4: Choosing group size is a trade-of between filtering and storage.

While many rules are possible to group points, it always results in a trade-off (see Figure 4). Having small groups usually means many groups, which is bad for storing, because it will increase the number of rows, thus the size of the indexes and associated overhead, and reduce compression possibilities. On the opposite, having large groups is bad for filtering (and maybe compression, if points becomes too dissimilar), because, to get one particular point, the whole group has to be read.
However, we stress that all groups do not have to follow the same rules, thus group size can be adapted to local characteristics of the point cloud, for instance to geometry (grouping depending on density) or semantic (grouping depending on attributes or classification). See Section 3.1 for an example of varying grouping size based on geometry, where groups are merged/split in 8 (similarly to voxels) until the target number of points per group interval is met.

2.2 Loading

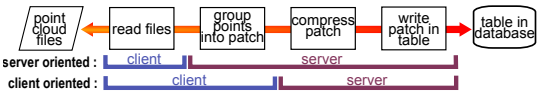

Figure 5: Processes necessary for load (and output) of points into the database can be performed by the client(s) or the server, depending on the application.

Writing data in a PostgreSQL RDBMS is standard. Clients exist in all major programming languages. Because DBMSs are built for concurrency, all the presented methods use parallelism.

For the specific application of writing point clouds, the goal is to go from point cloud files to (compressed) patch of points stored in tables inside the database (See Figure 5). To this end, several intermediary steps have to be performed. In a server/client architecture, we conceptually separate solutions depending on who is supposed to cover most of the process. In a "Server oriented" design, the server does almost everything. In a "Client oriented" design, the client does almost everything. Please note that this division is only conceptual.
Section 3.1 page 11 contains more details about how patches are compressed.

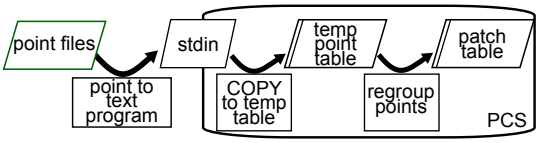

Figure 6: Conceptual schema for parallel loading.

'Server oriented loading' Our first loading method (Figure 6) reads point cloud files, converts them to a stream of attributes and writes them to temporary tables in the database. The database groups points into patches and adds the patches to the final point cloud table. Please note that the database could directly read point cloud files.

Distributed 'Client oriented loading' In the previous method, the database performs the grouping of points into patches and the actual writing of patches into tables. We could lessen the workload of the server by allowing the client to do the grouping.
We design a 'client oriented' (Python) loading method. It is similar to the method adopted by the PDAL[7] project, which we also test here. The clients read point cloud files, group points into patches, and send the patches to the database. The database compresses the patches and writes them into the final point cloud table. Please note that the client could also perform the compression.

2.3 Point Cloud and Context

Historically, RDBMS databases have been designed to create and maintain relationships between data. Because our method relies on a RDBMS, we can leverage this capacity. One of the goals of our system is to manage point clouds rather than points. This way, a point cloud is not considered a set of points, but rather a set of points associated with various meta-data and contextual information (and maybe processing methods). In particular, our system can store the full meta-data model, as well as more indirect meta-data like the trajectory of the sensor. Meta-data can also be organised in a relational way in order to be consistent between different point clouds. By integrating point clouds into a relational database, and having several representations for patches (See the generalisation concept, in Section 2.1.1), we can easily create coverage maps. It also enables to use several point clouds together as well as mix point clouds and other GIS data (raster and vector), directly or after converting point clouds to other GIS data types.

[7] www.pdal.io

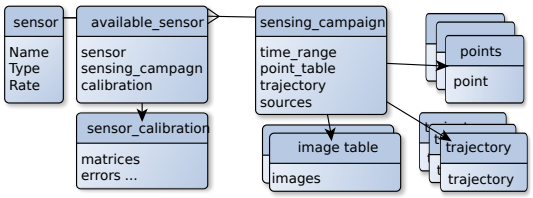

Figure 7: Example of a data model to store meta-data. See Hofle (2007, p. 15) for a real model.

Managing metadata The point cloud server offers the perfect framework to regroup all the meta-data concerning the point cloud. For instance, the popular .las file format proposes to store a project id, date of acquisition, and name of the hardware. Being based on standardised and limited fields, very few meta-data are stored. Furthermore, the information can be missing or erroneous. Using a RDBMS, it is possible to create an unlimited relational model of the meta-data and to easily (and automatically) enforce it. For instance, instead of a project id, we could refer to a list of projects, each having an associated list of partners, start and finish dates, associated authorisations, etc. Instead of the name of the hardware, we could refer to a list of hardwares, having each different configurations, typical precision of the sensor, point type, methods for reading the raw data, etc. See Figure 7 for a basic example of meta-data schema. We stress that, in fact, several point clouds meta-data are already related to each other in an implicit way. For instance, the date of acquisition of a point cloud can be implicitly related to the date of acquisition of another point cloud.

The benefits for point cloud management are numerous, from simple, such as looking for point clouds based on those meta-data (e.g. find all point clouds in a given area with a given density, acquired in a given time range, whose geometric error is less than 1 cm), to much more complex, such as on-the-fly re-registering of the point clouds when the estimated sensor position is updated.

Such meta-data could also be used in the filtering step. For instance, for an application relying on colour, the user would be able to automatically get points acquired through stereo-vision and not through Lidar.

In a more generic way, we point to the success of the Resource Description Framework (RDF) in the last decade as a sign that meta-data management is important and expected by users.

Coverage map Using the server, we create meta-data-like point clouds coverage maps (Figure 17, 18, page 14) , which are essential to quickly understand point clouds coverage , similarly to (Lewis et al., 2012, Figure 8), or Youn et al. (2014). The idea is to have several representations of a point cloud. The 2-level representation would be to represent the area covered by the point cloud by polygons at a detailed level (large scale). For performance and map-generalisation reasons, the point cloud could also be represented by a single point when viewed from afar (small scale)

With this set of maps, one can instantaneously and easily check what type of point cloud is available in a given area using a colour code (for instance). Because the Point Cloud Server mixes GIS and point clouds, going a step further than the two-levels visualisation (a point at small scale, detailed polygons at large scale) toward a 4-level visualisation is natural. More generally, mixing Remote Sensing data and GIS data enables much more advanced applications (Aubrecht et al. (2009)). See Section 3.3 for full details on coverage map creation.

Extended metadata We can extend the classical notion of metadata a step further and consider that it also concerns the raw information that was used to create the point cloud. For Lidar point clouds, this would be the trajectory and position of the sensing device, along with the raw sensing files. For stereo point clouds, this would be the camera poses for every image used to construct the point cloud, along with the images. This information can be stored in the server, and leveraged in filtering (see Section 3.3), processing and uncertainty management (for instance registration).

Using several point clouds and other GIS data Point clouds are created by different sources, like stereo-vision, aerial Lidar, terrestrial Lidar, RGBZ device (Kinect), medical imaging devices (MRI), etc. The Point Cloud Server mixes all these data, along with other GIS data (rasters and vectors). Vectors and rasters are stored and exploited using PostGIS (2014). We can use georeferenced point clouds together, thus exploiting their complementarity.

In a typical scenario, a user interested in a place queries the Point Cloud Server. He automatically obtains points from the several point clouds available at this place, for instance a low resolution, large coverage 1 pt/m^2 aerial Lidar point cloud, complemented by a more detailed but very local 10 $\mathrm{kpts}/\mathrm{m}^2$ stereo-vision point cloud.

Point cloud as raster or vector In the spirit of generalisation (see Section 2.1.1), it is advantageous for some applications to convert points to other GIS data types, such as raster or vectors, directly within the database. We propose several in-base groups of points vector representations, such as bounding box, oriented bounding box, concave envelope similar to alpha shape (Edelsbrunner et al., 1983), and bounded 3D plans (Figure 19). These representations can be used to extract information at the

patch level, accelerate filtering, enable large scale visualisation, etc. We also propose two in-base means to convert points to multi-band rasters by a Z projection.

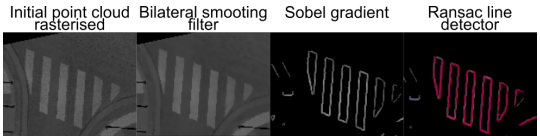

Figure 8: A part of a point cloud is converted to a raster. We use bilateral smoothing, gradient (Sobel), and line detection (RRANSAC by Chum and Matas (2002)) to reconstruct the pedestrian crossing. These operations are much faster and easier on rasters rather than points.

Rasterisation is a common first step in the literature because neighbourhood relationships are explicit between pixels, unlike points. Figure 8 illustrates how a first conversion to raster allows to use powerful and classical image processing methods to extract information from point clouds.

Point Cloud patches as Graph / Topology The specificity of point cloud is to not embed neighbourhood information. Yet, graph analysis offers powerful tools. We propose to take advantage of pgRouting (2015), which is a PostgreSQL extension that adds basic graph functions.

We can build a weighted graph embedded in 3D over the groups of points (ie. a graph whose vertices are the groups of points and edges the neighbouring relations between those groups, while the weight of an edge is the 3D distance between centroids of groups of points). (See Figure 9)

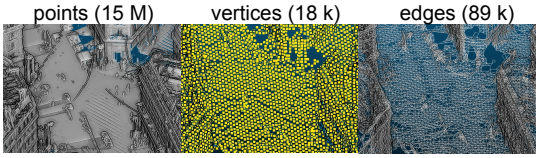

Figure 9: Building a graph embedded in 3D over point groups. Up-Left: the original part of the point cloud. Up-Right: the node of graph (centroid of patches). Down-Left: graph edges: the adjacency relationship between patches. Down-Right: visualisation in GIS.

This graph is in fact a very rough approximation (up to the patch size) of the point cloud surface. We can leverage it for fast geodesic distance computation along this surface.

Orthogonally, we can build a simplified graph and use it as the starting point of road network reconstruction. Road network reconstruction is a large topic with widely different types of strategy (see Quackenbush et al. (2013) for an introduction), we only use this topic to show the PCS capabilities, (see Section 3.3 and Figure 24 page 15).

2.4 Point Cloud Filtering

Point clouds are big; yet, we often need a very small part of them (Figure 10, parameters in 3.3 page 14). Thus, the capacity to filter a point cloud is essential for many uses. Acceleration structures (commonly called index) are the accepted solutions. This essentially creates indexes of the data to accelerate searches. Octree, B-tree, R-tree, and Morton-curves are popular acceleration structures. Designing and optimising these indexes is a major research subject (see Kiruthika and Khaddaj (2014), for instance) and is also the main designing factor in point cloud management systems.

Filtering strategy Because our system stores patches (groups of points), we can separate the filtering and the retrieving of data. The strategy is then to first efficiently filter data at the patch level by rejecting most of the patches (reducing points from billions to millions, for instance), then, if necessary, further filter the remaining points.

Indexing Our system extensively uses n-D indexes (BTree, RTree) that are native to PostgreSQL. We index patches (not points), and we expect that there are a few million patches. Basically, these indexes answer in the $0.01to1srange$ to any filtering queries (depending on the number of patch filtered), such as 'What are the patches with $f(patch)$ between .. and ..'; provided that $f(patch)$ is appropriatly indexed. $f(patch)$ can be anything, a spatial position, an attribute of the points contained in the patch, a function, etc.

Indexes of functions are very powerful and can save a lot of space (no need to add an extra column to store the value of f for every row). For instance, we may have a fast function f that gives a measure in $[0; 1]$ of how much the patch looks like a vertical cylinder. Now, when looking for all the patches p that really resemble cylinder ($f(p) > 0.8$), for instance), we do not need to recompute f each time for every patch, nor store all values of f. The server will only stores simplified f values (typically stored on fewer bits) within the index, uses it to remove the vast majority of useless patches, then recomputes f for the few remaining patches that are good candidates.

PostgreSQL also determines whether to use available indexes or not, and in which order. This feature may prove essential. Indeed when accessing a large amount of information, it will be faster to simply read all the data rather than use the index (sequential vs. random access).

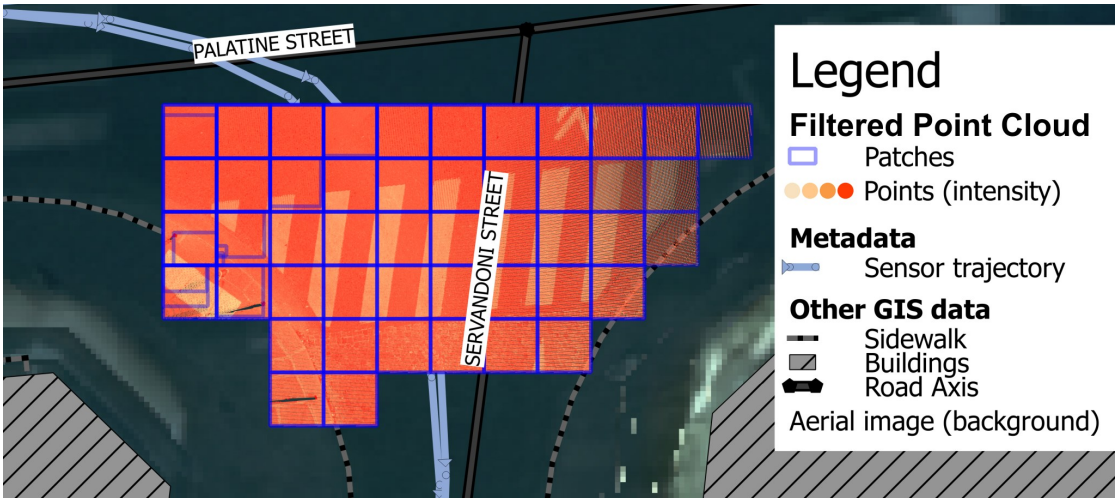

Figure 10: Illustration of a filtering condition presented in Section 3.3 p.14. Among billions of points, only those respecting complex filtering conditions are found in ~ 0.1s. Results are shown in QGIS.

Filtering on :	SQL query
	SELECT gid, patch FROM my_patches WHERE
Spatial position (using any geometry)	ST_Intersects(patch::geometry, ...) = TRUE AND
attribute of patch (density)	Pc_NumPoints(patch) BETWEEN 10 AND 100 AND
attribute of patch (source file name)	file_name ILIKE E'file_.*2.ply' AND
attribute of points in patch (intensity)	rc_range(patch, 'intensity') && numrange(0,1.5) AND
Spatial position (using another vector layer)	EXISTS (SELECT 1 FROM buildings AS b WHERE ST_Intersects(patch::geometry,b.geom))

Figure 11: Example of a filtering query on patches with various types of filtering conditions.

This decision is automatically made based on statistics on tables and a genetic optimisation engine[8]. The database can actually estimate how many rows will be needed by a query. Then, knowing the cost of reading one row via index (random access), and the cost of reading the whole table (sequential access), it can decide which strategy to choose. The genetic optimisation engine is used to choose how the query will be executed, using join, index, inner loops, etc.

Figure 11 illustrates a filtering query combining various conditions.

2.5 Exporting

Similarly to Section 2.2 (Input, page 5), we divide the output methods into two categories. The first family of solutions is when the server performs most of the output processes (”Server oriented exporting”), for instance writing the points in a file, or letting the user access the points through queries.

The second family of solutions is when the clients perform most of the output process (”Client oriented exporting”). We can also see the output as a service, be it for classical GIS software (using the lens), or for WebGL browser.

2.5.1 Client oriented export

Client oriented: Distributed export We also designed a Python method to perform massive parallel export. Similarly to Section 2.2 (Input), the goal is to reduce the work done on the server and increase the work done on the clients. In this version, the server sends raw binary uncompressed patches (groups of points), and the transformation to points is done by the client(s). This is similar to PDAL workflow.

[8] www.postgresql.org/docs/current/static/geqo-pg-intro.html

2.5.2 Server oriented export

Server oriented: PLY File As a Service (PLYFAS) Our system can be used transparently with a file-based workflow. Indeed, users may already have legacy processing tools that work with files. Of course, these tools could be easily adapted to read points from the database and not from files, but users may want to use their usual tools as-is. For this case, we propose PLYFAS, an easy mean to export points from the database and create a .ply file (please note that PDAL could also be used to export PLY files). The user can use the small PLYFAS API to request the database to create a ply file from any set of points. The user may simply want one of the exact original point cloud files that were loaded into the point cloud server (mimicking a traditional workflow). However, the user has also access to much more power and can request a file with filtered points by any means introduced in Section 2.4, or with the additional processing results of Section 2.6. For instance, the user can request all the points in a given area that have been classified as 'building parts' with a given confidence, and that were sensed during the second week of March 2014. In addition, the user can ask to get a target point density (Level Of Details), and to get deduplicated points (duplicated points are removed from the result), etc. Some of this operations are easy to perform in SQL (See for instance Section 3.3 page 17).

Server oriented: Using PostgreSQL drivers/connector PostgreSQL can be accessed using many programming languages, thus any PostgreSQL driver can be used to connect to the server and output points. This workflow is very similar to what a classical processing program would do. The classical *'open point cloud file, read points, do processing, write results'* becomes *'connect to server, read points, do processing, write results on the server or elsewhere'*.

By using the server to access points, the user gets additional capabilities. For instance, the user does not have to read a whole file (or any files) if he is interested in only a few points. Using the point cloud server, the user can directly filter the point cloud to obtain only the desired points, and even use in-base processing or LOD to further limit the points obtained. Furthermore, the PCS can be accessed concurrently by several users, facilitating parallel processing, and more notably parallel writing (for the results for instance) in the server.

Server oriented: Lens for traditional GIS Point clouds are best visualised in dedicated software. Yet, point clouds are also geospatial data, and benefits much from being visualised and analysed in powerful GIS tools (like QGIS). However those tools do not scale over the Million points range. We propose a "lens" that reveals the points it covers (Figure 12)). The idea is that a user moves a polygon acting as a "lens" over a place of interest in the map, revealing the underlying points, using any GIS client that can access the database. The concept has already been used to improve an interface (See Bier et al. (1993); Lobo et al. (2015)), Pindat et al. (2012) also proposed a lens with varying shape. Using triggers and a view, we ensure that the points are updated when the lens changes. Moreover, the lens also allows to choose the density of points it displays (using Level Of Details), and the vehicle pass we are interested in (temporal filtering). This feature is necessary, because the registration error between several passes may be a problem (See Figure 12).

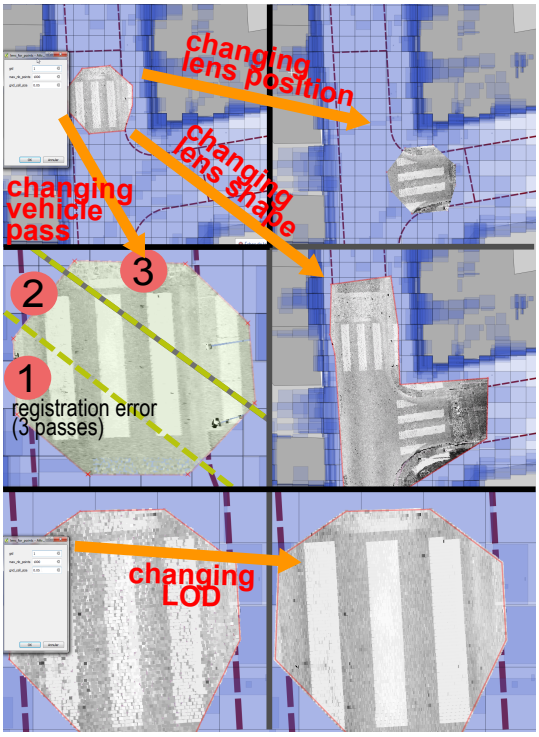

Figure 12: A polygon representing a lens that reveals the points underneath it (among billions), with a given Level Of Detail and vehicle pass. Points are automatically updated when the lens is modified by the user, using a pure in-base solution, which makes it compatible with any client.

Server oriented: Asynchronous point cloud streaming to browser The last output possibility is to stream points in a web context. The goal is to display a point cloud into a web browser with background loading (*i.e.*, the points are displayed as they arrive, the user keeps browsing and the loading is non-blocking).

For this, we use a Node.js server between the client and

the Point Cloud Server, which enables non-blocking interactions. We stress that from the PCS perspective, the task is standard (give points that are at a given place), again possibly taking avantage of LOD (send only the necessary number of points, and not all the possible points, which may be critical for limited-hardware situation like mobile phone or tablets). We use an implicit LOD scheme which is described in Cura et al. (2016) (working paper).

2.6 Processing Point Cloud with the Server

Processing point clouds We think it is important to offer both point clouds and adapted tools to users (leaning toward giving access to services). Indeed, for most of the users, a point cloud is only a mean to get another information via processing.

Our system can be used for processing in two ways. The most classical is out-of-base processing. A client obtains the points, does something, and writes the results in the server or elsewhere.
However, our system also offers in-base processing. In this form, the user directly adds processing methods within the PCS. Processing methods become very close to the data and can be reused or combined to create more complex methods. The client does not have to install anything (all methods live within the server), which is more practical (version management, dependencies ...).

An advantage of in-base processing with the PCS is that it is easy to add new processing methods. These methods can be written for efficiency (C, Cpp) or using high level languages (Python, R) for very fast prototyping. We determined that the most useful in-base processing functions should be fast and simple. This way, the newly written functions can be used in other aspects of the point cloud server, such as indexing, or be combined directly in SQL queries. For instance,

```
SELECT classify(extract_plan(patch),
    extract_feature_1(patch), ...)
  FROM patches
  WHERE compute_verticality(patch)>0.8
```

Of course both type of processing can be used conjointly. In the previous scenario, the classifier would be trained outside of PCS for better memory and performance management.

3. RESULTS

3.0 General System Test

We design several experiments to test all thecomponents of the point cloud server on several datasets (Figure 13).

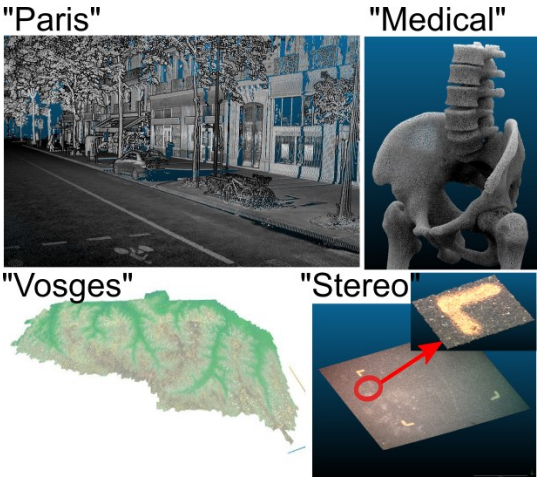

Figure 13: Paris data set (terrestrial Lidar), Medical Imaging data set (X-Ray CT Scan), Vosges data set (aerial Lidar), Stereo data set (Stereovision)

All the experiments have ample room for optimisation, and can easily be reproduced (using only open source tools). We use PostgreSQL 9.3, PostGIS 2.2, PgPointCloud 1.0, Python 2.7, PDAL 1.1, Numpy 1.10 and Scipy 0.17. We stress that all the facts should be indicative at best, because dealing with massive data introduces a strong hardware factor. For instance, the same loading method (parallelized PDAL) used in the reference benchmark van Oosterom et al. (2015) is 4 times slower with our hardware. Moreover, the PCS uses several layers of caching whose influence may blur the results.

Result at the system level Overall, we load several Billion points into the PCS, perform several processing in and out of base (second to hour), extensively use simple and complex filtering (response time from millisecond to second), convert points to images and vectors, and output points ($\geq$ 100k pts/s). The entire system works as intended and is efficient and powerful enough to be used in research settings.

Data sets used For this work, we mainly use four data sets (including IQmulus (2014)) illustrated in Figure 13. They were chosen to be as different as possible (sizes from millions to billions of points, sensing from active to passive, wavelength from a few nanometres to Near Infrared, nature of sensing from surface to volumetric, different numbers of attributes from none to 21) to further evaluate how the proposed methods can generalise on different data. We emphasize that the Vosges data set is a massive aerial Lidar point cloud covering mountains and forests.

Hardware We tested all our methods on two settings corresponding to two target users. The first use case is

Table 1: Figures of the Point cloud data sets used in the experiments.

data set	Type	Nb. of points	Nb. of original files	Spatial coverage	Nb. of attributes	Typical spacing	File Type
Vosges	Aerial Lidar	5.2B	∼1450	1330 km^2	9	1 m	.las (bin)
Paris	Terrestrial Lidar	2.15B	∼ 750	42 km	21	1 cm	.ply (bin)
Stereo	Stereovision	70M	16	3 m^2	6	0.1 mm	.ply (bin)
Medical Imaging	Medical Imaging	1.95 M	1	20 dm^3	0	0.6 mm	.ply (bin)

a non-specialised user with non-dedicated hardware. To this end, the setting is simple and portable (the point cloud server is hosted on a virtual box on an external drive). The second use case is for a specialised user, with dedicated hardware. The setting is powerful and offers much more storage space (dedicated server with 12 cores, 20 GB RAM, SSD for OS, regular disk for storage, Ubuntu 12.04). Timings are indicative because of influence of caching and configuring.

3.1 Storing groups of points in a RDBMS

Table 2: Creating and indexing patches for the test data set.

data set	Grp Size	Grp Dim	Patch nb (k)	Avg pts/patch (kpts)	Patch index size(MB)
Vosges	50m	2D	580	8.9	27+15
Paris	1m	3D	6570	0.325	300+150
Stereo	$\frac{1}{250}$m	2D	180	0.4	12+3
Medical	$\frac{1}{100}$m	3D	4.8	0.4	0.4

Table 2 gives the results on the grouping and indexing aspects.

Point grouping strategy Points must be categorised into groups that will make sense for subsequent uses of the point cloud. Groups of points must be big enough so the number of rows is tractable, but not too large because getting only one point still necessitates obtaining the entire group. We designed these grouping rules with two criterias. First, the number of rows is less than a few millions so that the index fits in the server memory (Table 2). Second, the range of number of rows is still manageable for classical GIS software (e.g. QGIS). We can afford to have arbitrary large groups as a result of the PostgreSQL TOAST[9] storage system.
Grouping is done at data loading, but we can change the groups and grouping rules at any time. Index creation is very fast (a few seconds to a few minutes), and the index size is ≤ 1 % of the point cloud size.

Storing patches and not points Indexes are built on patches and not on points, and thus are several orders of magnitude smaller and much faster to build. We estimate the size of indexes if we were to store one point per row rather than one patch per row. For this, we measured how PostgreSQL index size and build time evolves with the row number. The results are mostly linear (tested from 10 kpoints to 10 Mpoints), as seen in table.

XYZ points table size	RTree index size	RTree index building time
65 MB/Mpts	52 MB/Mpts	18 s/Mpts

Using this scaling behaviour, we estimated the spatial index size if the point were not stored by groups but individually (one point per row). Without surprise, index size and built time would become intractable if points where stored one by one and not by groups.

data set	Estimated point index size (GB)	Estimated point index build time (h)
Vosges	2600	290
Paris	1000	120
Stereo	35	4
Medical	132	1 min

Compressing point clouds Patches are compressed before storage using the dimensional pgpointcloud compression[10]. Table 3 compares loaded data set space occupation on the server with the original binary files on the disk (See Table 1 for original file format). In our case, patches are compressed attribute-wise, with either a run-length, common bit removal, or zip strategy. This means that, for each patch, each attribute (dimension) is compressed independently using the strategy which is deemed optimal for this attribute. The compressing efficiency widely varies depending on the data and the type of points attributes. As a comparison, a tool specialised on .las file compression like the one proposed by Isenburg (2013) achieves a *8.3* compression ratio on the Vosges data set. It uses a more adapted delta encoding approach for XYZ and time and does not compress the attributes.

Compressing and decompressing data introduces an overhead on the data access. We estimate it by profiling the uncompress and compress functions. Again, the overhead is dependent on the type and number of attributes. For instance, stereo contains double attributes that are

[9] http://www.postgresql.org/docs/current/static/storage-toast.html

[10] https://github.com/pgpointcloud/pointcloud#compressions

Table 3: Analysing compression ratio and compression/decompression speed.

data set	Points	Disk size	Server size	Compression ratio	Comp. Speed M pts/s	Decomp. Speed M pts/s
Vosges	5.2B	170 GB	39 GB	4.36	4.49	4.67
Paris	2.15B	166 GB	58 GB	2.86	1.11	2.62
Stereo	70M	1 GB	490 MB	2.05	2.44	7.38
Medical	2M	23 MB	7.7 MB	2.98	7.66	25.8

compressed with the zip strategy, which is slower in compression. See Table 3 for the result.

Spatial or temporal grouping In this experiment, we use two different grouping methods on terrestrial Lidar data. This type of Lidar (Velodyn) rotates around Z axis at 10Hz (see Figure 3 on page 4 for one rotation), and is commonly used to perform object tracking (see the work of Azim and Aycard (2012) for instance).
In such a case, the main filtering aspect may be temporal, and not only spatial. In the temporal scenario, we group points acquired every 4 ms together, and display the convex hull for easier visual understanding. In the spatial scenario, we group the points with a $1\mathrm{m}$ grid.
Without surprise, temporally-grouped patches have a more regular number of points, whereas the spatially-grouped patches have a much more diverse density (see the histogram of Fig. 14). In both cases the compression is similar, as the filtering time.

In the Spatial grouping, knowing precisely the sampling rate (10Hz), it is then easy to get points that are sensed during a turn n, but not before or after. This capability would be the basis of object/change detection. .

Adapt patch grouping rules In our solution, points are grouped at loading time with fixed rules. This system is well adapted to point clouds with homogeneous density, like aerial Lidar. However, these fixed grouping rules are not optimal for terrestrial Lidar, where the point density varies strongly based on the distance to the Lidar device.

We propose a mechanism to change the patch grouping rules on loaded data sets. The user fixes a target patch density depending on the expected number of rows and expected usage of the point cloud. In the example, we target a density between 0.5 and 2 kpts $/\mathrm{m}^3$. We then iteratively split/merge patches to try to meet this target. Figure 15 illustrates in 2D and 3D views of patches of different size but containing roughly the same number of points. With the proposed targets, the global number of patches is roughly the same, with the benefits of having less frequently too small or too big patches, which particularly shows in the histogram of the points per patch for fixed size and variable size patches (16).

This adaptive grouping size also reduces the global quantization error.

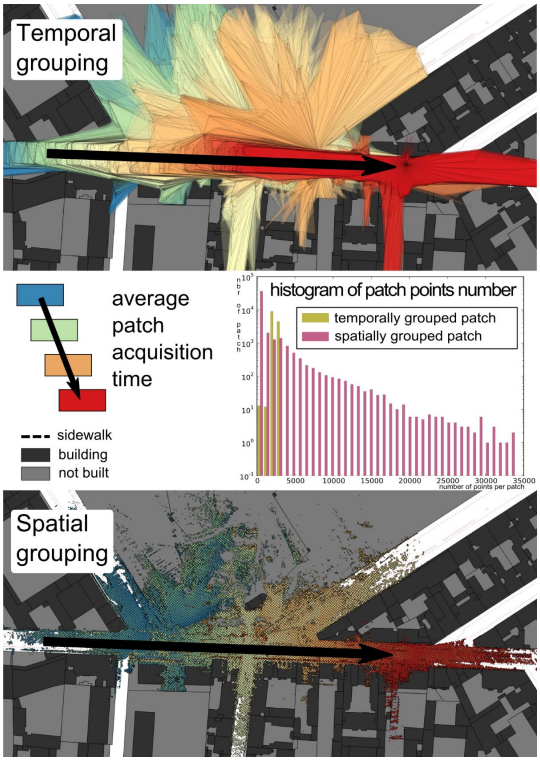

Figure 14: Temporal (top) and spatial (bottom) grouping on velodyn terrestrial Lidar data, with histogram of number of points per patch.

Choosing grouping rule Overall, the chosen grouping rule has no impact on performances as long as the number of rows remains in the same magnitude (few millions). The impact of using different grouping rules is essentially to enable different applications. For instance, the adaptative grouping rule produces patches with a much more constant density, which is useful for many processing methods. We refer the reader to Cura et al. (2016) (working paper) for more details on point cloud density correction.

3.2 Loading

'Server oriented loading' In one night, we aim at loading the data sensed by a Lidar system during one day.

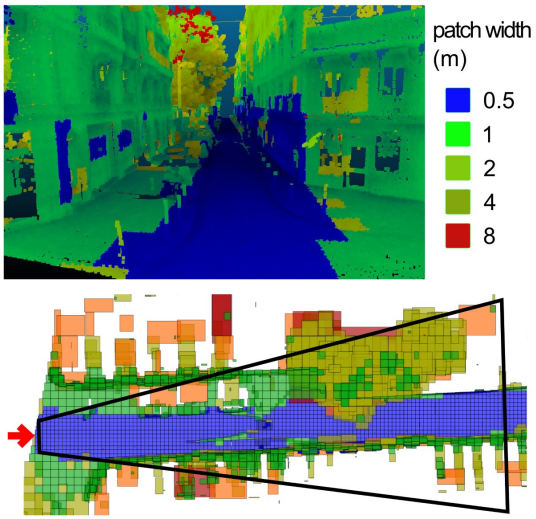

Figure 15: Illustration of variable patch size repartition in an urban point cloud.

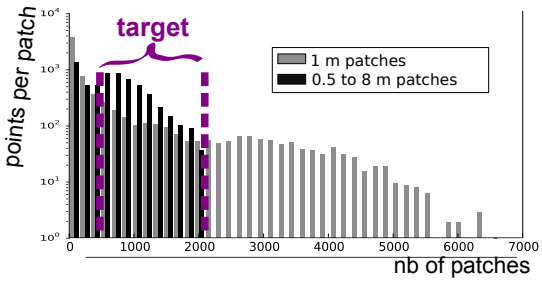

Figure 16: Histogram of points per patch for constant and variable size patch. Using variable size patch strongly reduces the number of very small or very large patches. Total number of patches is roughly conserved.

Indeed, a mobile mapping vehicle may be acquiring data all day long, but would most likely not operate during the night (street views would be useless). Thus the data acquired during the day must be dealt with during the night so as to keep up when vehicle is used everyday. Please note that this requirement is specific to only one of our four data sets.

The points are stored in files, over a gigabyte network. We use a specialised program to convert the points file into ASCII values (CSV). For .ply point files the program is a modified version of the RPly library[11], for the .las files the program is LAStools [12]. The points in ASCII values are streamed to a 'psql' process that is connected to the database. The 'psql' executes a 'COPY' SQL command that reads the ASCII streams and creates and fills a table with the values from the ASCII stream. When the file has been fully streamed, we use a SQL query to create points from attributes and group them into (compressed) patches. These patches are inserted into the final patch table. This pipeline (Figure 6, page 5) is executed several times in parallel, each pipeline working on a different file. The process is pipped so there is no intermediary file written to disk.

Distributed 'Client oriented loading' In this experiment, we use clients to send uncompressed patches to the server . The clients read point cloud files (.ply in our experiment, using the plyfile[13] Python module). Then, each client groups the points into patches using a custom Python module. The patches are sent to the server through Python. The server compresses these patches and adds them to the final point cloud table. This experiment is a proof of concept; therefore, we limit the number of clients to one computer, using seven threads.

Result We load "Vosges" and "Paris" data sets through 'Parallel loading', and "Stereo" through Distributed parallel loading' (Table 4). Morevover, we load a part of the Vosges data set using PDAL (parallelised using a bash script) to enable comparison of our result with van Oosterom et al. (2015). We estimate the loading speed of the Vosges data set with PDAL at 300kpts/s, and the writing speed at about 750kpts/s.

We try to roughly estimate the bottleneck of each method in the following way: we vary the number of cores used. If the timing is linear with the number of cores, the process is CPU-bound, that is CPU is the bottleneck. Else, input/output (I/O) is the most likely bottleneck.

For PDAL, the bottleneck is clearly the CPU, for both of our methods, the input/output (I/O) may also play a role. Indeed, the point files are read over the network, and the point tables are stored on the SSD, but the final patch table is stored on the regular disk, which also limits how many threads can write data on it at the same time (we observe almost linear scaling for all methods up to 7 threads).

Please note that PDAL and our methods do not use the same grouping rules (PDAL uses fixed max size (streaming friendly), while we group points into fixed size cubes (necessitate to read the whole input file before grouping)). Results are in the Table 4.

3.3 Point Clouds and Context

First, we demonstrate the construction of several two-dimensional (2D) vectorial visualisations of point clouds.

[11] http://w3.impa.br/~diego/software/rply
[12] http://lastools.com
[13] www.github.com/dranjan/python-plyfile

Table 4: Loading and writing time for each point cloud data set, using various methods.

data set	Loading time	Parallelism	Loading speed kpts/s	Writting speed Mpts/s	writting limitation
Vosges	11h30	8	125	1.1	write speed
Paris	8h	6	74.5	0.2	read / uncompress
Stereo	7'20	7	160	0.55	read

The PCS can work on all point clouds at the same time, transparently for the user. Point clouds can also be used conjointly with other GIS data (raster and vector). Lastly, we show an example that demonstrate the use of the sensor trajectory meta-data (§3.3 p. 15), and the creation of graph / topology at the patch level.

Coverage visualisation Creating a coverage visualisation is easy (about 30 SQL lines) and fast (about 150s for Paris, one thread) with our point cloud server.

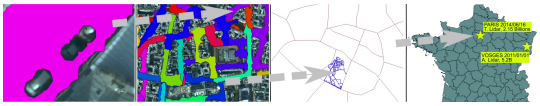

Figure 17: Successive visualisations (various scale) of point cloud coverage, see 3.3 for details.

Indeed, instead of working with billions of points, we can work with millions of patches (generalising the points).

We created several visualisations for the Paris data set, ranging from 5MByte to 100kByte, each adapted to a different scale and purpose. (see Figure 17)

- 1:25 to 1:1500: Precise, occlusions visible ($\sim$1m).
- 1:1.5k to 1:15k: Understand acquisition structure and road morphology ($\sim$8m).
- 1:15k to 1:200k: Use the trajectory. If not available, fabricate a trajectory-ersatz through basic straight skeleton.
- $\geq$1:200k: A simple point with text attributes for details, linked to a relational model.

As a proof of concept, we propose a coverage hexagonal grid (see Figure 18), conceptually identical to a regular grid, with some benefits (see Sahr (2011)). The idea is to visualise both what was sensed and what remains to be sensed in a given area (here whole Paris city), to help plan data-sensing missions. The whole process is fully automatic. We fabricate an hexagonal grid over the extent of Paris (about 30s), and remove the hexagons that are in buildings or too far from a road (about 60s). Then we colour the hexagons depending on whether the point cloud actually covers it or not (about 30s). Such

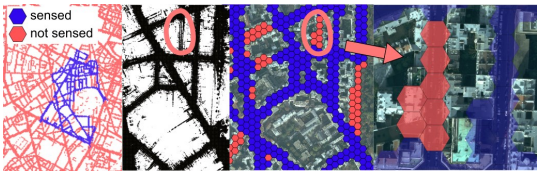

Figure 18: "To-Do" hexagonal map showing the places where the mapping vehicle has not sensed enough points (red hexagon), and where the sensing is sufficient (blue hexagon). Without this map, the zoomed missing part is challenging to notice on raw data.

a visualisation is easy to create (about 30 minutes of design), and could be tailored to more specific needs.

Using several point clouds As a proof of concept of integration of several point clouds, we demonstrate the conjoint use of stereo-vision point cloud and Lidar point cloud. For this experiment, we choose to use PostgreSQL inheritance mechanism. The idea is to create a 'parent' table. We set the Lidar table and stereo-vision table to be a 'child' of the 'parent' table. Then, we can query the 'parent' table as if it was a super-point cloud comprised of all the others. Querying the 'parent' point cloud is as fast as querying one point cloud, and we correctly attain points from both point clouds. We stress, however, that all the child point clouds index are used, which would limit the scaling of this method.

Conjoint use with other GIS data We commonly used vector data with point clouds for various research projects. Here, the scenario is that we have an accurate but slow pedestrian crossing reconstructing process. We want to reconstruct the pedestrian crossing at a given intersection, so we use advanced filtering to provide the appropriate points to the reconstructing process (see Figure 10). To demonstrate the possibilities, we use the following:

- Corrected version of ODParis[14] building layer (350 k rows),
- Lidar sensor trajectory (42 k points regrouped in 900 rows),
- Road network data of BDTopo[15] (32 k rows),
- Aerial image in a PostGIS raster table (110k rows, each 30 $\times$ 30 pixels (10 cm)).

We filter Paris point cloud to obtain patches :

- near street 'Palatine' and 'Servandoni' ($\leq$ 10 m+ road width),
- near Lidar acquisition centre trajectory ($\leq$ 3 m),
- far from buildings ($\geq$ 1 m),

[14] http://opendata.paris.fr/page/home
[15] http://professionnels.ign.fr/bdtopo

Table 5: Result of filtering.

Total points	Found points	Filtering no rast.	Filtering w. rast.	Filtering optim
2.15 Bpts	1.2 Mpts	$\sim$30ms	$\sim$ 5s	$\sim$30ms

- with high density ($\geq$ 1000 points $/\mathrm{m}^3$),
- where the aerial image has a colour compatible with street. markings (240 $\leq$ mean intensity $\leq$ 350)

The point cloud server finds all the patches concerned in about 0.6s (with index and optimally written query) (see Figure 10 and Table 5).

Point cloud as a raster or vector We construct abstract representations of patches that are sufficient for one task, and are much more efficient than using the points, including the following:

- 2D bounding box ('bbox') (default)
- oriented bounding box ('obbox'), light
- multi-polygon obtained by successive dilatation and erosion of points ('closing'), big to store, very accurate

These generalisations are about 0.5% of the compressed patch size. We also tested 3D generalisations, either by extracting primitives (plan, closing on 3D plan, cylinder) or using LOD.
Lafarge et al. (2013) showed that the urban point clouds can be accurately represented by primitives. For instance, a dozen plans accurately explains (distance $\leq$ 1cm) 70 % of this scene (Figure 19). We extensively tested an orthogonal approach, where instead of making a new object to generalise a group of points, we represent it by a subset of well chosen points of this group. The method and its applications (adaptive LOD, density analysis, and classification using density features) are explained in Cura et al. (2016) (working paper).

Using trajectories with point clouds We imported the Paris trajectory data (the successive positions of the Lidar sensor every few ms). In fact, using a constrained data model resulted in discovering errors in the raw trajectory data (some rows were corrupted). Trajectories can be used for filtering point clouds (for instance, Sec 3.3).
We demonstrate the use of trajectories for processing in the following scenario. The goal is to localise all the pedestrian crossings of the Paris data set (few minutes). We (conceptually) walk along the trajectories, and every three metres we retrieve the patches closest to the trajectory. We use a crude marking-detection function on these patches (percent of points in given intensity range). By thresholding this score, we can be conservative or very selective (i.e. favour recall or precision). Recall is the amount of correctly found crossing over the total number of crossing. Precision is the amount of crossing that were correctly found divided by the total number of found crossing.

For instance, with a recall of 0.95, we have a precision of 0.5, and we already filtered the point cloud by a 4.8 factor. This indicates that we reduced the number of points to consider by a factor 4.8 at the price of dropping 5 % of the pedestrian crossing to be found. This could directly be used as a prefiltering step for a more costly pedestrian crossing detector, which would work on 4.8 times less points (at the price of missing at least 5 % of pedestrian crossings).

Orthogonally, with a recall of 0.16 we have a precision of 1, filtering the point cloud with a factor 100. This indicates that we can guarantee that the found pedestrian crossing are effectively pedestrian crossing for 16% of those. This could be useful for fast prototyping. Indeed we may want to test a more subtle pedestrian crossing detector. In this case knowing there is no false positive is important to evaluate the new method.

Point cloud as Graph / Topology We generate a graph embedded in 3D from patches and propose three examples of applications. First we generate the graph by creating a node per patch, the node being at the patch centroid. We de-duplicate results to correctly deal with the fact that the acquisition vehicle made several passes at this place. Patches are regularly spatially placed (for instance forming cubes of 1 m^3). Then we construct an edge for each pair of nodes that are spatially close. Depending on the threshold, we can use 4, 8 or more connectivity (the graph is in 3D). Figure 21 illustrate the different kind of connectivity.

The edge weight is the 3D geometric distance between the nodes (or a more complex measure).

Shortest path The first example takes advantage of geodesic distance to compute the shortest path between two groups of points (see Figure 22). We construct two graphs, one for the regular fixed-size-grouping (1 m^3), and one for a variable-size-grouping (0.25 to 1 m wide patches). We use PGRouting to find the shortest path (about 0.1 sboth cases). Both results are similar, with the shortest path along varying size patches being a better approximation, as expected. This functionality of geodesic distance could typically be used by other advanced processing methods.

Semantic isochrone The second example is intended for semantic point clouds. We suppose that each patch already has a rough classification available (aggregated from available point classification or the result of a direct patch classification(Cura et al. (2016), working paper). We integrate this classification with the geometric distance to create a semantic-geometric distance. We use

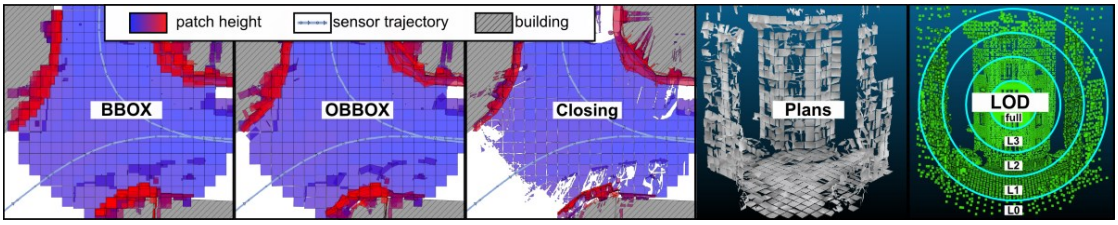

Figure 19: Streets and buildings. Generalisation like Bounding Box, Oriented BBox, Spatial closing. Closing on 3D plan detection. Level Of Details.

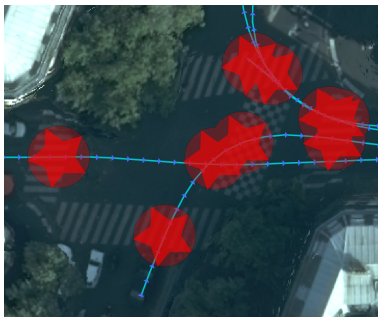

Figure 20: Rough pedestrian crossing detector.

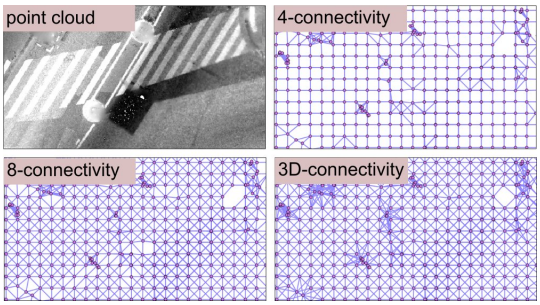

Figure 21: Example of possible connectivity by connecting patch centres that are closer (3D distance) than a threshold (1.4, 1.5, 1.9 metres).

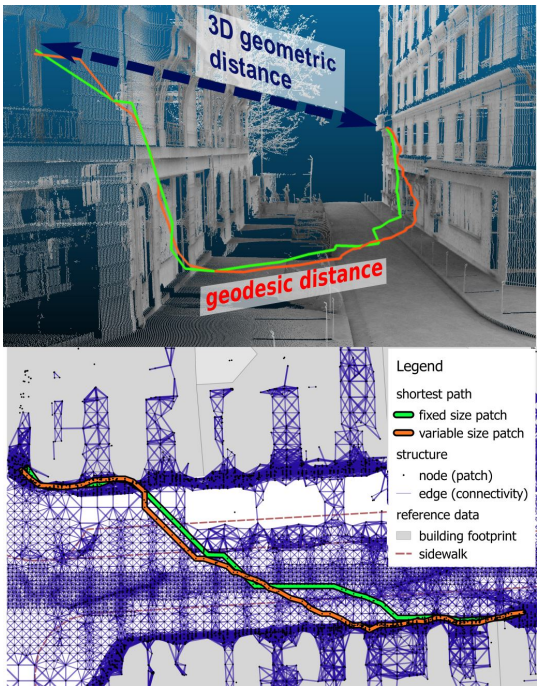

Figure 22: Two views of shortest path on regular (yellow) and varying (orange) size patches.$\sim$ 100ms

the geometric distance divided by a measure of similarity of the patch classes. For instance, two patches are spatial neighbours, one being a ground patch and another a building patch. The semantic-geometric distance will be large.

The graph can then be used for assisted selection. In this scenario, we would like to get all the points pertaining to a façade. A user selects one patch on the façade, then all the patches within a given semantic-geodesic distances are selected (red/yellow) using PGRouting isochrone functionality. This results in selecting only the given façade, as opposed to using a simple geometric distance which would also select point on other façades (geometrically close, but not connected see Figure 23).

More generally high level object reconstruction algorithm would need this kind of feature.

Road network reconstruction The third example use a simplified graph as the starting point to reconstruct a road network (See Figure 24). The idea is to regroup nodes of the graph to simplify it. We generate the simplified graph by sampling patch centroid on a voxelic grid (8 m) coarser than the typical patch size (1 m wide), taking in priority the patch with greatest number of points, and removing patch that are not flat enough to be on the ground. We could directly use semantic information if it was available to keep only ground patches. Edges are created as usual (geometric proximity).

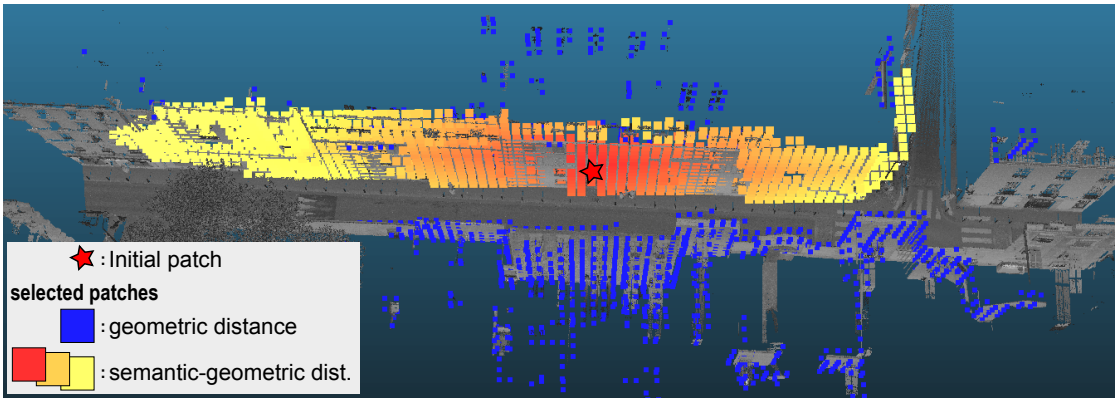

Figure 23: Isochrone from the red to yellow, blue points are selected when using geometric distance, but not selected when using semantic-geodesic distance. About 400 ms.

Please note that such down-sampling of patches is easy to write and fast, and could as easily replaced by more subtle aggregates:

```
SELECT DISTINCT ON (floor(X), floor(Y),
    floor(Z))
  patch_centroid AS node
FROM patch_table /*X,Y,Z are patch
    centroid coordinates*/
WHERE /*trying to characterise ground
    patches to avoid getting facade and
    trees patches*/
  num_points > 1000
  AND patch_height < 0.4 --in meter ...
ORDER BY floor(X), floor(Y), floor(Z)
  , num_points DESC -- large patches have
      priority
```

Using PGRouting and PostGIS, we compute the accumulated graph-distance between all the pairs of nodes close enough (50 m). We can see those $(node1, node2, t\_distance$ as edges of another graph. We create the sparse affinity matrix of this graph using Networkx (Hagberg et al. (2008)). Then, we use spectral clustering with Scikit-Learn (Pedregosa et al. (2011)), each node is then attributed to a cluster. Networkx ans Scikit-Learn are python modules that we use in base via plpython. We can replace the cluster by their centroid, and build a network by computing cluster adjacency relations. We then simplify this network by "healing" 2-connected edges. We compare this result with a more traditional straight-skeleton based approach (use morphological operation to produce a polygon with holes representing the surface of the streets, use straight skeleton to produce centre-lines, clean the straight skeleton result with morphological operations, build a network, topologically clean the network).

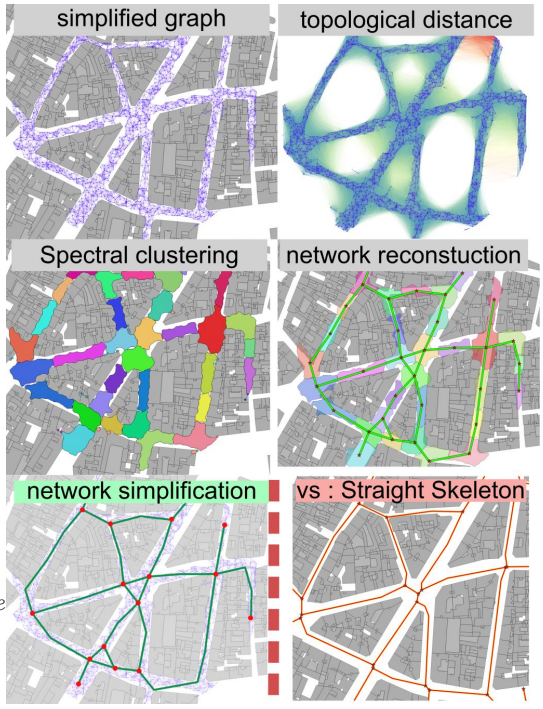

Figure 24: From simplified graph, all topological distance from a vertex to any vertices close enough are computed with PGRouting, the result is clustered using spectral clustering of Pedregosa et al. (2011). We then reconstruct the network with PostGIS Topology, and perform a final simplification step. For comparison, an automated result using a straight skeleton based method.

3.4 Point Cloud Filtering

Filtering overview Overall, filtering of patches is very fast on the Point Cloud Server when an index is used (≤ 0.1s). We tried a great variety of combination of filtering conditions, and always observed this kind of timing provided that indexes were used. The Section 3.3 was designed to give an overview of filtering conditions. Advance filtering condition examples can be found in the method wiki [16]. Finding the patches is almost always much faster than actually retrieving them.

Because of caching and the influence of how the query is written, figures are only indicative. Filtering patches using the indexed functions takes about 0.01s, even when using many conditions at the same time. This includes filtering with spatial (2D and 2D+Z), temporal, any other attributes, density, volume, etc. This also includes using vector generalisation. Filtering with other GIS data (vector) is slower (10s), except when special care is taken to optimise the query (0.01ms). This includes using distance to other vector layers, using other vector layer attributes (e.g. height of building), using time associated with vectors, etc. Lastly, very complex filtering may take from 10s up to several minutes depending on the number of patches concerned.

3.5 Exporting

In this section, we list the results for the various output methods of the PCS.

We estimate the output speed using parallelised (8 threads) PDAL to $750k points/s$ on the Vosges data set, by simply measuring the time taken to output a few hundred million points.

Client oriented: Using PostgreSQL drivers/connector We create a Python method that works on a client computer. It reads uncompressed patches from the server and directly writes them to disk (saving it as Numpy double array). Using seven parallel workers, the result is in Table 4 p. 14.

Server oriented: PLY File As a Service (PLYFAS) We create a service that writes ASCII .ply files at a given network place. The functions (API) have options to perform all kinds of filtering. We exported several files from the Paris data set, with various filtering options and LOD (from Cura et al. (2016), working paper). The global output time observed is around 15 k points/s per worker with a scaling of up to seven workers.

[16] http://bit.ly/21gI81w

Server oriented: Lens for traditional GIS Points are in fact a PostgreSQL materialised view that store points that are defined by the lens spatial extent and attributes. We also add a trigger on the lens table so to refresh the point view upon changes on lens. Optionally, a QGIS plugin[17] can also be used to improve interactivity (instantly autosaving changes concerning PostGIS layers). If the lens is small enough, this method is interactive ($\sim$ 2s). See Figure 12 page 9.

Server oriented: Streaming to browser We performed a test of point cloud streaming to a WebGL application, using a Node.js server as the 'man in the middle'.

- The browser is set to a geographical position, and then requests the points around this position to the Node.js server.
- The Node.js server connects to the PCS to request the points.
- The PCS uses indexes to find patches and extract points that are then streamed to Node.js server through cursor use.
- The Node.js server compresses the point stream and sends it to the web browser.
- The web browser parses the stream, puts the points into buffers, sends the points to the graphics card and displays them through shaders (WebGL).

We observed a reduced throughput ($\sim$ 20 kpts/s, monothread) because data is inefficiently transmitted as text, and is serialised/deserialised multiple times.

3.6 Processing Point Cloud with the Server

The PCS can be extended by in-base and out-of-base processing. In-base processing are methods that are executed by the database from within, whereas out-of-base processing are regular processes executed outside of the database (possibly on other computers) that get data from database, process, and then write results in database or elsewhere.

We demonstrate how easy it is to create new in-base processing methods. As such, the methods are only cited to illustrate these capacities. Some details may be found in Cura (2014).

In-base processing Fast prototyping is vital for wider point cloud use. We demonstrate the potential of using high level languages within the database to write simple processing methods. The experiment is not to create state-of-the-art processing methods, but to measure what a Python/R beginner can do in two weeks (designing methods and implementing), using well established tools

[17] http://remi-c.github.io/interactive_map_tracking

like Scikit-learn (Pedregosa et al. (2011)) or the Point-Cloud Library (Rusu and Cousins (2011)).

((P): directly working on patches, can be used out of the box on all point clouds; (R): working on rasterised point clouds, need to use a point cloud to raster conversion method first (in base or out of base))

- (P) clustering points using Minimum Spanning Tree
- (P) clustering points with DBSCAN (Ester et al., 1996)
- (P) extracting primitives (plans and cylinder)
- (P) extracting a verticality index (using Independent Component Analysis)
- (R) detecting façade footprint
- (R) detecting cornerstones
- (R) detecting road markings

We also used the server for complex out-of-base processing (classifications), although, for the sake of brevity, this is detailed in a standalone article (Cura et al. (2016)).

4. DISCUSSION

4.1 Storing groups of points in a RDBMS

In this article, we introduce the storing of groups of points in Section 2.1 on page 3, and we study different grouping rules in Section 3.1 on page 11.
Storing groups of points in the PCS offers strong advantages in term of compression, indexing and generalisation. Yet, it all depends on the hypothesis that points are grouped into groups that are meaningful for the intended applications. Both spatial and temporal groupings produce good results. Spatial grouping with fixed size patches can be a problem when the density varies strongly (terrestrial Lidar), as patches may contain very few or too many points. We experimented with varying patch sizes and demonstrated that the resulting patches have a much more regular density.

During our usage of the PCS, we noticed a practical limitation concerning the point cloud types which are strongly constrained, thus adding or removing attributes is not immediate. As a perspective, using an inheritance scheme between point types would solve this problem.

We demonstrated that storing groups of points is well adapted to store billions of points per table. Yet, the PCS would have trouble going over a few thousand tables, theoretically limiting the total number of points to the 10 trillion-points range. To go beyond that, we would need to use supplementary PostgreSQL sharding and clustering capabilities. Those capabilities exist but have not been used yet for point clouds, to the best of our knowledge.

Storing groups also enables a generalisation approach, which may have the potential to accelerate and facilitate many point cloud usages. In this work, we only considered a few generalisations, and used them in limited ways. Much more advanced generalisations and usages would be possible (for instance, using Gaussian Mixture).

4.2 Loading

We present several methods to load points into the PCS (§ 2.2 on page 5), and test them on several datasets (§ 3.2 on page 12). We successfully demonstrate a sufficient speed to fulfil our practical requirement of loading one day of sensing data in less than one night. Examining (Martinez-Rubi et al. (2014), Table 2) shows that data loading could be much faster. In the 'Server oriented loading' scenario, points are converted to ascii and streamed, which is a waste of resourcess. The PCS could directly read .ply or .las files. In this scenario, the database performs the grouping via generic SQL queries. It might be faster to create a tuned C function to do this.

In the Distributed loading ('Client oriented loading'), the client performs the grouping, but the database still performs the compression. The client could also compress, saving bandwidth and computing time for the server. Moreover, our prototype is written in Python and could be written for efficiency in lower level languages. As such, the relatively recent initiative, PDAL[18] has gained maturity, and would be the ideal candidate to solve these two limitations.

In a more distant perspective, we could skip reading post processed .las or .ply files, and directly read the raw sensor data, which might be nevertheless difficult due to current lack of driver and standard accessors.

4.3 Point Cloud and Context

Point clouds are not only sets of points and also contain very important meta-data (§ 2.3 on page 5). We used these meta-data in several ways (§ 3.3 on page 13).

We demonstrated that such meta-data can be useful to create multi-scale visualisations of point cloud coverage, as well as help to analyse sensed area ("Todo" map).

Each point cloud meta-data scheme must defined and enforced by the user, making it hard to share. A standard minimal data model would be necessary to facilitate exchanges, similar in spirit to the INSPIRE[19] European directive.

A shared meta-data scheme allows to use several point cloud together. We tested the PostgreSQL inheritance

[18] http://www.pdal.io/
[19] http://inspire.ec.europa.eu/

mechanism so all point clouds are parts of one meta-point cloud. Current limitations of this mechanism would prevent it to be used on more than a dozen point clouds, but perspectives exist to solve this problem (using rule system or enforcing a table-wide pre-filter based on table coverage).

In the PCS, the point clouds also have representations compatible with other GIS data, such as vectors and rasters. Conjointly using vectors, rasters and point clouds offers a new world of possibilities. We face data fusion issues, like difference in precision, generalisation, fuzziness, etc. Moreover, vector, raster and point cloud data may be acquired at different dates.

In-base conversion from point cloud to raster are currently very slow and tailored, being based on SQL queries. A python-based prototype[20] method may solve these limitations.

Going one step further, we demonstrate that point clouds could be generalised as graphs, which opens new possibilities, such as graph based distance, semantic selection and road network reconstruction. However, the current approaches build the graph using plain PostGIS SQL queries that can not scale well beyond the million of patches. The bottleneck is simply the conversion from patches centroid to a graph based on adjacency, and could be done directly using powerful specialised library, such as Boost graph library[21]. Moreover, the reconstructed road network by either methods has large improvement margins (topologically and geometrically). It would be possible to mix PostGIS Topology (2D partition of the space) for graph queries.

4.4 Filtering point clouds

The PCS has very advanced capabilities to access a subset of the point clouds (Filtering, § 2.4 on page 7). Filtering relies on indexing and we demonstrate it is very fast provided the filtering conditions are indexed and the points are grouped in meaningful groups (3.4 on page 18). Level Of Detail and meta-data (trajectory) naturally integrates well into the filtering conditions.

The filtering conditions are especially useful when using other GIS data. It would be possible to go much further towards complex filtering, by performing algebra between several rasters, using attributes of vectors to filter patches, etc.
Our entire strategy relies on filtering patches first, then filtering points. In cases where the patch filtering condition does not filter much, the system becomes useless.

[20] https://github.com/Remi-C/PPPP_utilities/blob/master/pointcloud/patch_to_raster.py

[21] www.boost.org/doc/libs/1_58_0/libs/graph/doc/

4.5 Exporting

The PCS being based on a popular RDBMS, many ways exist to access the data stored in it (§ 2.5 on page 8). We demonstrated an array of export methods (§ 3.5 on page 18), from classic server based export to multi-client based, up to the notion of point clouds as service.

The point cloud server can output data in many ways and thus be easily integrated into any work-flow. We, however, feel that the current speed (100 k points /s, around 2 MByte /s) is too low. It could be easily accelerated using binary outputs and by decompressing patches directly on clients.
Similarly, the lens feature is limited, adapted LOD could be chosen automatically, but this involves modifying the GIS used for visualisation. Perhaps the true evolution of the point cloud server would be to stop delivering points, and instead deliver a service that could be queried through standard mechanisms. For instance, the transactional Web Feature Service (WFS-t) format could be used to send points out of the box, simply using a geo-server between the client and the point cloud server. This could be a revolution in point cloud availability, similar to what happened to geo-raster data (e.g., google map WFS).

4.6 Processing Point Cloud with the Server

One of the advantages of using the PCS is the opportunity to not only store point clouds, but also the methods to process it (§ 2.6 on page 10). We demonstrated the PCS capabilities for fast prototyping of in-base processing methods (§ 3.6 on page 18).

The methods we designed are proof of concept, and far from real state of the art. In-base processing offers many opportunities because it is close to the data and can be written with many programming languages. Yet, it is also intricately limited to one thread and the amount of memory allowed for PostgreSQL. The execution is also within one transaction. It may also be hard to control the execution-flow, during the execution. However, the Python access both from within and outside the database shows the possibility to write more ambitious processing methods with several parts executing in parallel as well as communicating, dealing properly with errors, etc. We sucessfully integrated the PCS into a more complex classification framework in Cura et al. (2016).

4.7 Future work

Patches and their generalisations are perfect candidates to perform fast and efficient registration (cloud-to-cloud, cloud-to-raster, etc.) (See Figure 25). Indeed, the classical solution for registering two point clouds relies on many point to point distance computations (in Iterative

Closest Point for instance, Besl and McKay (1992)). With large point clouds, the problem grows intractable, and, thus, a common solution is to subsample the point clouds to reduce the number of points. This introduces errors, and is still less than perfect. Using extracted primitives would be better, as visually explained in Figure 25. Indeed, higher level primitives contains much more information (more abstract), and are much less numerous (more synthetic), both being great for faster registration.

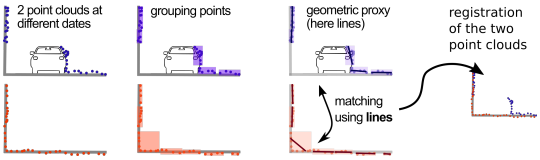

Figure 25: A schematic example of the benefits of using generalisation of points for fast registering. The critical part of matching could be done on geometric proxies instead of points, reducing the number of entities to be matched by a factor of at least 10^3.

Having all the meta-data, the trajectory (or camera position matrices), and the raw data, it would be possible to change the trajectory (matrices) and regenerate the point cloud with updated coordinates, all of this from within the database. Indeed, the trajectory or camera position are usually known up to a positioning error, being the result of a process (Structure from motion, GPS positioning, etc.). Yet, those positions could be improved by exploiting other data, manual correction, etc. In this case, the improved trajectory/ positions could be used to re-generate the point cloud, leading to more accurate point clouds and limiting data duplication. Processing of point clouds would extract landmarks, which could be matched with a landmark database.

5. CONCLUSION

In this article, we presented a comprehensive point cloud server system based on groups of points (patches). Using these patches as generalisations, we propose solutions for basic point cloud user needs (loading, storing, filtering, exporting and processing). The system is fully open source and thus easily extensible and customisable using many programming languages (C, C++, Python, R, etc.). Our system opens new possibilities because of intricate synergy with other geo-spatial data. Lastly, we proved through real-life uses that this system works with various point cloud types (Lidar, stereo-vision), not only for storing point clouds, but also for processing. As a perspective, we could explore in-base re-registration from trajectory and raw data, in-base cloud-to-cloud registration, in-base classification, and point streaming, as well as scaling to thousands of billions of points.

6. ACKNOWLEDGEMENTS

The authors would like to thank the reviewers for their in-depth suggestions and corrections. This work uses many open source projects, we thank their communities for great features and softwares, especially Paul Ramsey, the author of pgpointcloud. We thank our colleagues for their ideas and help both theoretical and practical.

This work was partially supported by an ANRT grant (20130042).

7. BIBLIOGRAPHY

References

Aubrecht, C., Steinnocher, K., Hollaus, M. and Wagner, W., 2009. Integrating earth observation and GIScience for high resolution spatial and functional modeling of urban land use. Computers, Environment and Urban Systems 33(1), pp. 15–25. 6

Azim, A. and Aycard, O., 2012. Detection, classification and tracking of moving objects in a 3D environment. In: 2012 IEEE Intelligent Vehicles Symposium (IV), pp. 802–807. 12

Besl, P. J. and McKay, N. D., 1992. Method for registration of 3-D shapes. pp. 586–606. 21

Bier, E. A., Stone, M. C., Pier, K., Buxton, W. and DeRose, T. D., 1993. Toolglass and magic lenses: the see-through interface. In: Proceedings of the 20th annual conference on Computer graphics and interactive techniques, ACM, pp. 73–80. 9

Chum, O. and Matas, J., 2002. Randomized RANSAC with td, d test. In: Proc. British Machine Vision Conference, Vol. 2, pp. 448–457. 7

Cura, R., 2014. A postgres server for point clouds storage and processing. https://github.com/Remi-C/Postgres_Day_2014_10_RemiC/tree/master/presentation. 18

Cura, R., Perret, J. and Paparoditis, N., 2016. Implicit LOD for processing, visualisation and classification in Point Cloud Servers. CoRR. 10, 12, 15, 18, 19, 20

Edelsbrunner, H., Kirkpatrick, D. and Seidel, R., 1983. On the shape of a set of points in the plane. IEEE Trans. Inf. Theory 29(4), pp. 551–559. 6

Ester, M., Kriegel, H.-p., S, J. and Xu, X., 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In: proceedings of 2nd International Conference on Knowledge Discovery and Data Mining, AAAI Press, pp. 226–231. 19

Hagberg, A. A., Schult, D. A. and Swart, P. J., 2008. Exploring network structure, dynamics, and function using NetworkX. In: Proceedings of the 7th Python in Science Conference (SciPy2008), Pasadena, CA USA, pp. 11–15. 17

Hofle, B., 2007. Detection and utilization of the information potential of airborne laser scanning point cloud and intensity data by developing a management and analysis system. PhD thesis, Institute of Photogrammetry and Remote Sensing, Vienna University of Technology. 2, 6

Hug, C., Krzystek, P. and Fuchs, W., 2004. Advanced lidar data processing with LasTools. In: The International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences, pp. 12–23. 2

IQmulus, 2014. IQmulus & TerraMobilita contest. http://data.ign.fr/benchmarks/UrbanAnalysis/. 10

Isenburg, M., 2013. Laszip. Photogrammetric Engineering & Remote Sensing 79(2), pp. 209–217. 11

Kiruthika, J. and Khaddaj, S., 2014. Performance issues and query optimization in big multidimensional data. In: 2014 13th International Symposium on Distributed Computing and Applications to Business, Engineering and Science (DCABES), pp. 24–28. 7

Lafarge, F., Keriven, R., Brédif, M. and Hoang-Hiep Vu, 2013. A hybrid multiview stereo algorithm for modeling urban scenes. IEEE Trans. Pattern Anal. Mach. Intell. 35(1), pp. 5–17. 4, 15

Lewis, P., Mc Elhinney, C. P. and McCarthy, T., 2012. LiDAR data management pipeline; from spatial database population to web-application visualization. In: Proceedings of the 3rd International Conference on Computing for Geospatial Research and Applications, p. 16. 2, 6

Lobo, M.-J., Pietriga, E. and Appert, C., 2015. An Evaluation of Interactive Map Comparison Techniques. In: CHI '15 Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, ACM Press, pp. 3573–3582. 9

Mackaness, W., Burghardt, D. and Duchêne, C., 2014. Map Generalisation: Fundamental to the Modelling and Understanding of Geographic Space. In: D. Burghardt, C. Duchêne and W. Mackaness (eds), Abstracting Geographic Information in a Data Rich World, Lecture Notes in Geoinformation and Cartography, Springer International Publishing, pp. 1–15. 4

Martinez-Rubi, O., Kersten, M. L., Goncalves, R. and Ivanova, M., 2014. A column-store meets the point clouds. FOSS4G-Eur. Acad. Track. 3, 19

Martinez-Rubi, O., van Oosterom, P., Gonçalves, R., Tijssen, T., Ivanova, M., Kersten, M. L. and Alvanaki, F., 2015. Benchmarking and improving point cloud data management in MonetDB. SIGSPATIAL Spec. 6(2), pp. 11–18. 3

Meng, L. and Forberg, A., 2007. 3D building generalisation. Challenges in the Portrayal of Geographic Information. Elsevier Science, Amsterdam pp. 211–232. 4

Otepka, J., Ghuffar, S., Waldhauser, C., Hochreiter, R. and Pfeifer, N., 2013. Georeferenced point clouds: A survey of features and point cloud management. The International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences 2(4), pp. 1038–1065. 2

Otepka, J., Mandlburger, G. and Karel, W., 2012. The OPALS data manager—efficient data management for processing large airborne laser scanning projects. The International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences 25, pp. 153–159. 2

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M. and Duchesnay, E., 2011. Scikit-learn: Machine learning in Python. Journal of Machine Learning Research 12, pp. 2825–2830. 17, 19

pgPointCloud, R., 2014. pgPointCloud. https://github.com/pgpointcloud/pointcloud. 3

pgRouting, d. t., 2015. pgRouting. http://pgrouting.org/. 7

Pindat, C., Pietriga, E., Chapuis, O. and Puech, C., 2012. JellyLens: content-aware adaptive lenses. In: Proceedings of the 25th annual ACM symposium on User interface software and technology, ACM, pp. 261–270. 9

PostGIS, d. t., 2014. PostGIS. www.postgis.org/. 3, 6

PostgreSQL, d. t., 2014. PostgreSQL. www.postgresql.org/. 3

Preiner, R., Mattausch, O., Arikan, M., Pajarola, R. and Wimmer, M., 2014. Continuous Projection for Fast L-1 Reconstruction. ACM Transactions on Graphics (TOG) 33(4), pp. 47. 4

Quackenbush, L. J., Im, I. and Zuo, Y., 2013. Road extraction: a review of lidar-focused studies. Remote Sensing of Natural Resources pp. 155–169. 7

Richter, R. and Döllner, J., 2014. Concepts and techniques for integration, analysis and visualization of massive 3d point clouds. Comput. Environ. Urban Syst. 45, pp. 114–124. 2

Rieg, L., Wichmann, V., Rutzinger, M., Sailer, R., Geist, T. and Stötter, J., 2014. Data infrastructure for multi-temporal airborne LiDAR point cloud analysis – examples from physical geography in high mountain environments. Computers, Environment and Urban Systems. 2

Rusu, R. B. and Cousins, S., 2011. 3D is here: Point Cloud Library (PCL). In: IEEE International Conference on Robotics and Automation (ICRA), IEEE, Shanghai, China, pp. 1–4. 19

Sahr, K., 2011. Hexagonal discrete global grid systems for geospatial computing. Arch. Photogramm. Cartogr. Remote Sens. 22, pp. 363–376. 14

Ummenhofer, B. and Brox, T., 2015. Global, dense multiscale reconstruction for a billion points. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 1341–1349. 4

van Oosterom, P., Martinez-Rubi, O., Ivanova, M., Horhammer, M., Geringer, D., Ravada, S., Tijssen, T., Kodde, M. and Gonçalves, R., 2015. Massive point cloud data management: Design, implementation and execution of a point cloud benchmark. Comput. Graph. 49(Special Section on Processing Large Geospatial Data), pp. 92–125. 3, 10, 13

Wang, F., Aji, A. and Vo, H., 2014. High performance spatial queries for spatial big data: from medical imaging to GIS. SIGSPATIAL Spec. 6(3), pp. 11–18. 3

Wu, J., 2011. Improving the writing of research papers: IMRAD and beyond. In: Landsc. Ecol., Vol. 26, pp. 1345 – 1349. 3

Youn, C., Nandigam, V., Phan, M., Tarboton, D., Wilkins-Diehr, N., Baru, C., Crosby, C., Padmanabhan, A. and Wang, S., 2014. Leveraging XSEDE HPC Resources to Address Computational Challenges with High-resolution Topography Data. In: Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment, XSEDE '14, ACM, New York, NY, USA, pp. 59:1–59:2. 6