3D geoinformation

Department of Urbanism
Faculty of Architecture and the Built Environment
Delft University of Technology

Storing and analysing massive TINs in a DBMS with a star-based data structure

Technical report 2015.01

Hugo Ledoux
h.ledoux@tudelft.nl

November 30, 2015

▸ To cite this document:

Ledoux, H. (2015). *Storing and analysing massive TINs in a DBMS with a star-based data structure.* Technical Report 2015.01, 3D geoinformation, Delft University of Technology, Delft, the Netherlands.

TO DOWNLOAD: https://3d.bk.tudelft.nl/pdfs/15_tech_pgtin.pdf

Storing and analysing massive TINs in a DBMS with a star-based data structure

Hugo Ledoux
h.ledoux@tudelft.nl

November 30, 2015

While a database management system (DBMS) is usually the tool of choice to handle massive amount of data, its use for point cloud datasets is somewhat problematic. Indeed, storing millions—or even billions—of unconnected points is feasible but to be able to analyse and manipulate them more is needed: we must reconstruct the surface represented by the sample points, and we must also be able to efficiently access this surface and derive values from it. I investigate in this paper the storage and the manipulation in a DBMS of one such surface: the *triangulated irregular network* (TIN). I present a data structure that permits us to store at the same time—in only one table—both the samples of a point cloud and a TIN (a Delaunay triangulation in this case); it is based on the idea of storing the *stars* of vertices. I describe the basic properties of the structure, explain how it can be implemented in a DBMS, and demonstrate with examples how it compares to alternatives. A star-based data structure has several advantages over current alternatives for storing TINs: (i) it is space efficient; (ii) topological relationships between triangles are stored explicitly, which allows the implementation of efficient spatial analysis functions (such as interpolation, slope, profile, etc.); (iii) a spatial index (such as an R-tree) is not needed as the triangulation implicitly stored can act as an index; (iv) the structure is dynamic and can be efficiently updated. I have implemented the structure in PostgreSQL and I report on experiments that were made with some real-world datasets that can be considered massive (i.e. around 280-million points, or more than half a billion triangles).

1 Introduction

New technologies such as airborne altimetric LiDAR (**Li**ght **D**etection and **R**anging) or multi-beam echosounders permit us to collect millions—and even billions—of elevation/depth points for a given area, and that very quickly and with great accuracy. One example of the use of these technologies is the AHN2 dataset[1] in the Netherlands. By the end of 2012 the whole country will

[1] Actueel Hoogtebestand Nederland. www.ahn.nl

be covered with at least 4 points/m^2, and up to 30 or 40 points/m^2 in certain areas, which means a dataset containing about 640 billions points. What is ironic is that while datasets like this one are being collected in several countries (because of their many possible applications such as flood modelling, monitoring of dikes, forest mapping, the generation of 3D city models, etc.), in practice they are seldom used since the tools that practitioners have, and are used to, cannot handle such massive datasets. As explained in Section 2, the traditional GISs and terrain modelling tools are limited by the main memory of computers: if a dataset is bigger then operations will be very slow, and will most likely not finish. Practitioners have to resort to *simplify* datasets, i.e. reduce the number of points. However, selecting important points require the use of an auxiliary data structure (Garland and Heckbert, 1995), which is a challenge to create in the first place for massive datasets (Agarwal et al., 2005; Isenburg et al., 2006a).

LiDAR or multi-beam echosounder datasets, also called *point clouds*, are formed by scattered points in 3D space that are—in most cases—the samples of a surface that can be projected on the horizontal plane (a so-called "2.5D surface"). To be able to *analyse* a point cloud, more then simply the points are needed: we must be able to reconstruct the surface implied by the points, and we must also be able to manipulate it (e.g. adding/removing new samples). The surface gives us the spatial relationships between otherwise unconnected points in 3D space, which is required if processing and extraction of values from the surface is wanted. Examples of point cloud processes useful for many applications are: derivation of slope/aspect, conversion to a grid format, control of double points, calculations of area/volumes, viewshed analysis, creation of simplified DTM, extraction of basins, etc. While several ways to reconstruct the surface exist (the simplest way would be with a grid), a triangulated irregular network (TIN) is arguably an attractive option since the original points are kept and it adapts to the spatial distribution of the points (Kumler, 1994).

I discuss in this paper the storage and the manipulation of massive TINs. As discussed in Section 2, while there are several solutions to deal with massive TINs, I investigate the use of database management systems (DBMSs) since they are arguably the best tool to store and manage very large datasets (of any kind), are already part of the toolbox of most GIS practitioners, and offer several advantages over file-based systems, e.g. security, versioning, scalability, etc. (Ramakrishnan and Gehrke, 2001). If a TIN is entirely stored in a DBMS (the points, the triangles, and their topological relationships), then one can rely entirely on the DBMS for memory management, and since the whole TIN is available (and does not need to be recomputed), one can readily query and manipulate it.

I present in Section 3 a new data structure for representing compactly a TIN. This structure, which uses recent advances in the compression of graphs in main memory (Blandford et al., 2005), departs from the current alternatives (described in Section 2) since the atom of the structure is not the triangle or the edge, but the *star* of a vertex (triangles are implicitly represented). The main benefits of the star-based data structure in a DBMS are many: (i) it is as space efficient as storing triangles with triplets of point labels, and yet it is fully topological; (ii) the use of a spatial index is not needed, since the triangulation itself acts as the indexing structure; (iii) it is dynamic (so new points can be added and/or deleted from a TIN without having to reconstruct it from scratch); (iv) triangulations with constraints are possible.

Section 4 demonstrates how this star-based structure has been implemented in a DBMS (as an extension to PostgreSQL in this case) and elaborates on its advantages over existing solutions. It also presents different spatial analysis functions that have been implemented in the PostgreSQL extension. Since storing massive TINs first implies that that these must be constructed, I describe in Section 5 tools to construct massive TINs and discuss an important concept for an efficient DBMS implementation of a star-based structure: *spatial coherence*. Finally, Section 6 reports on experiments that were carried out with different real-world massive datasets (containing more than half a billion triangles). These datasets were stored in a DBMS with both a star-based data structure and with explicit triangles (PostGIS Simple Features), and I compare them in terms of space needed, incremental updating and query time for different spatial analysis functions.

2 Related Work

While most GIS packages offer the possibility to manipulate and analyse terrains and TINs, the size of a computer's main memory decides how many points can be processed. Basically, once the main memory is full, transfer of data between the disk and the memory starts, and if this happens too much, the computations might simply stop (called *trashing*) (Isenburg et al., 2006b).

2.1 Reducing the size of a dataset

The most common solution to dealing with massive datasets is *thinning*, i.e. keeping only every *n*-th point in the original dataset. While it permits the user to process the data, it somehow destroys the idea of working with high-resolution altimetry data in the first place, and "important points" (representing for instance ridges or peaks) can potentially be discarded.

A second solution is *tiling* a big dataset into smaller parts that fit into memory, and working on one given part at a time. While this solution is viable in some cases (e.g. for unconnected points as implemented in Oracle Spatial 11g, see below), for TINs this is problematic since, unless many points are explicitly added on the border of the tiles, triangles will overlap several tiles. The processing of such a dataset can be problematic, especially if a global access to the TIN is needed (e.g. calculation of areas/volumes of features, simplification, and flow modelling).

A third solution consists of creating a multi-resolution representation of a terrain (De Floriani and Magillo, 2002). For instance, since version 9.2 ArcGIS[2] has the *Terrain* type, which implements this method (Peng et al., 2004). In a nutshell, all the points are stored in a DBMS but the triangles are not explicitly stored. The user must select so-called "vertical indexes" so that a hierarchy of points is created. For each level in the hierarchy, ArcGIS selects representative points to construct on-the-fly a TIN, when needed. Because each TIN is small in size when compared to the original dataset, one can use the usual ArcGIS tools for processing the TINs. The main problem with this approach is how representative points can be selected. The method used by ArcGIS is unclear, and it appears that they are randomly selected. Selecting important points involves constructing a

[2] `http://www.esri.com/software/arcgis/index.html`

representation of the surface (Garland and Heckbert, 1995), which is a problem with datasets larger than memory.

2.2 Storing TINs in a DBMS

While TINs are widely used in GIS, there are, to the best of my knowledge, very few options developed for efficiently storing one in a DBMS. I review here the few existing solutions, and discuss potential solutions that could be implemented, highlighting the pros and cons of each.

Simple Features. The simplest way is to store each triangle with a *Simple Feature*'s "Polygon" (OGC, 2006), and use a spatial index (e.g. a R-tree or one of its variants (van Oosterom, 1999)). This has however several drawbacks. First the adjacency relationships between the triangles are not explicitly stored and have to be computed on-the-fly (an expensive operation since intersections tests are involved). Second, indexing the data at the triangle level is problematic since the size of the spatial index can become huge; I give in Section 6 concrete numbers for the size on disk of spatial indexes and the use of that structure in PostgreSQL. Indeed, a spatial index usually requires the bounding box of every element; for arbitrary polygons adding two points per objects is sensible, but for triangles it increases instantly the storage by 67%. Also, spatial indexes are more complex trees than a standard index such a B-tree (van Oosterom, 1999). It should also be said that the number of triangles in a TIN is roughly two times that of the generating points: consider a dataset with n points and m points that lie on the boundary of the convex hull of the dataset, then there are $2n - 2 - m$ triangles (de Berg et al., 2000). (For real-world datasets m is often small compared to n, as Section 6 shows.)

Triangles and their neighbours. Although I am not aware of any DBMS implementation, it would also be possible to implement the well-known triangle-based structure, used by several triangulators (Boissonnat et al., 2002; Shewchuk, 1997). The atom of this structure is also the triangle, but adjacency relationships between triangles are explicitly stored. Storing it in a DBMS would require two tables: one for the points (where an ID is given to each), and one for the triangles (which are formed by triplets of points' IDs). The triangle table would also need to have an ID per triangle, and store the ID of the 3 adjacent triangles, ordered with respect to the 3 points. Spatially indexing such a structure would require first reconstructing the geometry of each triangle with a join between the two tables, and then indexing these geometries. In practice this join is however not efficient for massive datasets and implementations of topological data structure avoid it by storing explicitly the geometries in the triangle table, see for instance van Oosterom et al. (2002), Oracle (2012) and the new implementation in PostGIS[3], which is the PostgreSQL extension that adds support for geographical objects. In brief, such a structure would take up as much space as the previous (since triangles would also be explicitly stored and indexed), but some queries (where adjacency information is needed) could be sped up (although that would require indexing also the triangles' IDs, which would also add storage space).

[3] `http://trac.osgeo.org/postgis/wiki/UsersWikiPostgisTopology`

Edge-based data structures. Similarly, edge-based topological data structures developed for storing triangulations and planar subdivisions in main memory could be used; examples are the DCEL (Muller and Preparata, 1978) or the *half-edge* (Mäntylä, 1988). These permit us to store explicitly not only the adjacency relationships, but also the incidence relationships between vertices, edges and triangles. The same dilemma between storing explicitly the geometries of the edges and reconstructing them with a join is present. However, we should keep in mind that for a dataset with n points, the number of edges will be $3n$, 1.5 times that of the triangles ($2n$ triangles have each 3 edges, shared by 2 triangles). Also, the bounding box of a straight-line edge takes the same space as the edge itself. This will lead to a bigger spatial index than that for the triangles.

Oracle Spatial's `SDO_TIN` To my knowledge, the only DBMS types specifically designed for TINs is the Oracle Spatial 11g `SDO_TIN` (Oracle, 2012). Although at this moment it is still under development and is not fully functional, it is worth reviewing to gain insights about a TIN-specific type. The `SDO_TIN` type is essentially the `SDO_PC` type (for the storage of points clouds see Ravada et al. (2010) and Finnegan and Smith (2010)) with extra information about the triangles: each triangle is defined as a reference to 3 vertices, and no adjacency information is stored. The `SDO_PC` type subdivides a point dataset into *blocks* with a given maximum number of points (e.g. 5000), and these blocks are then spatially indexed (thus the indexing is not performed at the point or triangle level). Points, and triangles, inside a block are not further spatially indexed, since it is assumed that a relatively small number of elements will be in a block. For creating the blocks, a variant of the well-known kD-tree is used: nodes of the tree can either partition the space or act as a "bucket" containing a maximum number of points (see Bentley (1990) for more information); the spatial index used is an R-tree. While it is not possible to report on the performances of the type at this point, the approach has the problems of tiling: several triangles overlap more than one block and thus these are represented more than once (they have to be deleted on-the-fly when queries are performed). It should also be said that the `SDO_PC` and `SDO_TIN` types permit users to store their point clouds and TINs and perform basic queries (range queries with a rectangle or a circle are possible), but no spatial analysis functions are available (at this moment). Also, perhaps the main drawback for practitioners is that the data structure used is not dynamic, i.e. if one wants to add a new point, then the whole dataset must be re-processed.

2.3 Constructing and manipulating massive TINs

External memory algorithms. To deal with massive datasets, one can also design external memory algorithms (Vitter, 2001). These basically use disks to store temporarily files that do not fit in memory, and instead of using the mechanism of the operating system, design explicit rules for the swapping between the disk and the memory. Agarwal et al. (2005) construct massive TINs this way, and Arge et al. (2006) and Agarwal et al. (2008) have implemented spatial analysis functions on TINs based on that paradigm. The main drawbacks of this approach are that the design of such algorithms is rather complex, and for different problems different solutions have to be designed. We can consider the approach as creating an optimised DBMS for a specific problem.

Streaming of geometries. An alternative approach to dealing with massive datasets is *spatial streaming* (Isenburg and Lindstrom, 2005; Isenburg et al., 2006a,b), which mixes ideas from external memory algorithms with different ways to keep the memory footprint very low. The basic idea of this paradigm is that of a *streaming mesh*: a format for representing triangulations (or meshes) as a set of interleaved vertices, triangles and *vertex finalization tags* that indicate when a vertex will not be used anymore. Standard mesh formats do not use finalization and can therefore suffer if the mesh is larger than memory. A streaming mesh basically documents the *spatial coherence* of a dataset, which Isenburg et al. (2006a) defines as: "a correlation between the proximity in space of geometric entities and the proximity of their representations in the stream [the file]". They also demonstrate that real-world point cloud datasets often have natural spatial coherence and they exploit this coherence to compute Delaunay triangulations of massive datasets (instead of reordering the points, which is expensive); this coherence is expected since LiDAR samples are often stored in the order they were collected. The streaming ideas are very useful for certain *local* problems (e.g. interpolation and creation of grids), but unfortunately cannot be used directly (or it would be extremely challenging) for *global* problems such as visibility or flow modelling. I nonetheless use streaming of geometries to construct massive TINs and load them in the DBMS, as explained in Section 5. This permits me to naturally cluster the points in the DBMS, which leads to faster query results, as Section 6 shows.

3 A star-based data structure

A star-based data structure considers the *star* of a vertex in the triangulation as its atom.

3.1 Stars and links

Given a set S of points in the plane, its two-dimensional triangulation (a simplicial complex) is formed by a set of k-dimensional simplices σ^k, where $0 \leq k \leq 2$. A simplex is the simplest element in a given dimension: σ^0 is a vertex, σ^1 an edge, and σ^2 a triangle. Notice that a k-simplex can be constructed by simplices of lower dimensionality. Also, to simplify the notation, a simplex formed by the vertices a, b and c is simply denoted abc.

Let v be a vertex in a two-dimensional triangulation. Referring to Figure 1, the star of v, denoted star(v), consists of all the simplices that contain v. The star-shaped polygon thus formed contains all the triangles and edges incident to v, but notice that the edges and vertices disjoint from v—but still part of the triangles incident to v—are not contained in star(v). The set of simplices incident to the simplices forming star(v), but "left out" by star(v), form the link of v, denoted link(v), which is the boundary of the star-shaped polygon.

3.2 Storing stars

Blandford et al. (2005) describe a star-based data structure for representing *compactly* triangulations in two and three dimensions. Their representation uses about a factor 5 less memory than

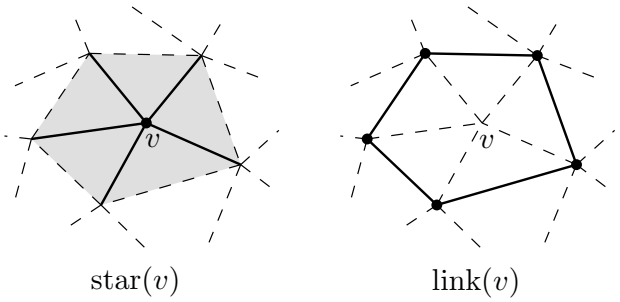
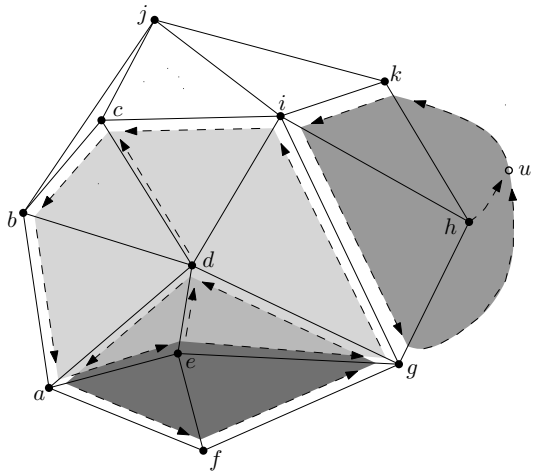


Figure 1: The star and the link of a vertex v in a triangulation.



$\text{link}(d) = [c, b, a, e, g, i]$

$\text{link}(e) = [d, a, f, g]$

$\text{link}(h) = [u, k, i, g]$

Figure 2: A triangulation of a point set S with the links of three vertices shaded (d, e and h) in grey. Observe that link(h) does not form a cycle (it is on the boundary of conv(S)), but if u, which represents the "infinity vertex", is added then a cycle is created.

traditional representations and at the same time can be queried and dynamically modified; most compressions of triangulation, such as Taubin and Rossignac (1998), are designed for storage on disk and cannot be manipulated while compressed. To achieve this compression, Blandford et al. designed a structure without any pointers. Instead, each vertex is assigned a label (an integer) and they compress based on labels: if possible they store the integers using 4 bits (they use differences between labels to achieve that) and use different tricks to keep the memory footprint low.

For an implementation in a generic DBMS, applying the compression mechanisms of Blandford et al. is not feasible. However, the uncompressed version of the star-based data structure is relevant, and is as follows. The link of a vertex v is represented as an ordered list (counter-clockwise orientation) of labels of the vertices v_i forming link(v); for simplicity I refer to that list for a vertex v in the following as link(v). Figure 2 illustrates the ideas for three vertices part of an 11-vertex triangulation. The link list is of variable length, its minimum is 3 and its theoretical maximum is the number of points in S minus 1. Notice that the length of the list is the *degree* of v (the number of incident edges to a). A triangle is formed by v and 2 consecutive v_i in the list; the length of the

list also gives the number of incident triangles to v. In Figure 2, link(d) contains for instance the triangles *dcb* and *dba*, but also *dic* since the list represents a cycle. The ordered list also permits us to represent the edges of star(v): v with a v_i; and the edges of link(v): 2 consecutive v_i. Since the list of vertices is ordered, the simplices implicitly represented are also ordered (triangles are all counter-clockwise, edges are directed). In brief, observe that for a vertex v by storing its ordered list we implicitly represent both star(v) and link(v).

3.3 Convex hull

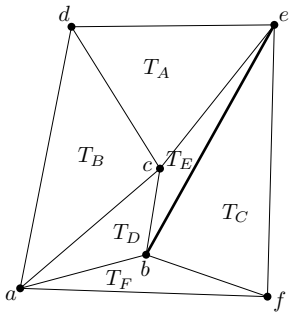
A triangulation of a point set S subdivides completely conv(S), the convex hull of S. The link of a vertex v on the boundary of conv(S) does not form a cycle, but simply a path. For example, in Figure 2, star(h) is formed by 2 triangles: *hki* and *hig*. To ensure that every link is a path, I modify the structure of Blandford et al. (2005) so that the so-called "infinity vertex" is present (Liu and Snoeyink, 2005). It is used by most implementations because it simplifies the construction of triangulation and their manipulation (Boissonnat et al., 2002; Shewchuk, 1997). With this extra vertex, two triangles, *huk* and *hgu*, are implicitly added to the triangulation.

3.4 Operations on stars

When all the stars in a triangulation are represented, the overlap between the stars gives us not only all the simplices of the triangulation, but also their adjacency and incidence relationships. Notice that a triangle is present in exactly 3 stars (its 3 vertices) and that an edge is present in 2 stars (its 2 vertices). Edges are also present in links: for instance, referring to Figure 2, the edge *gi* is present in star(g), star(i), and in link(d) and link(h) The data structure is thus akin to the half-edge or the DCEL as several topological relationships are explicitly stored.

Incidence query. Given a vertex v we can find an incident edge or triangle in constant time: an edge is formed by v and the first vertex listed in link(v); a triangle by v and the first two vertices in link(v). Visiting, in counter-clockwise order, all the edges or triangles incident to v is similarly trivial. For an edge *ab*, one incident triangle *abc* is formed by selecting the vertex c located after b in link(a). This query is performed is constant time if the degree of a is bounded. We know that the average expected degree of a vertex in a two-dimensional Delaunay triangulation (DT) where the points are distributed according to a Poisson distribution is only 6 (Okabe et al., 2000, p. 314), and the real-world datasets used for the experiments in Section 6 corroborate this.

Adjacency query. Given a triangle *abc*, finding one adjacent triangle *acd* involves selecting the vertex d located after c in the link(a).



$\text{link}(b) = [a, f, e, c]$
$\text{attr\_tr}(b) = [\emptyset, T_C, T_E, \emptyset]$
$\text{constaints}(b) = [\text{False, False, True, False}]$

$\text{link}(c) = [d, a, b, e]$
$\text{attr\_tr}(c) = [\emptyset, \emptyset, \emptyset, T_A]$
$\text{constaints}(c) = [\text{False, False, False, False}]$

Figure 3: A constrained triangulation of 6 points and 1 constraint, with the link, the attributes attached to each triangle ($attr_{tr}$), and the constrained edges (*constraints*) for 2 vertices. T_i represents the attributes attach to a triangle (e.g. its normal), the bold edge is a constrained edge, and $\varnothing$ means that nothing is stored.

3.5 Attributes and constrained triangulations

Attaching attributes to the k-simplices is also possible, although one must be careful since edges and triangles are present in multiple stars. I exploit the fact that each vertex has a unique label (which can be ordered) and attach the attributes for edges and triangles to the star of the vertex having the lowest label. For example, attributes for the triangle *abc* are stored in star(a), and that of edge *mn* in star(m). The attributes can be stored either in the same link list (alternating vertex labels with attributes), or in another list (having the same length as the list of the star). Figure 3 shows one example, notice that the attributes for the triangle *cbe* (T_E) are not stored with c, but with b.

In a similar way, we can store explicitly constrained triangulations with a star-based data structure. For each edge, a list of bits (True or False) can be stored.

4 Implementation in a DBMS/PostgreSQL

I describe in this section my implementation of the star-based data structure in a specific DBMS (PostgreSQL), but since the data structure is based solely on lists of labels, implementing it in another DBMS should be straightforward. Several engineering decisions had to be taken when implementing the structure in PostgreSQL, and I report here on the main ones.

The implementation, which is currently on-going, uses some built-in types and mechanisms of PostgreSQL. New types and new functions have been implemented as an extension to the server, these have been programmed in C.

4.1 An extension to PostgreSQL

As shown in Table 1, implementing this data structure in a DBMS is straightforward: a unique ID

ID	x	y	z	link[]	constraints[]
1	3.21	5.23	2.11	[0, 2, 44, 55, 61, 23]	[0, 1, 0, 0, 1, 0]
2	5.19	29.01	4.55	[0, 1, 7, 98, 111, 233, 222]	[0, 1, 0, 0, 0, 0, 1]
3	22.43	15.99	8.19	[4, 101, 73, 23]	[0, 0, 0, 0]
...	...	...	...	...	...
5074	221.19	15.23	37.81	[4909, 4902, 4993, 5111]	[0, 0, 0, 0]

Table 1: One example of a (fictious) table storing the star of each point in a constrained triangulation.

must be assigned to each point, and one extra column is needed for the star/link of the vertex it represents in a triangulation. This is akin to the Simple Feature paradigm (OGC, 2006) as implemented by PostGIS. For the IDs of the points, the type `bigint` (64-bit integers) is used since 32-bit integers would limit the dataset to $(2^{32})/2$ IDs (around 2 billions). For the star/link of each vertex I use, like most implementations of geometries in a DBMS[4], a variable-length type since the link of a vertex can be formed by 2 to an infinity of vertices. The built-in type `array` is used. The coordinates of each point can be stored for instance with 3 separate columns (x, y, z), or with built-in types such as PostgreSQL's `point` or PostGIS's `POINT`.

Triangle type. Since triangles are only represented implicitly, a new type `triangle` has been implemented in PostgreSQL, it is a triplet of IDs (ordered counter-clockwise). The return value of a point location query (see next section) is for instance a `triangle`.

Attributes and constraints. Since PostgreSQL does not allow different types in an array, a new column with an array of a given type must be used for storing attributes for the edges and triangles, and for storing the constraints. In Table 1, the column "constraints" is an array of booleans for instance.

Storage space. Let us look at a generic implementation of such a structure, without considering the size of the required indexes. For a TIN with n points, the implementation of that structure in a DBMS requires first storing the points: one table with n rows is required, each with 1 ID and 3 coordinates (I use double-precision floats for this implementation). For the triangulation itself, only 2 IDs per edge are required: each edge is stored twice, one for each direction; this is equal to storing 3 IDs per triangle (each triangle has 3 edges, which are shared by 2 triangle).

By comparison, consider a simple structure where triangles are formed by triplets of vertex IDs (as is the case in Oracle Spatial's `SDO_TIN`), exactly the same storage space for the points is required. The triangulation requires another table where each row has 3 IDs and represents a triangle; this structure is however not topological, and thus does not permit efficient analysis. As stated in Section 2, adding adjacency information requires having an ID for each triangle, plus 3 IDs for the

[4]To store polygons, which can have an infinity of points, PostGIS uses the well-known binary (WKB) representation; Oracle Spatial its own object type.

neighbouring triangles; the total is thus 7 IDs per triangle. Edge-based data structures require 6 IDs per edge (2 references to the vertices plus 4 to the previous and next edges), which means 9 IDs per triangle.

4.2 Spatial indexing

Perhaps the biggest advantage of a star-based structure is that a spatial index such as an R-tree is not necessary to access efficiently the points, the edges and the triangles. Instead, the triangulation itself is used to solved the following two spatial queries:

- **Point location**: given a triangulation and a query point p, determine which triangle contains p.
- **Range query**: determine which points are located inside a query bounding box.

Both queries use the adjacency relationships between the triangles to navigate in the triangulation. To ensure fast access when the star of a given star is needed, an index on the ID column of each point is used (a standard B-tree[5]); keep in mind that a B-tree is a simpler tree than an R-tree, and that its size for a given dataset is two times smaller.

Point location. I have implemented the *walking* algorithm as described in Mücke et al. (1999). It is a sub-optimal algorithm that is favoured by practitioners since it does not require auxiliary data structure and yields fast practical performances (Mücke et al., 1999; Devillers et al., 2002). As Figure 4 shows, it goes as follows: starting from a triangle σ, we move to one of the neighbours of σ (we choose one neighbour such that the query point p and σ are on each side of the edge shared by σ and its neighbour) until there is no such neighbour, then the triangle containing p is σ. To minimise the number of triangle visited, the starting triangle should be close to p. Mücke et al. (1999) investigated a "bucketing" approach where a certain number of triangles are randomly selected, and each walk starts from the closest one (selected by a simple Euclidean distance test); it is called the *jump-and-walk* method. The result of a query is a `triangle`; if p is exactly on an edge or a vertex than one randomly selected `triangle` is returned.

I have implemented a slightly modified version of that algorithm for stars where the bucketing is based on a simple regular subdivision of the spatial extent of the dataset. As Figure 4 shows, each cell of the virtual grid stores the ID of one vertex (randomly chosen) to start the walk. This information is stored in another table of the DBMS containing the 'key' of the cell (row, column) and the starting ID; a B-tree is used to index the cell 'key'. As Section 6 shows, this simple bucketing method gives good results in practice, but more advanced methods (with a quad-tree for instance) should be investigated.

[5] A hash table is in theory more efficient since no range queries on IDs are performed. However, not all DBMS has an implementation, and that of PostgreSQL is not recommended.

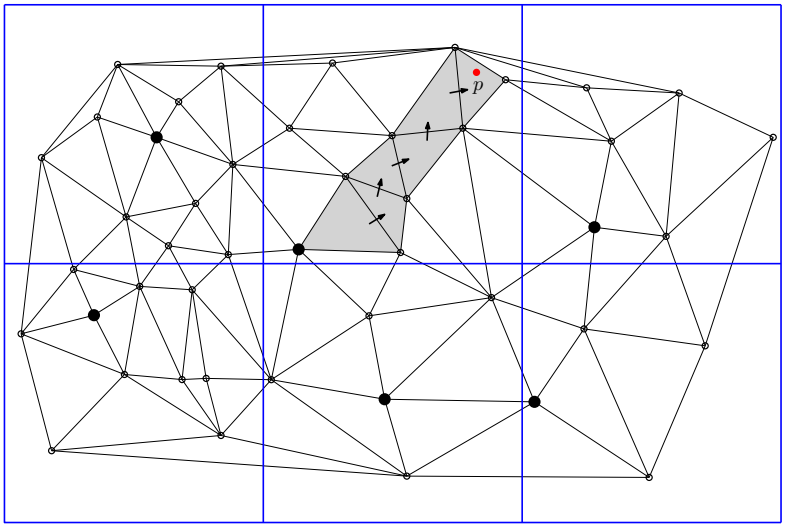


Figure 4: Walking in a triangulation to obtain the triangle containing the query point p (in red); only the grey triangles are visited. The grid cells are in blue, each one contains one and only one starting point for the walk (larger black points).

Range query. Zhu (2000) shows how the triangulation of a point set can be used to solve range queries. The idea is similar to walking, except that *marching* is used: only the triangles intersected by a line segment are visited. Figure 5 illustrates the general idea of the algorithm. First a walk is perform to obtain a (one corner of the query box Q), and then 4 different marches are performed along the edges of Q, collecting the vertices inside Q. Notice that only the vertices incident to edges crossed by Q will be selected (black points in Figure 5b). Second, a depth-first search on the subset created is performed to identify all the vertices inside Q. This is performed efficiently since the triangulation is stored and we can navigate in it.

4.3 Insertion and deletion of vertices.

Since the star-based structure is topological, inserting new points in a TIN can be done efficiently. The incremental insertion algorithm as described in Guibas and Stolfi (1985) has been implemented, and the deletion algorithm of Mostafavi et al. (2003) would be straightforward to implement. Both need a point location function, and then the updates to the TIN are performed by modifying locally the triangles (which involves modifying the stars or vertices), and by either adding or removing one row to the table of points (and thus updating the B-tree). In theory, it is assumed that the updating is done in constant time, since the degree of a vertex is exactly 6 (which means for one insertion, 6 stars will be modified in most cases). At this moment, only Delaunay triangulations without constraints are supported, but I plan to add support for constrained edges in the future. Notice here that insertion and deletion operations do *not* require a complete rebuilding of the triangulation, the triangulation is simply locally updated and queries can be performed as soon as the triangulation is updated (only the B-tree for the points needs to be updated). Other

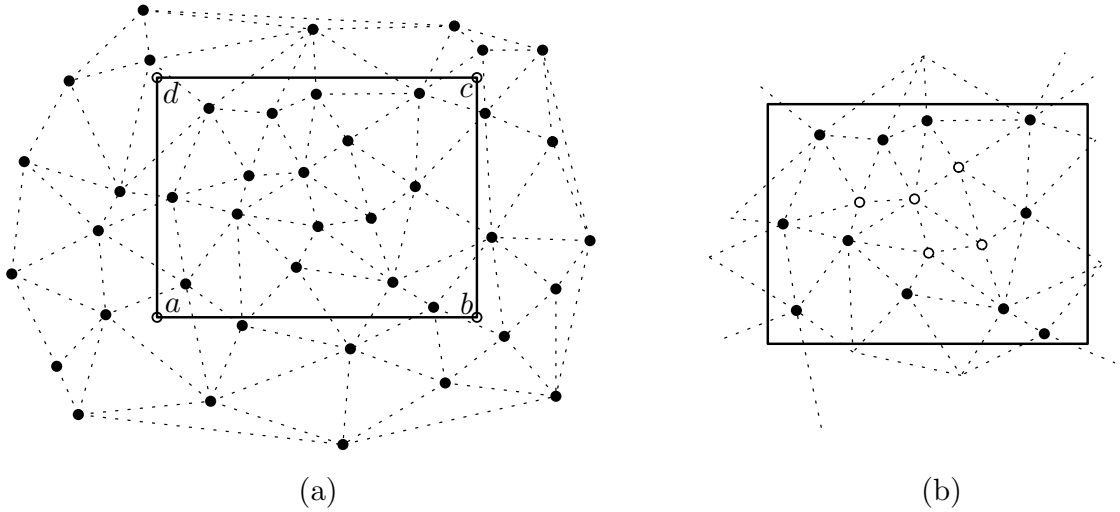


Figure 5: **(a)** The range query *abcd* performed on a point set A; a triangulation is used as a supporting data structure. **(b)** A subset of A is first obtained (black points) and then a depth-first search is performed to retrieve the points inside *abcd* whose star does not intersect the query box.

approaches, such as that of Oracle Spatial, require a complete rebuild if only one point is added to the TIN.

4.4 Spatial analysis operations

The topological data structure with the two spatial queries functions permit us to perform several common spatial analysis operations. The following are examples.

Statistics about the dataset. With all the stars stored as lists of IDs one can calculate easily general statistics about the point cloud datasets and its TIN. The degree of vertices, implemented as function, is obtained in constant time (length of the link list), and thus one can obtain with a SQL query the average, or maximum, degree for a dataset. For the obtaining the points on the convex hull, it suffices to verify if the link list contains the ID "0".

Deriving edges and triangles. While only vertices and stars are stored, it is possible to extract edges and triangles using the same trick as for attributes: given a vertex v, only extract a simplex incident to it if the ID v is the lowest. That permits us to show to the user the triangulation and to export it to another format.

Interpolation and slope. Interpolation in the TIN is performed by a walk followed by a linear interpolation on the triangle obtained. Other interpolation methods, such as weighted-average methods (Watson, 1992), could also be implemented as the adjacent triangles are readily available. Calculation of the slope at a location (x, y) is likewise performed.

Generation of profiles. The profile, along a given line, can be extracted from the TIN by first locating the starting triangle, and then *marching* as in the case of a range query and calculating the intersection between the profile line and the edges of the TIN (with the elevation linearly interpolated). If the vertices of the TIN have high spatial coherence, the marching is performed efficiently since each query (to walk from one star/triangle to the adjacent one) is performed on data that are closed on disk.

5 Construction of massive Delaunay triangulations with streaming

To compute the Delaunay triangulation of massive point cloud datasets and to populate the star-based structure in a DBMS I make use of the spatial streaming framework developed by Isenburg (Isenburg and Lindstrom, 2005; Isenburg et al., 2006a,b). They developed two modules to construct triangulations:

- `spfinalize` introduces finalization tags for point clouds: a tag documents when no more points will be inside a given region (the input dataset is tessellated into regions based on a quadtree).
- `spdelaunay2d` creates a Delaunay streaming mesh, based on the `spfinalize` input.

The particularity is that these two modules can be used in a *pipeline*, where the output of one module is the input of another (the disk is not touched). The finalization tags allow the modules to start producing their output before the whole dataset is read.

I have developed, with the Python scripting language[6], a small module (`smb2star.py`, "smb" being the binary representation of a streaming mesh) that reads as input a streaming mesh and outputs stars in the format I use for the DBMS (Table 1). Figure 6 shows where the module fits in the spatial streaming framework. It basically contains a dictionary with vertices and their temporary links (IDs of other vertices); as vertices and triangles are output from `spdelaunay2d`, they are put in that dictionary. When a vertex finalization tag is read, it means that all the triangles incident to that vertex have been read, which permits us to construct the topological cycle of triangles (and construct the link), and then to clear from memory that vertex (and its link) and write it immediately to the DBMS.

The ID assigned to each vertex is the one obtained from `spfinalize` and `spdelaunay2d` (based on the order in which they come out of the pipeline), which is not exactly the order of input. Indeed, `spfinalize` introduces more spatial coherence in datasets by reordering the points in each cell of the quadtree it creates, and release them all at once, incrementing the ID counter as they are

[6] `http://www.python.org`

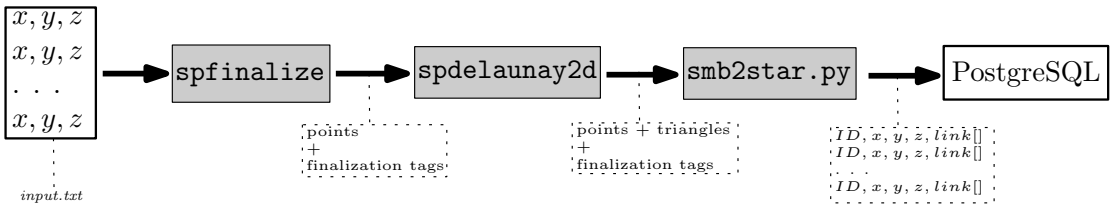


Figure 6: My streaming pipeline to create the star of each vertex. The point cloud dataset is first read from disk, then `spfinalize` and `spdelaunay2d` create a streaming mesh, which is then transformed by `smb2star.py` into a list of stars that are piped directly to PostgreSQL.

released. Also, `smb2star.py` takes care of vertices on the boundary of the convex hull: when a vertex is finalized the incident triangles are ordered in a topological cycle, if that fails the link of the vertex starts with a flag (ID "0") indicating that the other vertex IDs form a path.

Observe that the more correlation there is between the IDs and the proximity in space of the vertices, the better the DBMS will perform for queries. If vertices closed to each in space have similar IDs, they will be closed to each other on disk (and perhaps even on the same DBMS page). Queries indeed usually involves finding the star of close vertices: the closer they are in the index the faster the DBMS is likely to retrieve them.

Notice that the pipeline shown in Figure 6 will work only if the input dataset has enough spatial coherence. Isenburg and Lindstrom (2005) define the maximum number of non-finalized points at any time as the *width* of a stream, the smaller it is the better the spatial coherence is (the smaller the memory footprint of the application is too); I report in the next section on the width of the datasets used for experiments.

6 Experiments

I report in this section on experiments I ran with three real-world datasets:

1. **AHN2:** a subset of the AHN2[7] dataset, covering an area of 5kmX5km around the city of Delft, in the Netherlands. Only the 'bare-earth' part of the dataset was used[8], i.e. samples representing buildings, cars or other man-made construction were not used. This also creates a dataset whose spatial distribution is not uniform. Figure 7 shows an example of the spatial distribution, notice how houses that were removed are visible in the top-left of the figure.
2. **serpent:** the dataset called 'Serpent Mound Model LAS Data.las', freely available at `http://liblas.org/samples`;
3. **msh:** the dataset called 'Mount St Helens Nov 20 2004.las' (from `http://liblas.org/samples`) copied and translated 41 times (total 42 times), to obtain one that does not fit in memory of

[7] Actueel Hoogtebestand Nederland. `www.ahn.nl`
[8] The "gefiltered" part, in Dutch.

					degree	
	points	duplicates	triangles	convexhull	avg	max
AHN2	281 884 687	214 050	563 768 199	1 173	6.00	63
serpent	3 265 110	17 584	6 494 998	52	6.00	39
msh	283 213 392	0	566 426 669	113	6.00	141

Table 2: Details concerning the datasets used for the experiments; convexhull is the number of points that are on the boundary of the convex hull of the dataset.

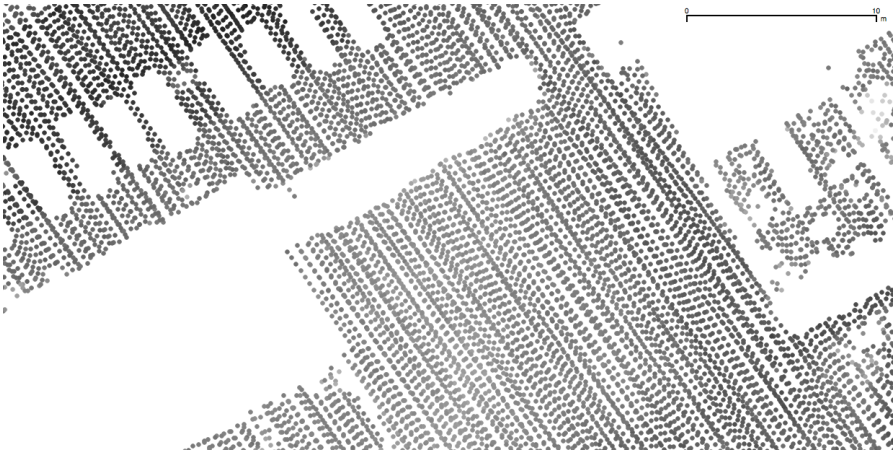


Figure 7: Spatial distribution of the **AHN2** dataset. The shades of grey for the points represent the elevation.

a commodity PC. The original dataset has regular distribution (grid), but this one has been rotated and the centre of each pixel is a sample that is triangulated.

Table 2 contains the details of the three datasets, and Figures 7, 8 and 9 show the datasets. Since at this moment `spdelaunay2d` only supports single-precision floats (32 bits), I have scaled and truncated the coordinates; that has resulted in a certain number of duplicate points (having the same (x, y) but different z values), which were removed while triangulating (the first point inserted had priority).

To compare the solution proposed in this paper with potential alternatives, I have stored the TINs of the three datasets in two data structures:

1. in the star-based data structure, with a B-tree index;
2. in PostGIS using Simple Features `Polygon`, with a GiST spatial index (Hellerstein et al., 1995).

The experiments were run on a commodity PC: an Intel Core2 Duo 3.16GHz with 4GB of main memory, running Linux/Ubuntu and PostgreSQL 8.4.4. Only the Delaunay triangulation (DT) of the points was created and stored, without any attributes or constrained.

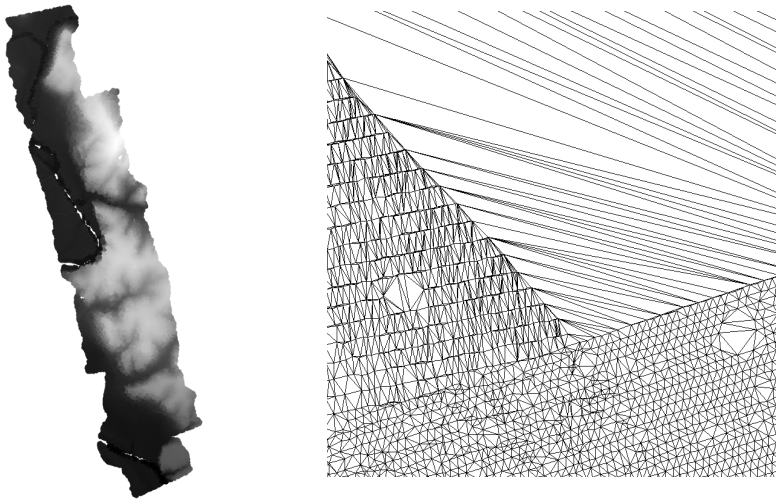


Figure 8: Left: the **serpent** dataset. Right: one part of the TIN of **serpent**.

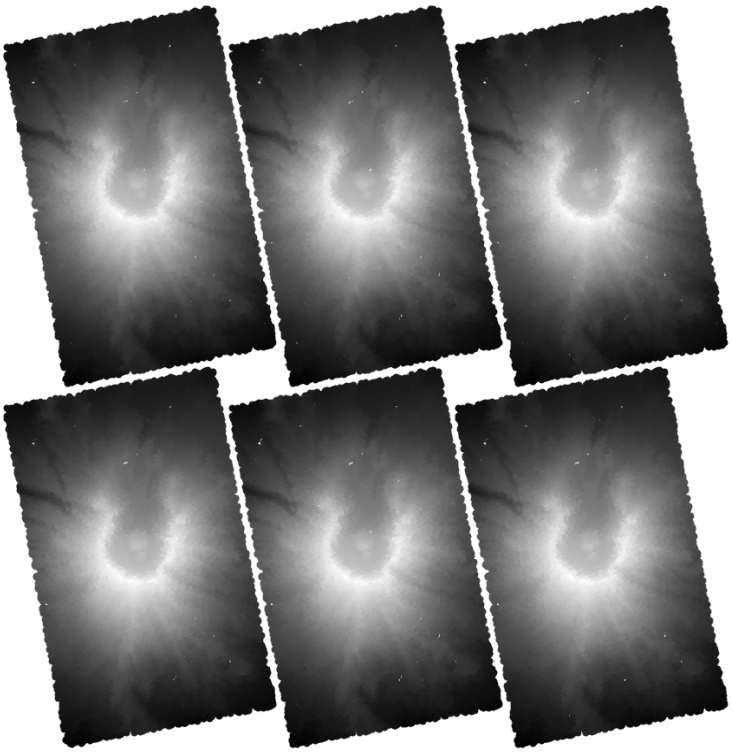


Figure 9: Six copies of the original datasets. The dataset **msh** contains 42 copies.

	star structure (min)			triangles SF (min)			width	
	construct	index	total	construct	index	total	avg	max
AHN2	178.3	24.7	203.0	285.8	619.0	904.8	34 191	53 003
serpent	1.8	0.3	2.1	2.6	6.1	8.7	3 871	7 054
msh	167.3	31.1	198.4	240.3	673.2	913.5	5 796	7 162

Table 3: For the two datasets: runtimes in minutes for the whole pipeline, the creation of the B-tree in PostgreSQL of only the triangulation (without the creation of the stars and the loading); and the maximum width of the stream.

6.1 Populating the database

The construct the stars of the input points, and to populate PostgreSQL, the pipeline shown in Figure 6 was used. An extra module `smb2sf.py` was developed to construct triangles (Simple Features) from the output of `spdelaunay2d` (it replaces `smb2star.py` in the pipeline).

The runtimes are shown in Table 3, along with the width of the stream created by the pipeline. First observe that populating the DBMS and indexing the datasets takes a significant larger amount of time for triangles in Simple Features (a factor 4.5 for **msh** and **AHN2**). While building the triangles took longer than the stars (around 1.4 times longer, since there the number of triangles is around twice as large as that of points), the main cause is the building of the GiST spatial index of PostGIS, which took 20 times more time. To illustrate, in the case of **msh**, after having populated the DBMS it took only 31min to index the stars, while the building of the spatial index took more than 11h. This is explained by the fact that the GiST is a more complex tree than a B-tree, but also by the fact that twice as much elements had to be indexed.

The average and maximum widths for the three dataset were very small: for instance out of around 283 millions points the maximum number of points in the dictionary of `smb2star.py` was only around 53 000, or 0.02% of the points. From this fact, we can conclude that the spatial coherence of the input datasets was high.

6.2 Properties of the datasets.

Table 4 shows the size of the tables in PostgreSQL once populated.

The fact that the GiST is a more complex structure than a B-tree is highlighted by the fact that it takes about 6 times more space on disk (e.g. the spatial indexes for **msh** and **AHN2** each take 29 GB, while their respective B-trees only 4.8 GB). Storing triangles in PostGIS also takes significantly more space, first because there are more triangles than points, and second because the bounding box of every triangle must also be explicitly stored for the GiST.

	star structure			triangles SF		
	table	index	total	table	index	total
AHN2	28 GB	4.8 GB	32.8 GB	64 GB	29 GB	93 GB
serpent	325 MB	56 MB	381 MB	746 MB	329 MB	1075 MB
msh	28 GB	4.8 GB	32.8 GB	64 GB	29 GB	93 GB

Table 4: Size of the tables and the indexes in PostgreSQL for the datasets.

With a star-based data structure, we can use a mix of standard SQL and functions added as an extension to query the datasets. This is how the details of the datasets in Table 2 were obtained. Two examples are shown here, the runtime of each query is also shown as an indication.

Convex hull points. Out of 281 670 637 points, 1173 are on the boundary of the convex hull. This is obtained with the function *convexhull()*, which returns true if the vertex is on the boundary of the convex hull.

```
AHN2=# select count(id) from points where is_convexhull(link) is true;
 count
-------
 1173
(1 row)
Time: 333050.861 ms
```

Degree of a vertex. The average and maximum degree of the vertices in **msh** are 6 and 141.

```
msh=# select avg(degree(link)), max(degree(link)) from points;
       avg         | max
-------------------+-----
5.9999995798221293 | 141
(1 row)
Time: 320112.537 ms
```

6.3 Point location

Properly benchmarking the point location function is a complex task that is beyond the scope of the current paper. Indeed, the DBMS performs caching between the queries and several parameters influence the results (e.g. the disk page size and the number of points in each buckets). Therefore, I report here on experiments that were ran with one bucket size and with the default page disk size of PostgreSQL, and I compare with the GiST as implemented in PostGIS.

The size of the buckets was chosen so that a relatively small amount of points is inside each buckets (around 400), irrelevant of the size of the dataset. You can see below two examples of the same query: one made with the stars (with the function *pointlocation()*) and one with PostGIS with triangles (Simple Features). With the function *pointlocation()*, the starting point is retrieved with one query from the buckets table, and then a small number of triangles (31 in that case) are visited (one query per triangle in the point table) to find the triangle containing the query point. With PostGIS, the spatial index is first used to retrieve all the triangles whose bounding box contains the query point, and then a point-in-polygon test is performed for each candidate.

During my experiments, for all the datasets, most queries were answered within 100ms, which is of the same order of magnitude as when the GiST is used. Once a query has been performed for a given location, running another query close in space to the previous speeds up greatly the query time (because the points are most likely in the same disk page) and query times of 1ms or 2ms are common. The spatial index of PostGIS has the same behaviour.

```
msh=# select pointlocation(611, 545);
WARNING:  closest pt to start is 64467668
WARNING:  start distance is 1.329440
WARNING:  # of triangles visited is 31
             pl
------------------------------
 (64468127,64468111,64468110)
(1 row)
Time: 73.897 ms

msh=# select astext(geom) from trianglesf
where st_intersects(geom, ST_GeomFromText('POINT(611 445)'));
                          astext
-----------------------------------------------------------------
 POLYGON((611.09 444.91,611 445.01,610.98 444.91,611.09 444.91))
(1 row)
Time: 67.255 ms
```

6.4 Linear interpolation and slope.

The performance of several spatial analysis operations on terrains are based on the performance of *pointlocation()*: most of them start with it, and then adjacent triangles or points in the vicinity are retrieved. Interpolation and slope are one example: their runtime is nearly identical to that of *pointlocation()*.

6.5 Incremental updating

With the star-based structure, three steps are required to insert a point p: (1) *pointlocation(p)*; (2) inserting p in the point table (and updating the B-tree); (3) updating the stars of vertices in the vinicity of p. Step 2 relies on the implementation of PostgreSQL, which during the experiments performed very quickly (often < 50ms). Step 3 is also efficient since it involves modifying to points (adding the ID of p). Below is one example of a query where a new point (with elevation 100) was added to **msh**; the ID of the new vertex is returned.

```
msh=# select insertpt (611, 545, 100);
WARNING:  closest pt to start is 64467668
WARNING:  start distance is 1.329440
WARNING:  # of triangles visited is 31
insertpt
-----------
283213393
(1 row)
Time: 94.472 ms
```

With the Simple Feature structure, inserting a new point p is more complex since triangles need to be deleted and then replaced by new ones, and, as a consequence, the spatial index needs to be updated. Indeed, simply inserting a new point p (without updating for the Delaunay criterion) means deleting the triangle that contains it, and then adding three new triangles (four changes to the spatial index, including one deletion). The current implementation of the GiST in PostgreSQL is rather slow when one element is deleted: for the dataset **serpent** it took around 10s, while for **msh** it took around 10min. The insertions of new geometries were however performed quickly (200-300ms);

6.6 Profile

Perhaps the biggest advantages of having a topological data structure is highlighted when one wants to make a profile in the terrain/TIN. When triangles are stored as Simple Features and a spatial index is used the operation is very costly since all the triangles whose bounding box intersect that of the query line are retrieved, then intersection tests are made and finally these intersections must be sorted (see Figure 10). The query time is therefore heavily dependent on the length of the profile line.

With a star-based data structure, the profile is also dependent on the number of triangle edges crossed by the profile line, but performs much faster because the topological relationships between the triangles are stored (bounding boxes are not used). Below is one example of a query where simply the triangles crossed by a profile line are retrieved. With the star-based structure, first a *pointlocation()* is performed (same example as above), and then the marching starts and returns how many edges were crossed. Observe here that if we subtract 100ms for the point location,

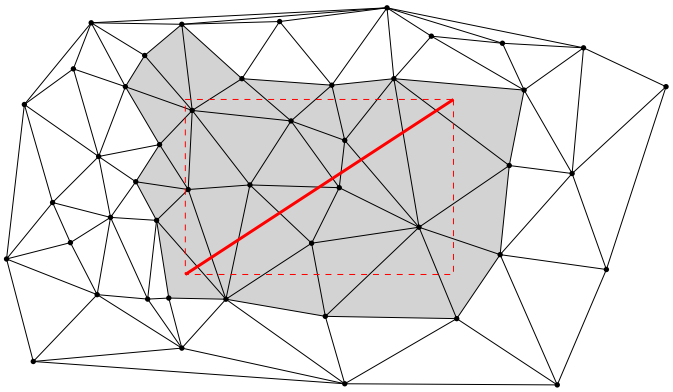


Figure 10: When a spatial index is used to identify the triangles crossed by a profile line (the red line), all the triangles whose bounding box overlap that of the profile line (the grey ones) are retrieve as candidate for the intersection test.

1196 queries in the point table were made in around 500ms, which shows again that the spatial coherence is high. With the explicit triangles, the same query takes significantly more time, and the intersections are not sorted yet along the profile line. The longer the profile line is, the bigger the difference between the two method is.

```
msh=# select profile_count_intersections(611, 545, 651, 595);
WARNING:  closest pt to start is 64467668
WARNING:  start distance is 1.329440
WARNING:  # of triangles visited is 31
 profile_count_intersections
-----------------------------
          1196
(1 row)
Time: 586.215 ms

msh=# select count(geom) from trianglesf where
st_crosses(geom, ST_GeomFromText('LINESTRING(611 545, 651 595)'));
 count
-------
  1196
(1 row)
Time: 17796.630 ms
```

7 Conclusions

Handling and analysing massive point clouds can be tackled by several different ways. I have investigated in this paper one the them: storing the TIN in a DBMS and relying on the DBMS for memory management. The star-based data structure I have proposed is significantly different from alternatives and has in my opinion several advantages over existing structures.

First, it is compact. If we consider only the space for the points and triangles, it uses no more space than a simple structure in which triangles are formed by triplets of vertex labels, and yet the stars permit us to have access to adjacency and incidence relationships between the triangles. I have shown that, based on a PostgreSQL/PostGIS implementation, if the walking method is used instead of a spatial index, then the structure with its B-tree index is around the same size as simply the spatial index for the triangles in Simple Features.

Second, it is very simple to implement in a DBMS as only lists/arrays of labels are needed. One only needs to add an extra column to a point table to store a TIN—this is inline with the popular Simple Feature paradigm popular for handling geographical objects in DBMSs.

Third, because it is topological, the TIN can be used as the supporting structure for spatial indexing the triangles (with the walking method). The walking method is only efficient if the points have good spatial coherence, so that successive queries in the B-tree can benefit from the caching mechanisms of the DBMS. I have shown that for real-world datasets, the spatial coherence is very good and operations such as extracting the profile from a TIN are very efficient, an order of magnitude faster than using a spatial index with bounding boxes.

Fourth, it is flexible: we can easily combine the star concept with other ideas and use another spatial index (since the star of a vertex can be "moved" with it, it becomes a simple attribute). As future work, I plan to investigate the use of the star-based structure with Oracle Spatial *SDO_PC*.

Fifth, although I have not yet implemented tools to construct them, constrained triangulations can also be stored with a star-based structure; attaching attributes to any simplex of a TIN is also possible.

Finally, it should noticed that the star-based structure can be used to store in a DBMS the triangulation of a closed volume (e.g. of a sphere), but that the *walking* unfortunately cannot be used as a spatial index since it works on the projection of the triangulation on the $x - y$ plane. Also, the star concept extends to three dimension (the stars of an *edge* is formed by the union of all its incident tetrahedra). As a consequence, the idea of storing stars of edges in 3D would permit us to store efficiently tetrahedralizations, and would offer an alternative to the structure of Penninga (2008) for representing 3D city models.

References

Agarwal PK, Arge L, Mølhave T, and Sadri B (2008). I/O-efficient algorithms for computing contours on a terrain. In *Proceedings 24th Annual Symposium on Computational Geometry*, pages 129–138. ACM Press, College Park, MD, USA.

Agarwal PK, Arge L, and Yi K (2005). I/O-efficient construction of constrained Delaunay triangulations. In *Proceedings 13th European Symposium on Algorithms*, pages 355–366. Palma de Mallorca, Spain.

Arge L, Danner A, Haverkort H, and Zeh N (2006). I/O-efficient hierarchical watershed decomposition of grid terrain models. In Reidl A, Kainz W, and Elmes G, editors, *Progress in Spatial Data Handling—12th International Symposium on Spatial Data Handling*, pages 825–844. Springer-Verlag.

Bentley JL (1990). K-d trees for semidynamic point sets. In *Proceedings 6th annual symposium on Computational geometry*, pages 187–197. ACM. doi:http://doi.acm.org/10.1145/98524.98564.

Blandford DK, Blelloch GE, Cardoze DE, and Kadow C (2005). Compact representations of simplicial meshes in two and three dimensions. *International Journal of Computational Geometry and Applications*, 15(1):3–24.

Boissonnat JD, Devillers O, Pion S, Teillaud M, and Yvinec M (2002). Triangulations in CGAL. *Computational Geometry—Theory and Applications*, 22:5–19.

de Berg M, van Kreveld M, Overmars M, and Schwarzkopf O (2000). *Computational geometry: Algorithms and applications*. Springer-Verlag, Berlin, second edition.

De Floriani L and Magillo P (2002). Regular and irregular multi-resolution terrain models: A comparison. In *Proceedings 10th ACM International Symposium on Advances in GIS*, pages 143–148. McLean, Virginia, USA.

Devillers O, Pion S, and Teillaud M (2002). Walking in a triangulation. *International Journal of Foundations of Computer Science*, 13(2):181–199.

Finnegan DC and Smith M (2010). Managing LiDAR topography using Oracle and open source geospatial software. In *Proceedings GeoWeb 2010*. Vancouver, Canada.

Garland M and Heckbert PS (1995). Fast polygonal approximation of terrain and height fields. Technical Report CMU-CS-95-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA.

Guibas LJ and Stolfi J (1985). Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123.

Hellerstein JM, Naughton JF, and Pfeffer A (1995). Generalized search trees for database systems. In *Proceedings 21st International Conference on Very Large Data Bases*.

Isenburg M and Lindstrom P (2005). Streaming meshes. In *Proceedings Visualization '05*, pages 231–238.

Isenburg M, Liu Y, Shewchuk JR, and Snoeyink J (2006a). Streaming computation of Delaunay triangulations. *ACM Transactions on Graphics*, 25(3):1049–1056.

Isenburg M, Liu Y, Shewchuk JR, Snoeyink J, and Thirion T (2006b). Generating raster DEM from mass points via TIN streaming. In *Geographic Information Science—GIScience 2006*, volume 4197 of *Lecture Notes in Computer Science*, pages 186–198. Münster, Germany.

Kumler MP (1994). An intensive comparison of triangulated irregular networks (TINs) and digital elevation models (DEMs). *Cartographica*, 31(2).

Liu Y and Snoeyink J (2005). The "far away point" for Delaunay diagram computation in $\mathbb{E}^d$. In *Proceedings 2nd International Symposium on Voronoi Diagrams in Science and Engineering*, pages 236–243. Seoul, Korea.

Mäntylä M (1988). *An introduction to solid modeling*. Computer Science Press, New York, USA.

Mostafavi MA, Gold CM, and Dakowicz M (2003). Delete and insert operations in Voronoi/Delaunay methods and applications. *Computers & Geosciences*, 29(4):523–530.

Mücke EP, Saias I, and Zhu B (1999). Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulations. *Computational Geometry—Theory and Applications*, 12:63–83.

Muller DE and Preparata FP (1978). Finding the intersection of two convex polyhedra. *Theoretical Computer Science*, 7:217–236.

OGC (2006). OpenGIS implementation specification for geographic information—simple feature access. Open Geospatial Consortium inc. Document 06-103r3.

Okabe A, Boots B, Sugihara K, and Chiu SN (2000). *Spatial tessellations: Concepts and applications of Voronoi diagrams*. John Wiley and Sons, second edition.

Oracle (2012). Oracle database 11g documentation.

Peng W, Petrovic D, and Crawford C (2004). Handling large terrain data in GIS. In *ISPRS 2004—XXth Congress*, volume IV, pages 281–286. Istanbul, Turkey.

Penninga F (2008). *3D Topography: A Simplicial Complex-based Solution in a Spatial DBMS*. Ph.D. thesis, Delft University of Technology, Delft, the Netherlands.

Ramakrishnan R and Gehrke J (2001). *Database Management Systems*. McGraw-Hill Science/Engineering/Math.

Ravada S, Horhammer M, and Kazar BM (2010). Point cloud: Storage, loading, and visualization. National Science Foundation TeraGrid Workshop on Cyber-GIS, Washington, DC, USA.

Shewchuk JR (1997). *Delaunay Refinement Mesh Generation*. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburg, USA.

Taubin G and Rossignac J (1998). Geometric compression through topological surgery. *ACM Transactions on Graphics*, 17(2):84–115. doi:http://doi.acm.org/10.1145/274363.274365.

van Oosterom P (1999). Spatial access methods. In Longley PA, Goodchild MF, Maguire DJ, and Rhind DW, editors, *Geographical Information Systems*, volume 1, pages 385–400. John Wiley & Sons.

van Oosterom P, Stoter J, Quak W, and Zlatanova S (2002). The balance between geometry and topology. In Richardson D and van Oosterom P, editors, *Advances in Spatial Data Handling—10th International Symposium on Spatial Data Handling*, pages 209–224. Springer.

Vitter JS (2001). External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2):209–271.

Watson DF (1992). *Contouring: A guide to the analysis and display of spatial data*. Pergamon Press, Oxford, UK.

Zhu B (2000). Fast range searching with Delaunay triangulations. *GeoInformatica*, 4(3):317–334.