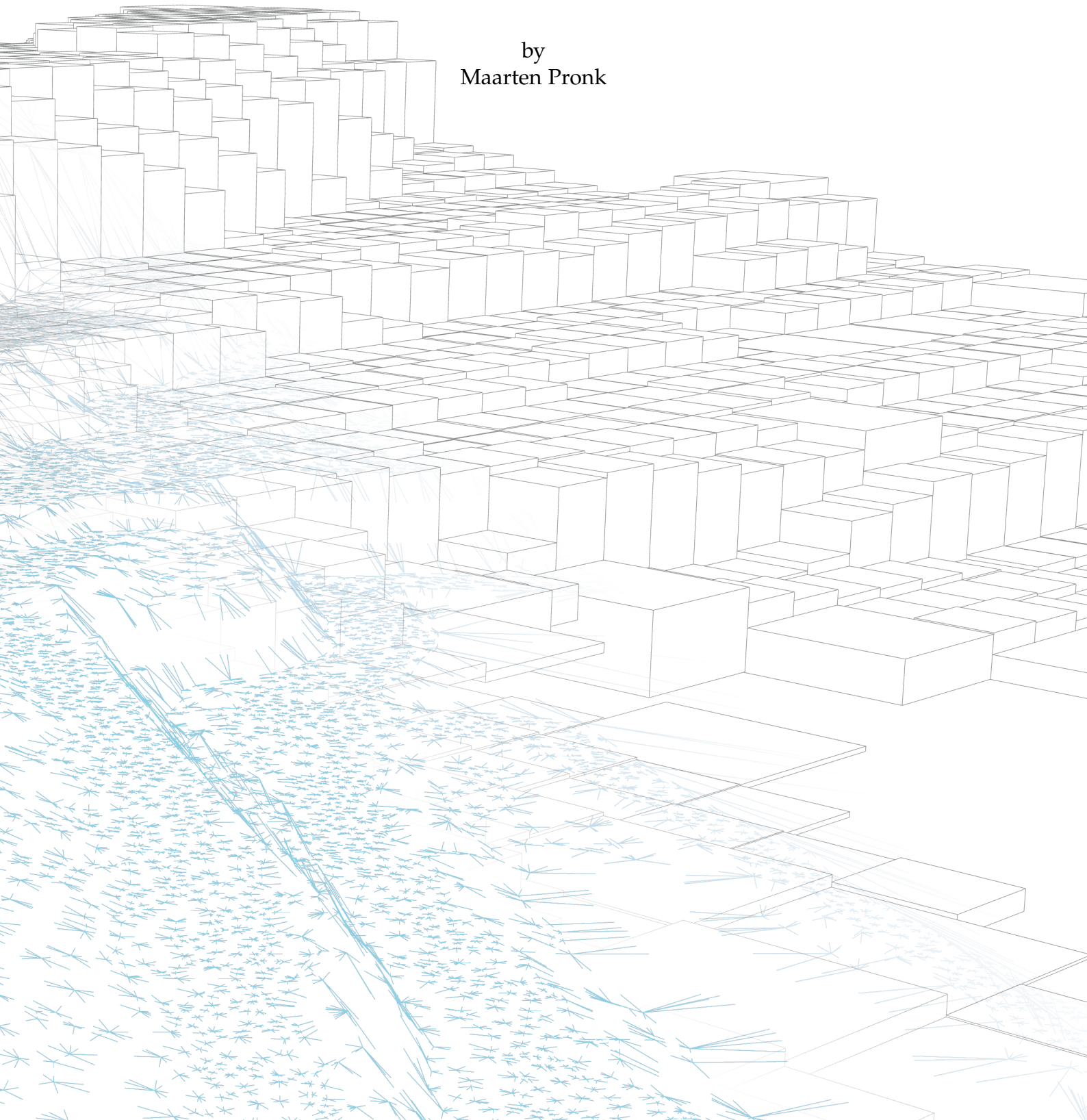
STORING MASSIVE TINS IN A DBMS

A comparison and a prototype implementation of the multistar approach

by
Maarten Pronk



STORING MASSIVE TINS IN A DBMS

A comparison and a prototype implementation of the multistar approach

A thesis submitted to the Delft University of Technology in partial fulfillment of the requirements for the degree of

Master of Science in Geomatics

by

Maarten Pronk

June 2015

Maarten Pronk: *Storing Massive TINs in a DBMS: A comparison and a prototype implementation of the multistar approach* (2015)

The work in this thesis was made in the:

TU Delft MSc Geomatics
Department of Urbanism
Faculty of Architecture & the Built Environment
Delft University of Technology

Supervisors:	Prof. Dr. Jantien Stoter
	Dr. Hugo Ledoux
Co-reader:	Dr. Ir. Martijn Meijers

‘We live in a society exquisitely dependent on science and technology, in which hardly anyone knows anything about science and technology.’

- Carl Sagan

ABSTRACT

Solutions have been introduced to handle massive point clouds in Database Management Systems (DBMS), namely by Oracle in 2011 and postgreSQL in 2013. Many common operations on these massive point clouds require knowledge about the original surface in order to analyse them. A possible method to recreate a representation of the original surface is reducing a point cloud to a 2.5D structure such as a Triangular Irregular Network (TIN). This thesis explores the possibilities to store such massive TINs in a DBMS and presents criteria to define an efficient approach. The term 'efficient' is defined by the size of the data structure and the performance of spatial queries.

Criteria for an efficient approach are developed by reviewing existing literature and comparisons of existing implementations. These criteria include: (1) the explicit storage of nodes of TIN, preventing duplicate information and resulting in small data structures. (2) the use of atomic functions such as slope, aspect and degree in order to enable analysis of the TIN inside the database. (3) the storage of topological relationships of the TIN, which is used for both atomic functions and spatial access to the TIN. (4) the use of buckets to split the TIN into non-massive sections that can be processed one at a time. A very large spatial index on each element of a TIN is therefore not needed. An index is only applied on the extents of the bucket. The TIN inside the bucket can be traversed using the topological relationships stored.

The main problem encountered is to combine solutions for massive datasets with a TIN data structure, such as the use of buckets with a topological data structure. These problems are overcome by a novel data structure to store TINs in a DBMS called the Multistar. This data structure implements the aforementioned criteria for an efficient approach.

The Multistar is compared in practice with the two existing database structures: the SDO_TIN by Oracle and the Simple Feature TIN type implemented in the PostGIS extension of PostgreSQL. This thesis shows that current implementations are not usable for both storing and accessing a massive TIN of 370M points. The Multistar is one of the smallest structures available and outperforms the other data structures significantly on massive TINs.

ACKNOWLEDGEMENTS

First of all many thanks to Hugo Ledoux, my supervisor for this thesis, for his comments and guidance. Thanks as well to the other mentors Jantien Stoter and Martijn Meijers, for their insight and questions.

Thanks to Tamara, who survived my graduation. Another Hugo helped me with useful insights into computer science and Joyce helped me with proofreading my thesis.

Finally, thanks to André Miede, for his ClassicThesis style, from which Lorenzo Pantieri, derived the ArsClassica template used in this thesis.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACRONYMS

AHN2	Actual elevation information of the Netherlands 1, 10, 53, 71
blob	Binary Large Object 31
CDT	Conforming Delaunay Triangulation 11, 33, 51
DBMS	Database Management Systems . . . vii, 3, 9, 13, 22, 25, 69
DEM	Digital Elevation Model 11, 13, 15
DT	Delaunay Triangulation 1, 2, 10–13, 16, 26, 57, 74
DTM	Digital Terrain Model 1, 3, 5, 11, 13, 69
GIS	Geographic Information System 1
SF	Simple Feature (Access) 28, 43, 60, 66, 68, 70–72
srid	Spatial Reference IDentifier 32
TIN	Triangular Irregular Network . . . . vii, 1, 9, 15, 23, 69, 83
WKB	Well Known Binary 28, 56, 92
WKT	Well Known Text 28

1 INTRODUCTION

Many practical applications of massive point cloud data require easy access and analysis of such point clouds. These operations often require an intermediary TIN. While massive datasets are already hard to acquire because of their sheer size, their derived TINs are even less likely to be used and distributed. Accessible database storage of such massive TINs enables easier access, analysis and distribution for those who work with point cloud data. Overall the use of TINs can be considered irrelevant to the general public. However, when used in a scientific setting such as in flood modelling, it can suddenly become relevant for the general public in the Netherlands.

The scientific relevance of this subject is made clear by the fact that a PhD position is available on this specific topic, which was made possible by a grant. This research is related to the open questions of how to handle massive datasets as well as the ongoing research in creating a data structure for a 3D Geographic Information System (GIS) in a database.

1.1 PROBLEM STATEMENT

In March 2014 the elevation data set of the Netherlands (AHN2) was made publicly available. The AHN2 contains around ~640 billion points and is an example of a massive point cloud dataset. Both the AHN2 and similar massive[1] pointclouds need to be stored and analysed. Database solutions have been introduced to handle these large amounts of data, including a pointcloud storage type produced by Oracle in 2011 and postgreSQL which has been extended in 2013.

In a point cloud as seen in Figure 1a, the elevation and thus surface between the collected sample of points is unknown. Many common operations on a point cloud require knowledge about the original surface in order to analyse it. Such operations include the creation of a grid [Isenburg et al., 2006b], viewshed analysis, watershed analysis [Lyon, 2003], volume calculations, slope and aspect calculations, among other analyses [Wilson and Gallant, 2000]. The original surface can never be reconstructed from a sample of elevation points such as a point cloud. However, there are several methods available to recreate the best possible representation of the original surface through spatial interpolation.

A common method of recreating a representation of the original surface is reducing a point cloud to a 2.5D structure such as a triangulation. This is called a Triangulated Irregular Network [Peucker et al., 1978] and if applicable is often used as a Digital Terrain Model (DTM). A DT (formally defined in Chapter 2) is often used for this purpose. The result can be seen in Figure 1b. In this figure the space between the original elevation points is defined by triangles. The elevation or other properties of the surface at that

1 I define massive as: 'multiple times larger than the amount of random access memory in a conventional computer'

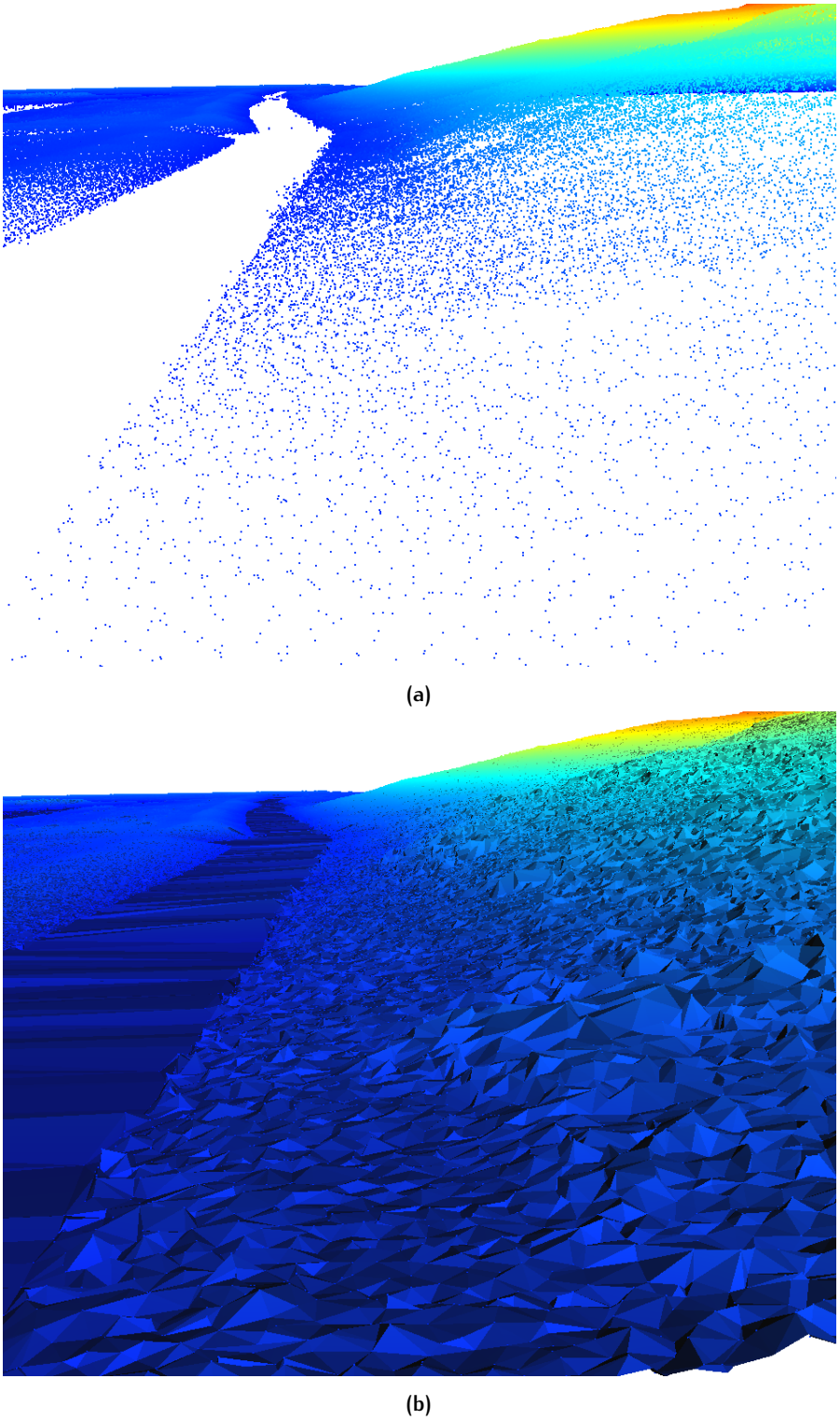


Figure 1: Point cloud and derived TIN. (a) A point cloud `demo.las` with a clear lack of information between the elevation points. (b) The derived TIN using a Delaunay Triangulation (DT) providing a reconstruction of the original surface.

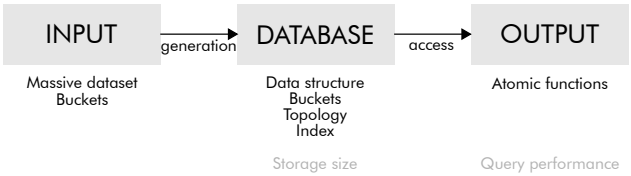


Figure 2: Conceptual scheme.

point can be derived through interpolation on these triangles. Triangulation is one of two principal methods of representing a network of elevation data. [Moore et al., 1991], the other being a grid or a Regular Network, and both methods have their advantages and disadvantages [Kumler, 1994].

Normally the triangulation is done on the fly from a pointcloud or stored as files on disk instead of a database. However, since point clouds are now often stored in databases [2] [van Oosterom et al., 2015], it could be more convenient to store the TIN in a database. This would save time and computing power and offer database advantages, such as security, scalability, versioning, integrity, constraints etc. [Elmasri and Navathe, 2006].

However, storing a TIN in a DBMS is much more complicated than storing points in a database. Both the storage of the TIN needs to be addressed, as well as the handling of massive datasets. A DTM in a database requires both topology and geometry [Fritsch, 1996] linked in a DBMS which is an open area of research [Zlatanova et al., 2004]. While the massive size of pointclouds is already a problem on its own [Vitter, 2001], storing the derived TIN in a database increases the storage size even more, since there are roughly two times more triangles than there are points in a Delaunay Triangulation [Berg de et al., 2000]. Solutions that solve the problem of massive datasets, such as tiling the point cloud, present problems for the TIN data structure: While point clouds can be arbitrarily divided into smaller buckets, this breaks the topology of a TIN.

1.2 RESEARCH QUESTION AND OBJECTIVES

Massive TIN storage in a DBMS thus presents several problems to which solutions should be compatible with as well. In Figure 2 these problems are divided in three distinct steps: the input of the database, the data structure in the database itself and its output. Problems such as the data structure, functions needed and the generation of a massive TIN are all part of an approach that stores massive TINs in a DBMS. The research question that follows is:

What are efficient approaches to store massive TINs in a DBMS?

Efficient approaches are defined by a decrease in storage size and an increase in performance compared to other, existing solutions. The following aspects cover the input, output and the data structure itself and are used to compare efficient approaches.

- Storage size of data structure
- Storage size of index

2 see https://github.com/pgpointcloud/pointcloud

- Performance of spatial queries
- Availability of atomic functions
- Loading time of the TIN, including construction

The following subquestions that need to be answered in order to create a comprehensive overview of approaches. These questions all relate to the problems touched upon in Section 1.1 and stated in Figure 2.

- What are reasons for creating buckets in a DBMS and what are the trade offs?
- Which spatial index is the most efficient?
- Which atomic functions are needed for a TIN in a DBMS?
- How can the topological relationships of a TIN be exploited in these functions?

The main objective of this thesis is to compare different approaches to store massive TINs in a DBMS, both theoretically and practically. Theoretically in their data structure and practically in their usage and performance. A prototype will be built to combine the best approaches found.

1.3 METHODOLOGY

The methodology used in order to answer the research questions can be seen in Figure 3. A literature study will be conducted to create an overview of the uses of a TIN and TIN data structures, from which the requirements for an hypothetically efficient approach in theory are derived. Also, an overview of current approaches to store TINs in a database will be made and compared theoretically with the requirements found in the literature study. To compare current implementations practically, one or more prototype(s) will be designed and implemented, based on the hypothetically efficient approach. Finally these prototypes will then be tested and compared with existing solutions. The data gathered from these tests and comparisons will give more insight into how efficient current implementations are and if these implementations are found lacking, as well as a direction for new research, based on the extent in which the efficient approach and prototype are either validated or invalidated.

The following tasks are planned in order to answer the aforementioned research questions.

- Study TIN, its topology and its usages.
- Study the loading/storage of data with sorting and splitting.
- Study spatial indexes.
- Design and implement an extension prototype for a TIN in postgreSQL.
- Implement prototype(s) with buckets based storage such as postgreSQL/Oracle.

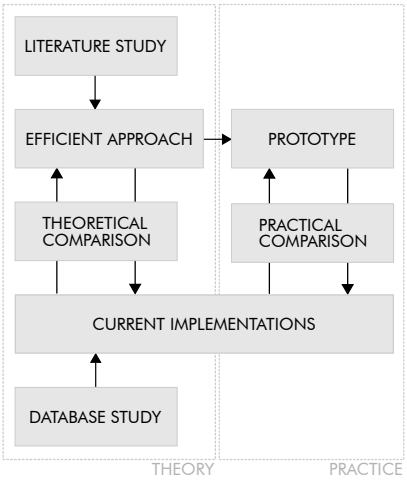


Figure 3: Methodology.

- Compare the results of these prototypes with other implementations in theory and in practice, based on the real-world datasets provided in section Section 1.4

1.4 USE CASE

The following use case is considered: A massive static TIN, computed from a tile of the AHN2 dataset,is stored in a database. In the database the TIN will be used as a DTM, with possible usages as the construction of rasters or calculating aspect and slope [Wilson and Gallant, 2000].

Some of the AHN2 data has been downloaded for processing. The following files, g37en1.laz and g37en2.laz are used for creating a TIN. These tiles cover the area around Delft with an area of 10 by 6.25 km, containing a total of 770 million points. Two smaller datasets are used for testing, demo.las and rijswijk.las, the extents of which are shown in Figure 4. Demo.las contains 5 million points and rijswijk.las contains 320 thousand points. Both datasets are used for smaller experiments. Lastools with the blast extension[3] is used to create a TIN.

1.5 SCOPE

An *efficient* storage method as defined in the research question has to be quantifiable, at least in the following standard ways: Data loading, storage size and query times, These standards will be compared to other existing solutions. Nonetheless, non database solutions do also exist and can sometimes be faster [van Oosterom et al., 2015]. Yet database solutions are preferable due to their inherent properties, such as multi user support, data integrity, spatial indexes and spatial operations with other geometries [Elmasri and Navathe, 2006], as well as a standard tool for every GIS framework and users. Non database solutions will therefore not be taken into account. The scope is limited by the following items:

3 see http://www.cs.unc.edu/~isenburg/lastools/

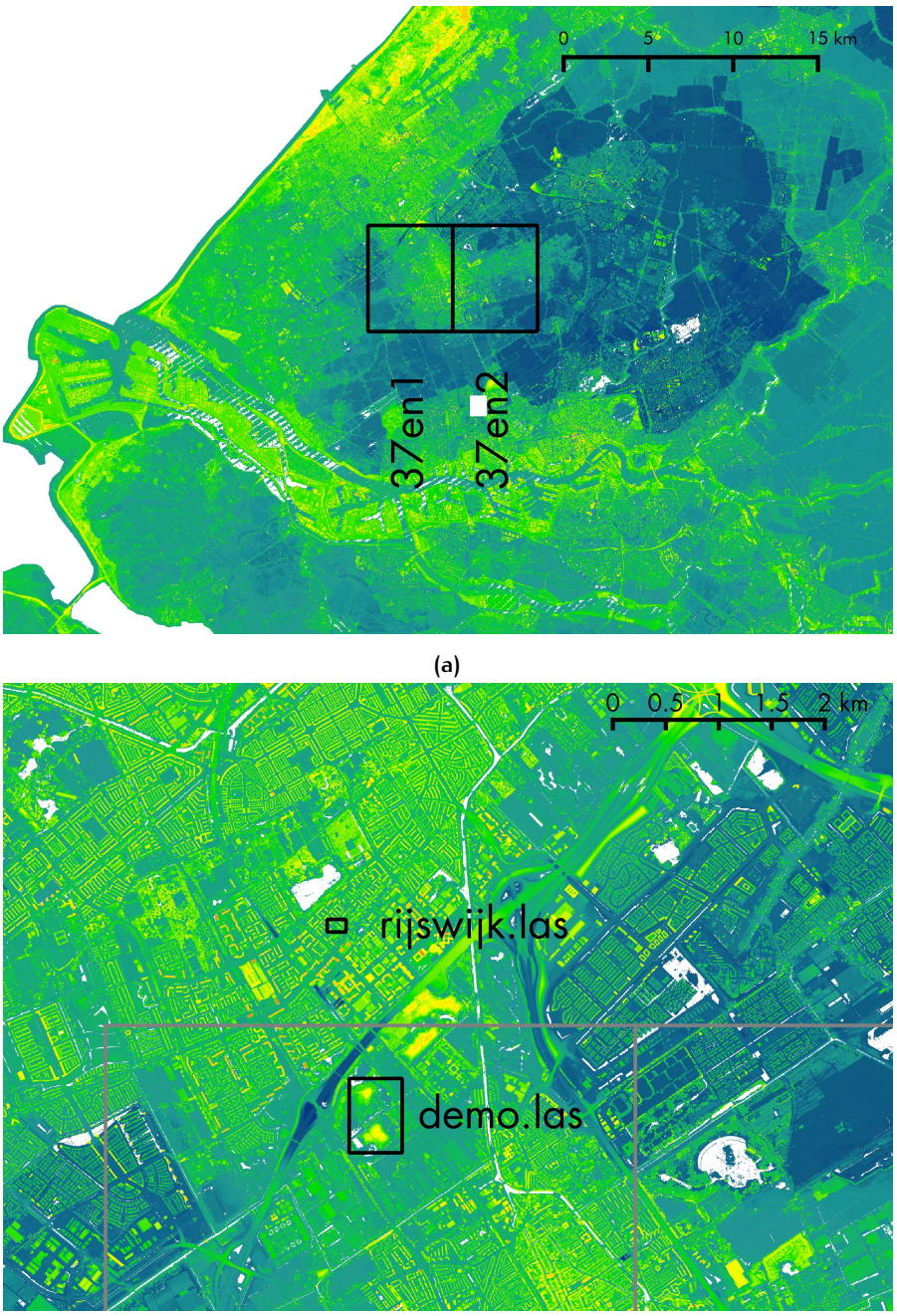


Figure 4: Extents of datasets used (a) Showing three tile extents. 37en1 and 37en2 are used for the filtered AHN2 dataset (7GB each), the area around Delft. (b) Extents for the smaller datasets using the filtered AHN2 dataset. demo.las is a park area at 105MB, rijswijk.las is 6.4MB.

- DBMS used for prototyping will be limited to PostgreSQL, as it is open source and well known and used for GIS operations.
- TIN is considered as a given, i.e. not constructed in DBMS.
- TIN is a Delaunay triangulation.
- Data is static, no updates are done within the database.

1.6 OUTLINE

The necessary background information is provided in Chapter 2 in which the TIN and its usages will be described, from which criteria for TIN structures in a DBMS follow. The hypothesis is that these criteria form an efficient approach. In Chapter 3 a comparison is made between the current existing solutions in theory, compared against the criteria found in the previous chapter. The related work on the input and output of massive TINs is also discussed in this chapter. Chapter 4 describes the proposed Multistar approach based on the hypothetically efficient approach found in the previous Chapter 3. Possible solutions to problems such as tiling a TIN will be discussed in depth, with motivation as to which approach is chosen. The implementation of this approach in the PostgreSQL database is found in Chapter 5 in which it is also benchmarked and compared to the other structures in practice. Conclusions with a discussion can be found in Chapter 6, as well as open research to be done in the future. The first Appendix A is a personal reflection on the thesis as a project in relation to the Master of Geomatics. Documentation (Appendix B) and code (Appendices C and D) of the implemented programs can be found in the other appendices, as well as the exact queries used for experimenting in Appendix E.

2 THEORY AND BACKGROUND INFORMATION

This chapter explores the definitions, uses and data structures of TINs, in order to formulate criteria for massive TIN storage. The *hypothesis* is that with these criteria, massive TIN storage in DBMS is feasible and more efficient than current database solutions.

2.1 FIELDS AND TESSELLATIONS

When storing and thus describing geographic information (in a DBMS), the question arises how to describe this information. Two main classes exists, the *field*based model and the *object* based model [Worboys and Duckham, 2004]. The field model treats geographic information as space (a set of locations) with attributes, such as elevation. The object model, on the other hand, reverses this concept and treats the attribute as a set of discrete entities with a specific location, such as a house.

This dichotomy becomes blurred when the density (field) of houses (object) is described, or when elevation (field) is assigned a category as below sea-level (object) [Worboys and Duckham, 2004]. Examples of such cases are encountered in Chapter 4.

In order to describe a field in computers, a finite part of it is often discretized into cells, a set of basic units, called a tessellation [Burrough and McDonnell, 1998]. Such a tessellation can be *regular* or *irregular*, based on the fact if the partition of the plane is done by the exact same object (regular) or not (irregular). The most well known regular tessellation is the grid, or a raster, based on a simple square, seen in everyday life as pixels. Only two other regular tessellations exist, those of triangles and hexagons [Worboys and Duckham, 2004].

Irregular tessellations consist of non regular polygons, of which the most known example is a TIN, consisting of irregular triangles.

2.2 REPRESENTING TERRAIN AS A TIN

A terrain is a model of a part of the surface of the earth, a two dimensional approximation in three dimensional space, because every point on the terrain is assigned a height [Berg de et al., 2000]. In reality, however, we only have a finite set of sample points at which the height was measured. The height of the other unsampled points on the terrain thus have to be estimated through spatial interpolation.

There are many other methods for fitting elevation surfaces to point data, see Moore et al. [1991].

Spatial interpolation is often done linearly based on a regular or irregular tessellation. To find the value of an unsampled point the polygon in which it lies is found, from which linear function the interpolated value can be established.

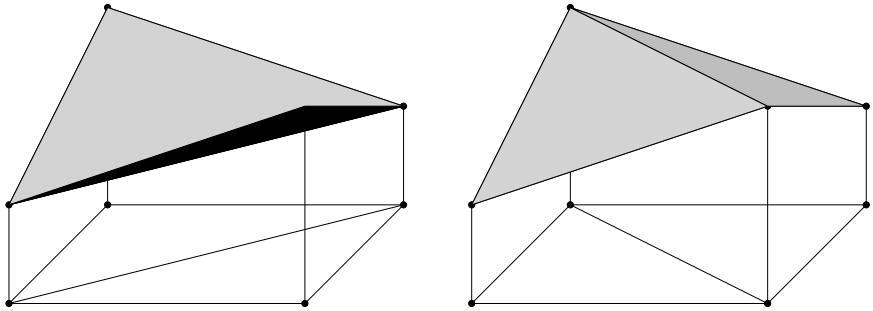


Figure 5: Two different triangulations of the same points. (a) Triangulation creating a valley (b) Triangulation creating a ridge.

There are two major ways of representing terrains in this way, TINs and grids Kumler [1994]. It will suffice to list a limited number of advantages for elevation modeling based on the AHN2 data. A TIN structure keeps the original sample of points, whereas a grid has to be interpolated. The major advantage stated by Burrough and McDonnell [1998] is because a TIN is irregular, the local density of the points in a TIN can adjust to degree of variety in height of the original terrain e.g. flat surfaces use fewer points than surfaces that vary greatly in height.

A TIN is a model of the surface using triangular facets [Peucker et al., 1978] where the height is a function of the coordinates: $H(x, y)$. Since a TIN is a 2D representation, points in a TIN cannot overlap, which means that perfectly vertical walls and overhangs are not represented. Because of this property, storing a third dimension (height) is often called a *2.5D* representation.

2.3 CONSTRUCTION OF TINS

The question is: given a set of sample points, how to triangulate these vertices into an irregular tessellation of triangles? When representing a terrain, some triangulations give better results i.e. approximate the original terrain better. In Figure 5 two different triangulations are visible, which represent very different terrains from the same seven sample points.

With the assumption that for a terrain, points closer together are more similar in height than points farther away [1] (called *spatial autocorrelation*), long and *skinny* triangles are bad representations of terrains. For a good representation of terrains, triangles should be *fat*, or more formally stated, a triangulation that has the smallest collection of small angles in its triangles.

A common approach is the DT, of which it is said that its triangles are as equilateral as possible [Worboys and Duckham, 2004]. Two main properties of the DT are as follows [Worboys and Duckham, 2004]:

For a given set of points P, *of which no sets of three points are collinear and no four points are co-circular:*

1. The DT is unique.
2. The circumcircle of each triangle contains no points of P.

1 Tobler's first law of geography

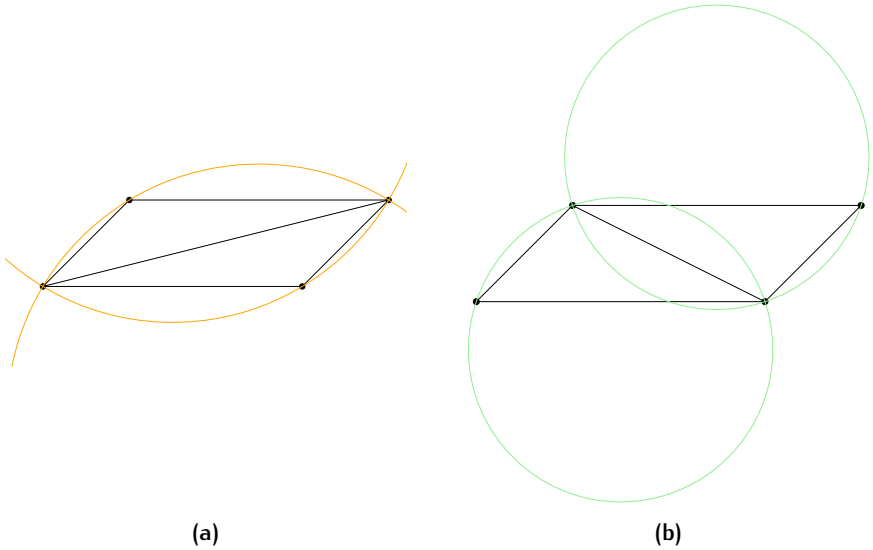


Figure 6: Circumscribed circles on the two triangulations of Figure 5. (a) Circumscribed circles show non DT (b) Circumscribed circles showing an DT.

The second property is seen in Figure 6, which shows the circumcircles on top of the triangles seen in Figure 5. It thus follows that Figure 6b is a DT.

Occasionally, even with a DT the terrain is not adequately approximated due to a bad sample of points, or a coarse resolution. Especially steep vertical drops are hard to represent in a TIN without having sample points directly below and above the slope. Therefore a *constrained DT* can be used, in which some edges are predetermined (as seen in Figure 7c, resulting in a better representation of the predetermined edge, but possibly a worse representation around it, since the triangulation is not a true DT anymore. By adding extra vertices along the predetermined edge, so-called *Steiner points*, a *Conforming Delaunay Triangulation (CDT)* can be created (as seen in Figure 7d, in which both the predetermined edge and the Delaunay property is preserved.

2.4 APPLICATIONS AND OPERATIONS OF TINS

The application of DTMs and Digital Elevation Model (DEM)s is common in many fields, mainly civil engineering, earth sciences, surveying and photogrammetry and resource management [Weibel and Heller, 1993; Li et al., 2010]. In these fields, many different application exists e.g. visualization, watershed modeling [Lyon, 2003], view shed analyses [van Kreveld, 1997]. TINs can even be used in modeling propagation of radio waves for a terrain [Djinevski et al., 2014] among many other applications [Moore et al., 1991]. TINs can be used for visualization as a basis for planning and management Oude Elberink et al. [2013]. It can also be used in civil engineering, for inspecting terrains [Li et al., 2010].

Visualization of terrains is often done by shading the polygons of the tessellation. These can be shaded by aspect, called *hillshading* or by slope

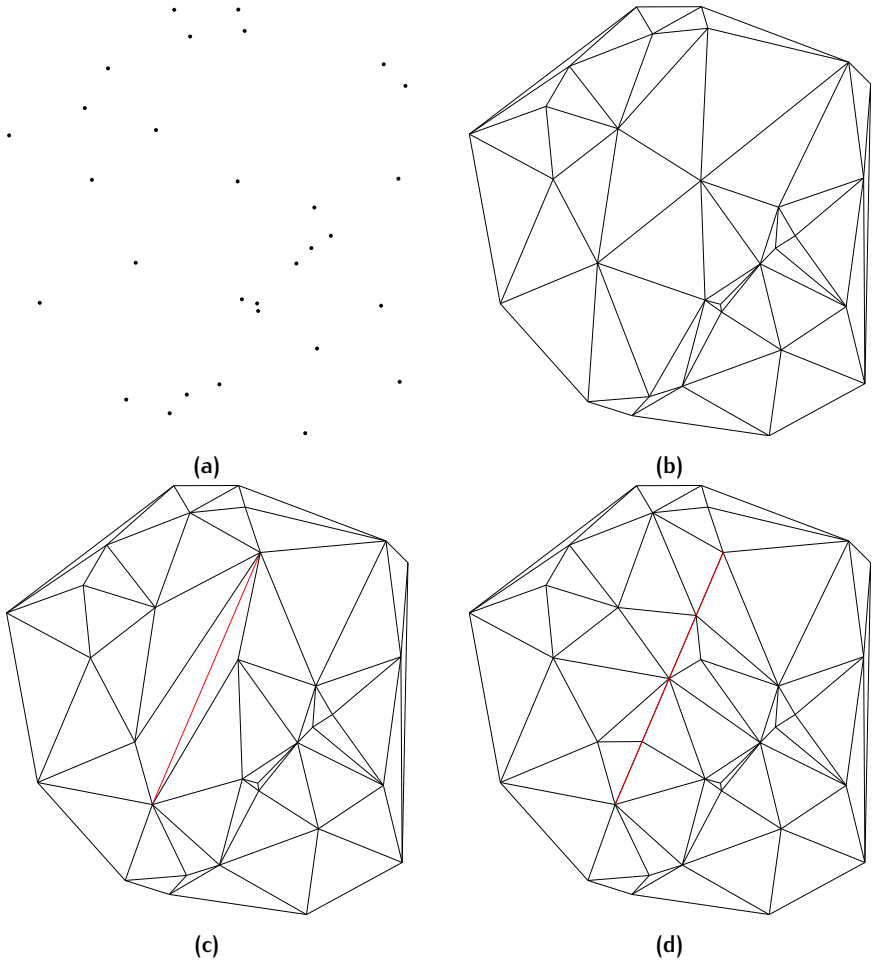


Figure 7: DT (a) A set of points P (b) DT of P. (c) Constrained DT of P (d) Conforming DT by adding extra points.

(or other user defined attributes such as height as in Figure 8a). Slope is the steepness of a plane seen in Figure 8b, while the aspect is the direction in which the slope faces, often visualized by simulating shade as seen in Figure 8c.

The filtered AHN2 dataset provides a DEM and can be used for many purposes, most notably hydrology related applications in the Netherlands. In Figure 9 a watershed model is shown. By knowing the aspect and slope of triangles, for each triangle the direction the water will runoff to can be determined. All triangles from which the water reaches the same local minimum (shown in the Figure 9 as a dot) is called a watershed [Lyon, 2003].

Contouring is the most popular way of visualizing a DTM, an overview for creating contour lines from TINs is given in van Kreveld [1997].

Drainage networks often require DTMs with specific properties e.g. no flat planes (triangles in TINs) [Yu et al., 1996], as well as clear definitions for drainage itself i.e. water follows the steepest descent and the steepest descent is unique [Agarwal et al., 1996]. For such analysis purposes, two types of points are often classified on a surface: *regular* and *singular* points [Peckham and Gyozo, 2007], where regular points have one direction of flow, while singular points have an undefined direction. This, for example, occurs in points that are lower than all the other points around it, a local minimum (pit) or the inverse, a local maximum (peak).

2.5 GRAPHS

2.5.1 Topology

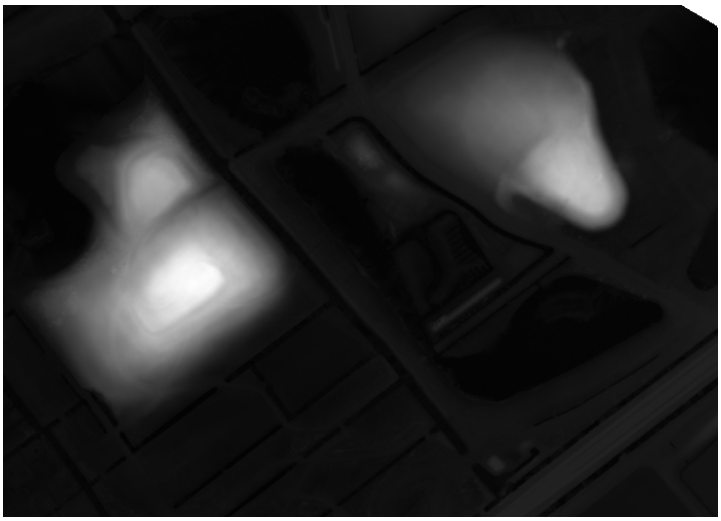
Topology is the study of topological transformations and the properties that are preserved by topological transformations, such as the simple point *in* polygon property. This is contrasted with non topological properties such as the distance between two points, which changes during topological transformations. Thus topology is about relationships between geometries and for the scope of this thesis, about neighbours. Several frameworks describing topological relationships exist Egenhofer and Franzosa [1991], but for TINs the important field is *combinatorial topology* where topology is applied to spatial data modeling [Worboys and Duckham, 2004].

For an overview of topological frameworks, see Zlatanova et al. [2004] and see van Oosterom et al. [2002] for their implementation in DBMS.

The fundamental tool to model geometric shapes and spaces with uses the notion of *simplicial complexes* [Worboys and Duckham, 2004]. A *0-simplex* is a point, a *1-simplex* is a line and a *2-simplex* is a triangle. A simplex is thus the simplest element in a dimension. Higher dimensionality simplexes can be constructed from those simplexes of lower dimensionality i.e. the triangle (2-simplex) can be build from edges (1-simplex) which can again be build from vertices (0-simplex).

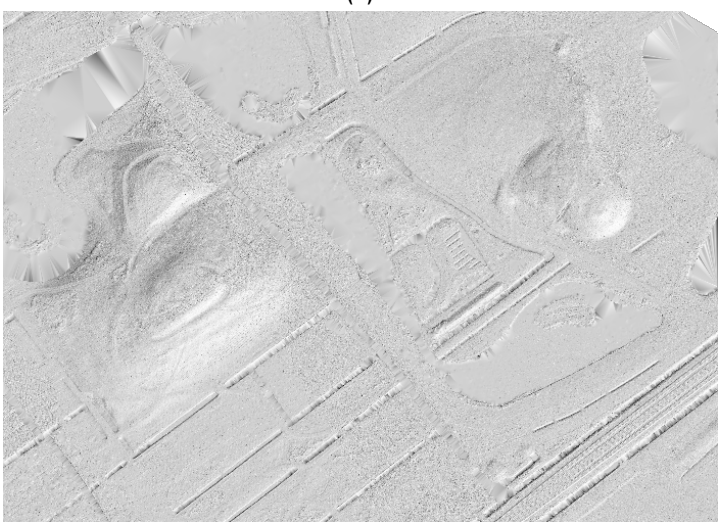
A simplicial complex is a finite set of simplexes with special properties. Each two simplexes are either disjoint or intersects in a simplex that is also part of the simplicial complex [Worboys and Duckham, 2004]. A triangulation such as the DT is such a simplicial complex. When embedded in the plane, it forms a planar graph [Edelsbrunner, 2001] in which the following relationships can be defined between simplicial complexes:

- `incidence` Two simplexes that are a part of each other are incident to each other e.g. a vertex a is incident to an edge ab.
- `adjacency` Two simplexes are adjacent when they share a incident simplex e.g. when two triangles share a common edge.



(a)

(b)



(c)

Figure 8: DTM visualization of `demo.las` (a) by elevation (black to white height) (b) by slope (black to white intensity) (c) by aspect (shadows cast from light source in the south east.)

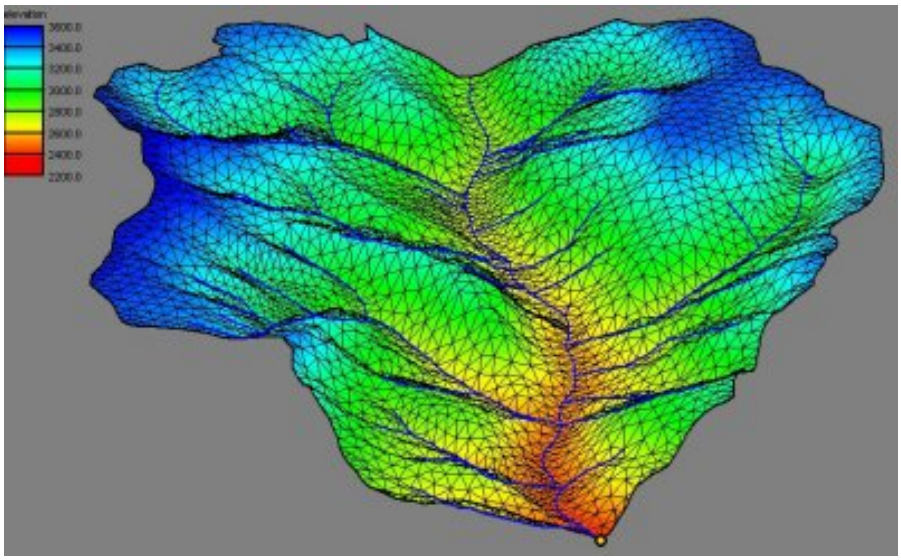


Figure 9: TIN used as a DEM for watershed modeling [XMS 2008].

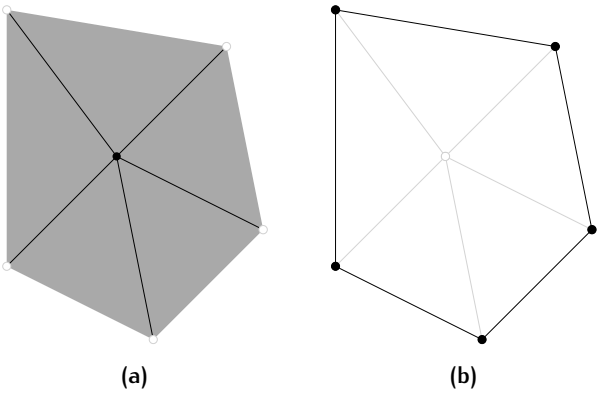


Figure 10: The star and link of a vertex (a) The black edges and gray triangles belong to the *star* of the black vertex (b) The black vertices and edges belong to the *link* of the gray vertex.

Functions discussed in Section 2.4 often require knowledge about the neighbours of a given point or triangle e.g. the local minimum function requires knowledge of the points surrounding it in the triangulation. This can now be described as the number of adjacent vertices, or the number of edges with which the point is incident. This number is called the *degree* of a point (node) [Worboys and Duckham, 2004].

Two more important subsets of simplicial complexes are the *star* (seen in Figure 10a) and the *link* (seen in Figure 10b) used to describe data structures in the following section Section 2.6.1. Formally described, a *star* of a simplex p are all simplexes incident to p, while the *link* contains all simplices in the *star* not incident to p [Edelsbrunner, 2001].

One other property of two-dimensional triangulations is important, which is derived from Euler's formula:
Given a set P *of* n *points in a plane, not all collinear and where* k *is the number of points on the convex hull, for any triangulation of P:*

- the number of triangles is $2n - 2 - k$
- the number of edges is $3n - 3 - k$

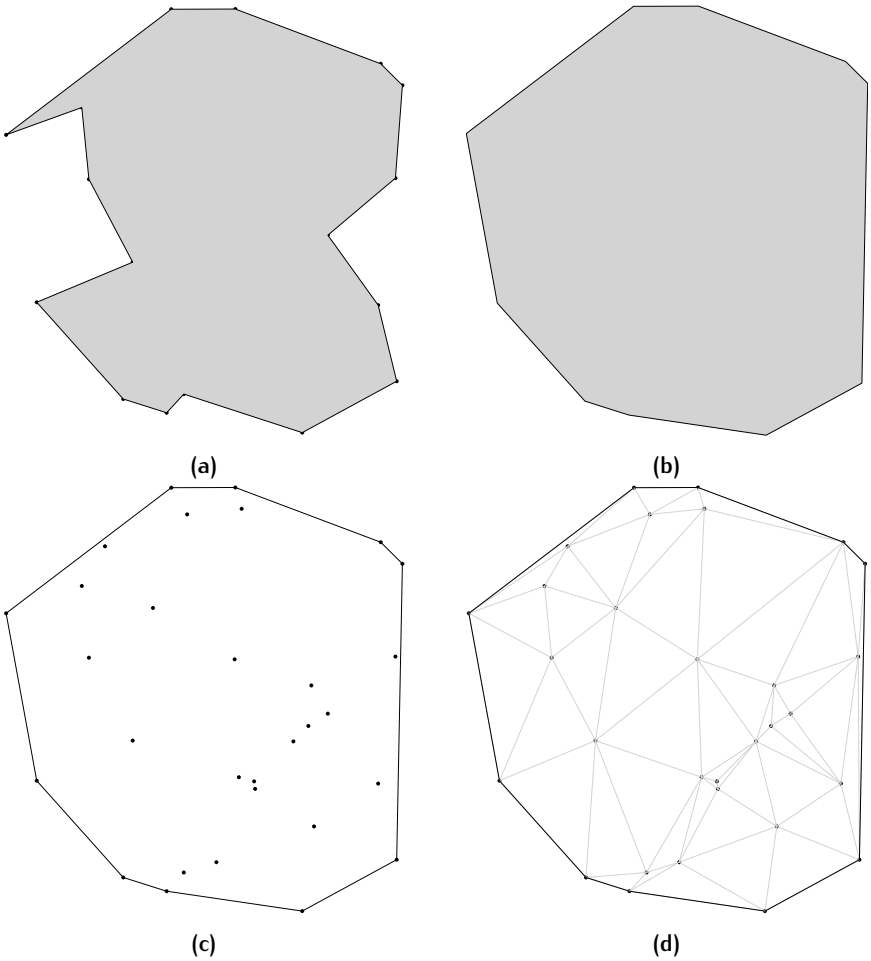


Figure 11: Convexity (a) Non convex polygon (b) Convex polygon. (c) Convex hull (d) Convex hull of a DT, which is the same as (c).

Together with the fact that in a given DT the average degree of each vertex is six [Okabe et al., 2009] it is now possible to estimate the size of TIN data structures in Section 2.6 for TINs computed by a DT.

2.5.2 Convexity

Convexity is a concept in euclidean geometry defined by Worboys and Duckham [2004] as: A set of points S is convex when every point in S is visible from every other points in S. The intersection of all these sets is called the *convex hull*. This concept is seen in Figure 11. A more intuitive approach of this concept is the idea that the convex hull is a string strung around a set of nails representing points. All nails will be either touching the string or will be inside the loop it forms. The use of the convex hull becomes clear when its applied to triangulations such as a TIN, where the boundary of the triangulation of a set of points P (the outer edge) is the same as the convex hull of P.

2.6 DATA STRUCTURE OF TRIANGULATIONS

Data structures for TINs can be node-edge or triangle-based, the only possible simplexes of a two-dimensional triangulation. These simplexes, and combinations of them, can result in very different properties for each data structure. In order to compare the sizes of these data structures, which is relevant for massive datasets, the following sizes are defined:

- Each node is stored as three (x, y and z) doubles (of 8 bytes) requiring 24 bytes.
- Each id and pointer to one is stored as an integer, requiring 4 bytes.

2.6.1 Node based

For a node based structure all the points would be stored explicitly, including possible attributes, which actually resembles a point cloud. A separate structure is needed to describe the triangles, by either referencing to incident edges or triangles. The combination of storing triangles and nodes is discussed in section 2.6.3.

STAR BASED Such an approach is the *star* based data structure described by Blandford et al. [2005] seen in Table 2. The construction of the star is explained in Figure 12, in which each point stores the id of the points in the *link* of p. This list is stored clockwise or counter-clockwise, forming a loop. p thus has [a, b, c, d, e, f, g]. Every combination of two subsequent elements in the array is an edge in the *link* of p. Triangles are formed by combining p with two ordered points from the array. A triangle is thus formed with pab but also with pbc etc. The star circles around, thus the last triangle is pga. Blandford et al. uses *difference* encoding, not storing the absolute reference in the link, but the relatively to the node itself, thereby saving some space.

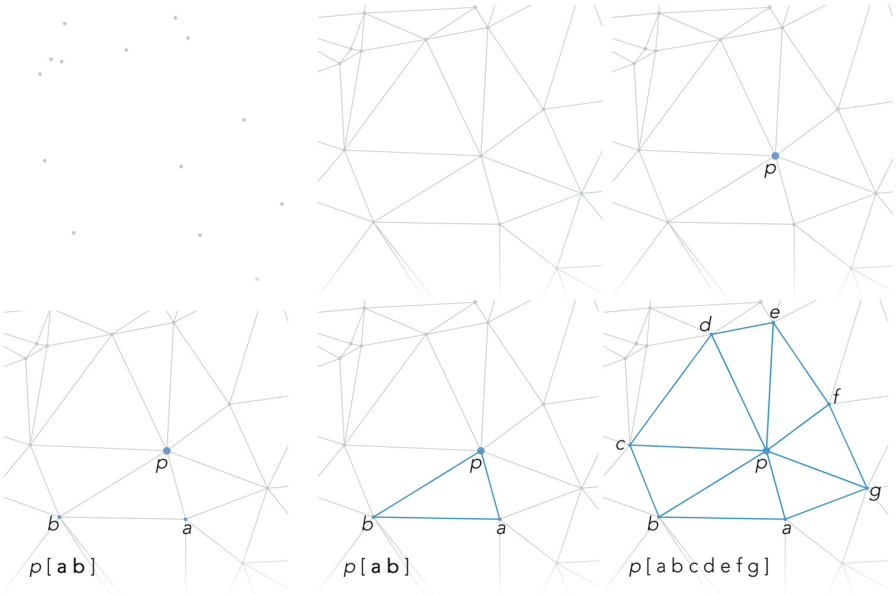


Figure 12: Six stages of storing a star based TIN, starting with the point cloud, ending with the star.

id	x	y	z	link
1	1.0	1.0	1.0	[1,2,3]

Table 2: Star based storage by Blandford et al.. The link is by difference encoding, each item should be added to the node id.

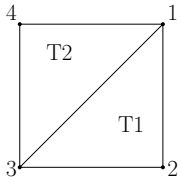


Figure 13: TIN example for Table 3.

A somewhat larger version of this structure is described by Burrough and McDonnell [1998], in which not only the nodes in link are stored, but also the triangles in the link of each node (fig. 10). The degree of each node is also stored. The -3200 in the link is indicative of the convex hull of the triangulation, which prevents the non existing triangle $1, 4, 2$. The same occurs in the triangle list, where the 0 refers to such a non existing triangle.

Node based data structures are inherently quite small compared to structures which store either edges or triangles, since these structures have at least thrice of twice the number of simplexes. The star based structure that only stores the link is the smallest thus far, requiring 1 id, 3 coordinates, and on average six references in the link, which sums to a size of $52n$ bytes, where n is the number of nodes in a TIN.

2.6.2 Edge based

A comparison of edge data structures is found in Kettner [1999]

Many edge based structures exist to represent topological structures. They require the definition of a *directed graph* in which each edge is assigned a direction [Worboys and Duckham, 2004]. The edge ab can now be represented twice, as the directed edge ab and ba. This enables referencing to elements to either the right or left of edges.

WINGED EDGE is an edge based data structure; first proposed by Baumgart [1975] it is shown in Figure 14. It stores eight references for each edge, the two vertices (PVT,NVT) incident to the edge, four incident edges with

id	x	y	z	degree	link	triange list
1	1.0	1.0	1.0	4	[2,3,4,-3200]	[T1,T2,0,0]
2	1.0	0	1.0	3	[-3200,3,1]	[0,T1,0]
3	0	0	1.0	4	[1,2,-3200,4]	[T1,0,0,T2]
4	0	1.0	1.0	3	[-3200,1,3]	[0,T2,0]

Table 3: TIN node data structure of Figure 13 based on Burrough and McDonnell [1998]. The combinations of the node id with the linked simplexes in link form edges, and to the right of each edge is the triangle described in the triangle list.

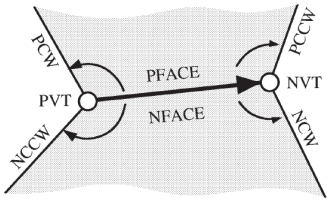


Figure 14: Winged edge data structure: for each edge eight references are stored [Kettner, 1999].

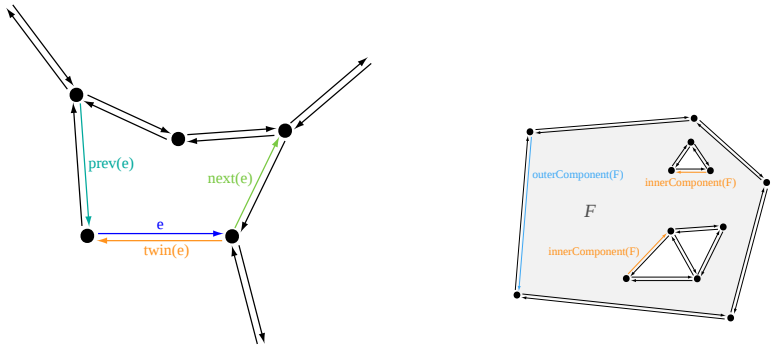


Figure 15: DCEL (a) DCEL information for each edge [Wikimedia, 2015] (b) DCEL information about incident faces. [Wikimedia,2015]

common faces, the so called *wings* (PCW, PCCW, NCW and NCCW) and two faces (PFACE, NFACE).

DCEL [2] is an edge based structure that stores directed edges (shown in Figure 15a, thus twice the amount of edges. Hence it only stores only information on one side of each directed edge, namely the incident face on its side (shown in Figure 15b, as well its opposite half edge, the vertices incident to its edge and two references to the edges adjacent to it that are also incident to the reference face (the NCW and NCCW in the winged edge structure) [Berg de et al., 2000].

The quad edge data structure by Guibas and Stolfi [1985] is similar, but stores four quad-edges for each edge. However it will not be discussed here, because it is inherently larger than other edge based solutions. All the edge based structures are very verbose, by storing many incident and adjacent simplexes, increasing the required storage space, but probably becoming faster as a result.

When calculating the required space we assume common database practices and create a node table, with n rows, containing an id and the coordinates, taking $25n$ bytes. The winged edge is the smallest edge based solution discussed here and requires an id of 4 bytes and 8 references of 4 bytes, with the assumption that faces are only implicitly stored. This would result in a size of $25n + 3 * 36n = 133n$ bytes.

2 This refers to the structure commonly called *half edge*

n_id	x	y	z
1	1.0	1.0	3.0
2	1.0	0.0	3.0
3	0.0	0.0	3.0
4	0.0	1.0	3.0

t_id	a	b	c
1	1	2	3
2	3	4	1

Table 4: Node and triangle table of the triangle array structure.

t_id	a	b	c	I	II	III
1	1	2	3	0	0	2
2	3	4	1	0	0	1

Table 5: The Triangle Array structure of Table 4 extended by adding references to three adjacent triangles. 0 is no triangle (convex hull)

2.6.3 Face based

Face based structures are among the most simple, because they store the highest dimensional simplex (the triangle) and need no further elements. These are also the first structures not explicitly designed for graphs.

SIMPLE FEATURES uses *rings* to store polygons, thus a triangle abc is actually stored as $abca$. It also stores the number of rings and the type. Implementations of simple features are further discussed in Section 3.2.1. The size of such a structure is very large, because the coordinates of each node are stored four times for each triangle. This results in a 96 bytes for every triangle, thus $288n$ bytes only considering storing coordinates. With a unique id for each triangle, the size becomes $300n$ bytes.

A more normalized approach would be to store the nodes in a separate table and to let the triangles only refer to these nodes, instead of storing them for each triangle as seen in Table 4 as described by Li et al. [2010]. This is a very common approach (from now on called the triangle array) and is quite small.

The size of this data structure $28n$ bytes for the node table and $2 * 16n$ for the triangle table, resulting in a data structure of $60n$ bytes. The face based structures described do not explicitly store topological relationships. In order to find the star of a node in this structure all the triangles have to be scanned, a costly operation, whereas in the the half-edge structure, by repeatedly finding the adjacent nodes and opposite half edges, it only needs a few lookups in the data structure.

For many operations described in Section 2.4 a topological data structure is thus highly recommended. In this manner, the triangle array structure previously described can be easily adapted. When for each triangle not only its three nodes are stored, but also the triangles incident to its edges, and thus adjacent to the triangle are stored, the structure has become verbose enough to describe topological relationships (see Table 5). Operations like the degree of a node do not require a complete scan of the table anymore. However, this does increase the size considerably. which now becomes $28n + 2 * 28n = 84n$ bytes.

2.7 STORING ATTRIBUTES

Pointclouds often have extra attributes such as colours and intensities. These attributes can easily be stored in the described data structures since all the data structures, except for the Simple Features, store nodes. The AHN2 dataset however, does not have extra attributes. Attributes for edges are not so common in TINs. Derived products from TINs such as drainage networks can store attributes for the edges, but are not common in TIN data structures itself. Lastly, attributes on triangles are common for cadastral datasets where TINs represent objects, but for a dataset like the AHN2 they are not.

2.8 EFFICIENT ACCESS TO A TIN

As discussed in the scope of the research (Section 1.5), the data structure needs to be efficient, meaning that the computational processes required to perform an operation (called an *algorithm*) such as computation of the slope of a triangle needs to be efficient. The efficiency of an algorithm is called *algorithmic complexity*, which can be further subdivided into *time* and *space* complexity [Worboys and Duckham, 2004] e.g. the time and storage space an algorithms needs to run.

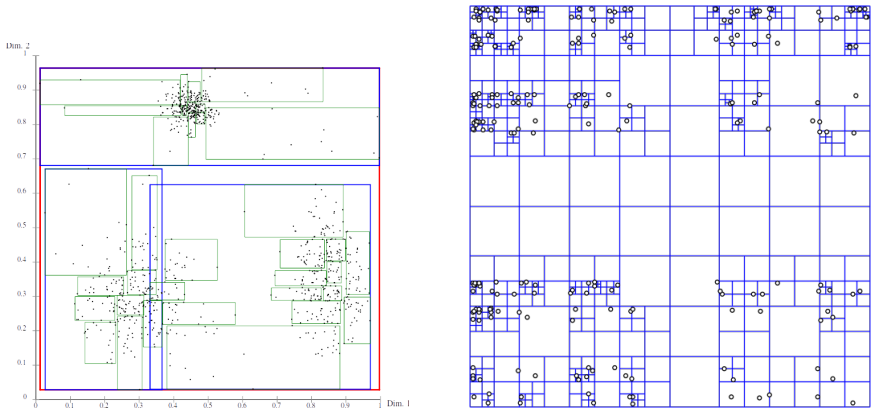
The time an algorithm takes to run is often dependent on the size of the input. This is often denoted in the so called "big-oh" notation i.e. $O(n)$ where n is the size of the input data linearly related to the time it takes for the algorithm to run. It thus expresses the relation between the input size n and the time taken by the algorithm. Sometimes the size of the input does not matter, while at other times, each extra item in the input increases the time exponentially.

This has led to the notations of $O(1)$ (constant time) where the time taken is not related to the input size, $O(\log n)$ (logarithmic time) where the time taken is logarithmically related to the input size and so on in order from fast to slow , $O(1), O(\log n), O(n), O(n \log n), O(n^k), O(k^n)$.

ACCESS METHODS Complexity is also relevant to data access. In a database with n rows, it will take an average of $(n+1)/2$ rows to find the row of interest [Worboys and Duckham, 2004]. In such a case the rows are read sequentially and the complexity is $O(n)$. When the rows are ordered by the attribute of our interest (e.g. height, from low to high) *binary* searches become possible. Instead of iterating sequentially, a binary search starts at the middle row and determines whether our value of interest is higher or lower and repeats this, effectively cutting the number of possible rows in half. This yields a performance of $O(\log(n))$, which is much faster than the sequential scan.

For more complex data structures however, *indexes* are used, much like the index of a book. These auxiliary data structures are ordered lists of attributes with references (called *pointers*) to the unordered sequence of rows, enabling strategies like binary search with the cost of increased storage size.

Until now these examples were one dimensional, the attributes were lists, a long sequence of values which can be ordered. For spatial objects however, queries for locations can be both in x direction as the y direction, requiring



(a) (b)

Figure 16: a) A rtree, bottom up index. [Wikimedia,2015] (b) A quadtree, top down index. [Wikimedia,2015]

two dimensional indexes or k-dimensional for k dimensions when adding more dimensions such as the height.

There is myriad of spatial indexes and variants for each type of geometry discussed extensively in [Samet, 2006].

Two types of spatial indexes can be distinguished: object and space based indexes. The first is most usable for objects , such as multiple polygons, while the latter is better for indexing fields [Samet, 2006]. A well known object based index is an R-Tree (seen in Figure 16a, while the field based index can be a Quadtree (seen in Figure 16b.

An Rtree indexes the *bounding boxes* of objects; the smallest rectangle still encompassing the object. Each bounding box has a pointer to the actual object. Only when spatial queries intersect with the bounding box, the more complex object is retrieved and also tested. A quadtree subdivides a field in four repeatedly, until a criteria is met, such as a number of objects in the region.

Ordering is also possible in two dimensions, which is often done by *space filling curves*. In Figure 17 the Morton curve is visualized, often called the Z-order curve, because on each level the same Z order is used for iteration. The coordinates for each node in the curve can be interleaved, resulting in the *Morton code* (which is how it is constructed). For more details see Section 4.3.2 and Sagan [1994].

Although a TIN is also a collection of triangles and other objects, an index for each single triangle uses a lot of storage space, storing two coordinates (bounding box) for each three coordinates (triangle). This effect only increases when indexing lower dimensional objects, such as lines and points.

2.9 VALIDITY AND INTEGRITY

One of the advantages of using DBMS is data integrity managed by the database [Elmasri and Navathe, 2006]. In order to keep the same integrity om a geometric level in the database, it is important that there are tests to validate geometry, based on the data structure defined for it. Operations on invalid geometry will otherwise yield incorrect results without the user being aware of it and data transfer of such geometry will lead to data loss [van Oosterom et al., 2005].

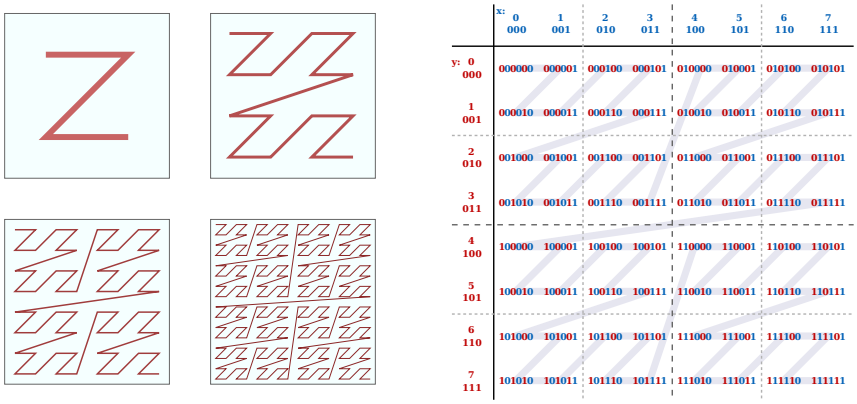


Figure 17: a) The Z-order space filling curve. Notice how all points are traversed by one curve. [Wikimedia,2015] (b) If the coordinates of a node in the Z-order curve are interleaved, we get a Morton code, an unique key. Coordinate 0,0 becomes 0, while coordinate 4,4 becomes 15, for a total of 16 items. [Wikimedia,2015]

If the link in the star based structure is stored clockwise instead of counter-clockwise as expected by the algorithm, while all the parts of data structure are present, queries returning the triangle to the right of an edge will actually give the triangle to the left of the edge.

Thus, the implemented prototype in this thesis should have a clear definition of its data structure and should provide several validation functions, capable of both validating the required elements of the data structure and the more complex cases such as the clockwise ordering of lists if applicable.

2.10 CRITERIA FOR TIN STRUCTURES

Some criteria that can be used to judge TIN implementations, both theoretically and practically, can now be established, providing an hypothetically efficient approach. An efficient approach:

- Stores nodes explicitly
- Stores topological relationships explicitly
- Does not index each individual object
- Provides several atomic TIN functions e.g. degree, slope, aspect, height, local minimum and maximum

For specific TIN operations and analysis, using topological relationships is a must in order to quickly find neighbouring simplexes. Spatial indexes should not be used to index every simplex (be it node, edge or triangle) in a TIN. The same operations require access to specific nodes, giving data structures storing nodes an advantage. Finally we can say that using the star based approach seems to be the smallest structure in theory and the simple features approach the least small, as seen in Table 6. The star based structure also stores topological relationships explicitly, while the simple features approach does not. Table 6 shows that only the star based approach

data structure	star	winged edge	SF	triangle array	triangle array+
size	$52n$	$136n$	$300n$	$60n$	$84n$
nodes stored	•	•		•	•
topological relationships	•	•			•

Table 6: Size of different theoretical data structures for storing TINs. n is the number of points in the TIN, size is in bytes. Triangle array+ is the triangle array that also stores topological relationships.

and the triangle array+ satisfy these three criteria, making them the ideal candidates for an efficient approach for storing massive TINs.

3 RELATED WORK

This chapter presents the related work on handling massive TINs, the problem they present and methods of constructing them. The existing database solutions to store TINs are described and theoretically compared by the criteria found in the previous chapter. A summary of existing approaches is presented at the end of the chapter.

3.1 CONSTRUCTION OF MASSIVE TINS

In order to to discuss the construction of massive TINs, the term *massive* needs to be defined, as well as the problems that occur when handling massive datasets and solutions to these problems.

Analysis and processing of datasets takes place in the main memory of the computer, which is therefore the deciding factor in the maximum size of the dataset that can be processed. With datasets that exceed the actual amount of RAM[1], the computer starts *swapping* data from disk to memory and back. This process takes up large amounts of CPU time, effectively bringing the actual process to a standstill. To prevent this so called *trashing* one can increase the RAM size. This, however, does not solve the problem. There are two main methods to address this issue: either decreasing the dataset size or using external memory algorithms.

The issue here is more complex than the simplification presented here. For a more detailed explanation (virtual memory) see Vitter [2001].

These are problems for both the construction of massive TINs from massive point dataset and in loading the data into the database itself, as well as requesting the complete data from the database. Thus the problem occurs on all levels of the usage of Massive TINs in DBMS and should be solved on all levels.

3.1.1 Reducing size

SPLITTING DATASETS. The simplest way of reducing the size of the dataset is by cutting it into smaller pieces, so as to create subdivisions. This is what happens when loading massive point data, the AHN2 is available as a tiled dataset for the whole of the Netherlands. Even these tiles are too big for practical use (see Table 21 in Chapter 5) and should be tiled into smaller pieces. However this assumes a dataset that can be cut in two. When working with a TIN, the dataset is connected i.e. there will be triangles at the cut (see Figure 18), which either results in overlapping triangles, gaps or requires complex construction with specific rules to address this issue.

Some loaders try to split the data automatically, such as the PDAL[2] loader which uses an adaptive kd-tree to divide the data in order for each subdivision or bucket to have the same number of points. However, this requires the data to be sorted, which results again in trashing. Loaders that use field

1 Random Access Memory
2 Point Data Abstraction Library see http://www.pdal.io/stages/writers.pgpointcloud.html

Figure 18: Buckets splitting a point cloud and a TIN (a) A point cloud can be subdivided using buckets. (b) Splitting a TIN in the same way presents problems, what happens to the triangles on the intersection of two buckets?

based indexes such as a quadtree, can establish in which region a point falls without having to wait for all the points.

The split dataset problem (Figure 18) can be overcome by merging one or multiple triangulations e.g. those created by splitting a massive pointcloud using a quadtree. Such a method is presented by Chen et al. [2006], using a parallel divide and conquer algorithm, which stitches split regions back together.

THINNING DATASETS. Another method is to thin the dataset, by only using a selection of the available data or points in the case of a TIN. Thinning is often implemented by a random selection, which can discard significant points of a TIN. With the assumption that all points are significant, this is not a viable solution at all. Non random methods for thinning datasets require an analysis on the complete dataset in order to reconstruction the terrain, which is again the problem it is trying to solve. Thinning a dataset is closely related to the *simplification* algorithms, which reduces the dataset, but keeps relevant points. Significant points can be found by a simple moving window and an arbitrary criteria [Heckbert and Garland, 1997] never needing to load a complete dataset. From this selection a DT can then be created. Yet this assumes that dataset itself is ordered, otherwise the moving window will never find all the points. Ordering is again an operation that requires to load a complete dataset.

An extensive overview of surface simplification algorithms is found in Heckbert and Garland [1997].

3.1.2 External memory algorithms

As mentioned before, reducing size or increasing the RAM available does not solve the inherent problem -the uncontrolled swapping of data between memory and disk. Algorithms that specifically control the flow of data (swapping) from disk (external memory) to the RAM (internal memory) are called external memory algorithms. These algorithms exploit *locality*, the fact that some data is used repeatedly by a program for a certain time and then moves on to other data [Vitter, 2001]. These programs decide when and where to swap data by exploiting knowledge about locality.

These algorithms exist for creating massive TINs or DEMs, most notably by Agarwal et al. [2006], which divides the construction of the DEM in three distinct phases: a segmentation phase, a neighbour finding phase and the final interpolation phase. The first phase creates a quadtree for all the points, writing data to disk when memory becomes full. The second phase finds all the neighbours for a given cell in the quadtree, which when combined can be processed one at a time by the interpolation phase. The key here is the incremental method, the quadtree can be created incrementally and the other phases run only on small parts of the data at a time. Similar external memory algorithms can be applied for the creation of massive TINs [Agarwal et al., 2005] or other analyses and operations on massive datasets [Vitter, 2001].

STREAMING OF DATA. Streaming algorithms are akin to external memory algorithms but do no swap at all in the meantime as they only use the disk (external memory) for input and output [Isenburg et al., 2006a]. Instead of keeping all the data in the working memory or swapping parts to external memory, only a part of the dataset is read from disk and processed at a time and when that part is no longer needed it is removed from the memory and written to disk. This requires knowledge that this part will never be needed again, which is implemented in streaming of geometries by Isenburg et al. [2006a] as *finalization* tags on topological objects. If an object is tagged, it will not be referenced anymore by the remaining program and it can be written to disk.

This knowledge is possible because of a scan of the complete dataset, collecting statistics about small pieces of the dataset, not unlike the quadtree phase of Agarwal et al. [2006] and combining this with the concept of locality. Geometric datasets express locality which can be expressed as *spatial coherence* i.e. vertices close together in the stream are also close in space. Streaming algorithms aims to document this property [Isenburg et al., 2006b] because, for local operations such as interpolation and triangulation, the higher the spatial coherence of a dataset, the faster is a point finalized, thus keeping the memory footprint small. These algorithms[3] can also be used to calculate grids [Isenburg et al., 2006a].

Isenburg et al. [2006b] also demonstrate that the spatial coherence of LIDAR datasets (such as the AHN) is high and are thus fit for streaming algorithms.

3.2 EXISTING TIN STRUCTURES IN DBMS

Some of the theoretical structures of a TIN seen in 2.6 have been implemented in spatial databases. Only two TIN datatypes are known to have been implemented by common spatial database providers. One is the SDO_TIN type by Oracle, the other the OGC Simple Features specification in use by several databases.

Another way to store TINs is either by using more general geometry solutions, such as the storage of multiple Polygons, or by using topology solutions available in some spatial databases.

3 blast extension at http://www.cs.unc.edu/~isenburg/lastools/

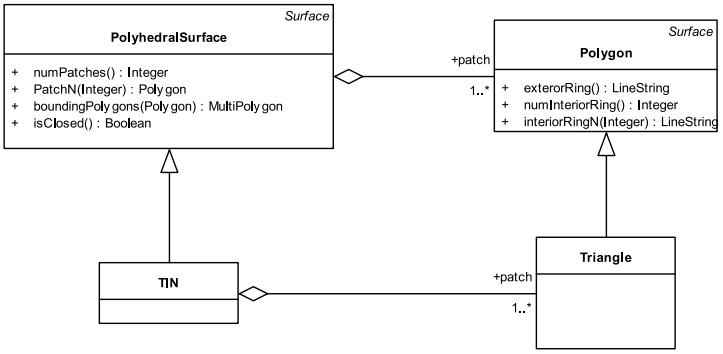


Figure 19: OGC PolyhedralSurface

	endianness	type	#triangles	wkb trianglez * num
type	bit	integer	integer	Table 8
value	1	1016	2	Table 8

Table 7: WKB TINZ structure.

3.2.1 OGC TIN

A face based approach is Simple Feature (Access) (SF) a standard (by OGC and ISO[4]) [OGC, 2006] to store geographical data. It has become well known since it was integrated in many of the spatial databases. Part 2 of the standard describes SQL implementations of these Simple Features. Several drafts and iterations of these standards exist and not all databases have implemented the latest version.

Since 2009 the draft specification (v1.2.1), based partly on the ISO SQL/MM draft, contains a specific Triangle and TIN type, which are subclasses of the Polygon and PolyhedralSurface respectively as seen in Figure 19.

A Triangle is defined as surface with one exterior ring with three different points, with no points being colinear and no interior rings (holes). It can have multiple dimensions: from 2 up to 4. A TIN is a patch of Triangles with the same orientation. A TIN can be defined as a collection of triangles that are connected by their edges. Types with multiple dimensions have different names and here TINZ and TriangleZ are used to denote that each coordinate also has an height attribute.

Although SF is implemented by most database providers e.g. PostgreSQL, IBM, Ingres, MonetDB, the latest version with support for 3D and TINs is only implemented by PostgreSQL PostGIS (v2) and Ingres (v10.2).

When using SF the geometry should be stored with a unique id, its bounding box and the geometry represented in Well Known Binary (WKB) format. PostGIS uses its own storage format [5] as seen in Figure 20 but it is very similar to the WKB format, which will be described for a TIN.

Table 7 describes the WKB format for the OGC SF TIN type, containing multiple TriangleZ data structures, as shown in Table 8. The Well Known Text (WKT) for the represented TIN is:

```
TINZ (((0 0 0, 0 1 1, 1 0 1, 0 0 0)), ((0 1 1, 1 1 2, 1 0 1, 0 1 1))).
```

Although not stated specifically, the TIN as a patch of triangles is a bucket in its own right, as well as all Multi geometries in the OGC SF. However,

4 ISO 19125
5 https://trac.osgeo.org/postgis/browser/trunk/postgis/README

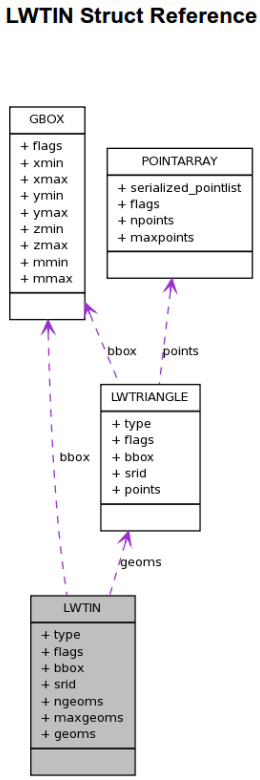


Figure 20: PostGIS TIN structure [PostGIS 2015]

	endianness	type	#rings	#points	coordinates
type	bit	integer	integer	integer	4 * num. points * double
value	1	1017	1	4	0 0 0 0 1 1 1 0 1 0 0 0 etc.

Table 8: WKB TriangleZ structure.

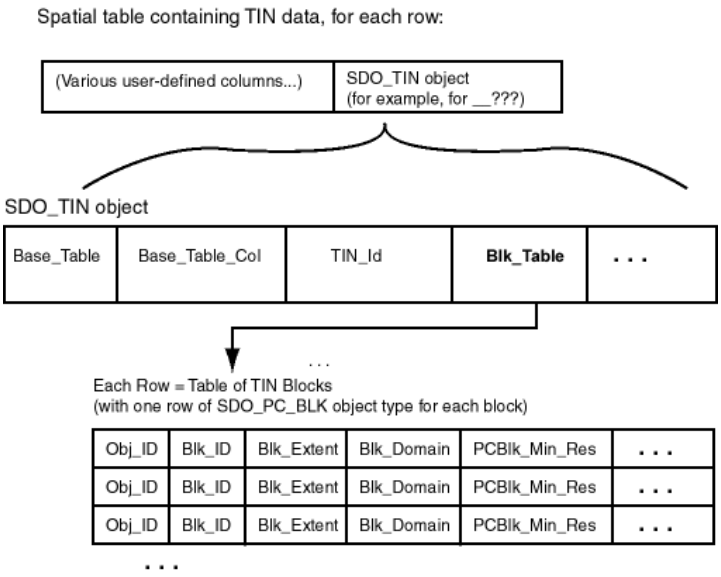


Figure 21: Oracle storage model of a TIN [Oracle, 2015]

these buckets do not come from subdividing a field as a regular tesselation. Instead, they are created bottom up by the geometries themselves. They are indexed by a GiST using an Rtree.

The OGC specifies many functions to be used, but none specific for TINs. TIN operations, such as slope and aspect, only exist in the raster implementation of PostGIS. PostGIS does implement the specific ST_DelaunayTriangles to generate a TIN from point geometry. Topological relationships are not stored, but PostGIS does have a seperate extension in PostgreSQL called `postgis_topology`. Nonetheless intersections on TINZs and other 3d geometries require a different backend than the normal used 2d GEOS library, requiring the CGAL backend which can be exposed via SFCGAL[6][7]. Although a TINZ can be forced in 2D form by `ST_FORCE2D` the type is not recognized and usable with the standard 2D functions.

3.2.2 Oracle SDO_TIN

Oracle uses a node and face based approach. It stores the nodes and a triangle table. The triangles are three references pointing back to the node table. This structure is the same as explained in Chapter 2. Oracle stores the nodes as the SDO_PC type, while the triangles are simply stored as several ids to the nodes [Oracle, 2015]. Hence, it is very similar to existing pointcloud solutions; only the triangle column is added.

Oracle Spatial uses three objects to store a TIN: a `SDO_TIN` object, which stores the metadata about the TIN, such as its id; the total extent of the dataset; and the table storing the actual data itself. Such a table is called a TIN block table, which represents *blocks* of the TIN. The TIN is thus subdivided into these blocks, which are now referred to as buckets. This enables a maximum 4^{18} points to be stored, which is several orders of magnitude larger than what fits in main memory [Ravada et al., 2009]. Each row in this table repesents a bucket. An overview can be seen in Figure 21.

6 Which defaults to first checking the validity of the geometry before other operations, making comparisons difficult

7 Obtained by recompiling PostGIS or using latest 2.2 experimental binaries

	d1	d2	d3	block_id	point_id
type	float	float	float	integer	integer
value	45112.0	1245.0	1.05	1	1

Table 9: Point in array of points in the oracle points blob.

	v1 b_id	v1 p_id	v2 b_id	v2 p_id	v3 b_id	v3 p_id
type	integer	integer	integer	integer	integer	integer
value	1	1	1	2	2	1

Table 10: Triangle array used in the Oracle data structure. Array is a blob, b_id stands for bucket id, p_id for point id.

A row in the block table has multiple fields which store the metadata about the bucket. The number of points and triangles are stored, the spatial extent, a unique id for identifying the block, the total number of dimensions, the resolution and two objects: A `points` Binary Large Object (blob) and a `triangle` blob.

The points blob is an array of points, storing the coordinates as doubles, together with the block_id and a point_id for every point as seen in Table 9. Although three dimensions (d1, d2 and d3) are displayed, the number itself can be defined in each bucket. It is possible for the points to be sorted, which can be deduced from available fields in the database object, such as `num_unsorted_points, pt_sort_dim and tr_sort_dim`, but this feature is not documented.

The triangle blob is an array of triangles, containing three vertices for each triangle. Each vertex is a pair of the block_id and point_id, thus referring back a point specified by the point_id in the points array of the block specified by the bucket_id. This structure is seen in Table 10. Take note on how each vertex of the triangle refers to a bucket and point in that bucket. The problem of splitting a TIN by buckets as seen in Chapter 1 is solved here by explicitly storing the reference to each bucket.

The `SDO_TIN` type is an expansion of the `SDO_PC` type or point cloud type, which has almost the exact same structure, but is missing the triangles blob. Only two operations on the TIN are available: the `SDO_TIN_PKG.TO_GEOMETRY` operation and the `SDO_TIN_PKG.CLIP_TIN` operation. The last operation clips a TIN and returns a clipped `SDO_TIN` object, based on either clipping geometry and / or resolution. Resolution filtering is based on a attribute stored on creation, no multiresolution TIN is stored. An optional filter is available to only clip certain buckets. The other operation casts the TIN to geometry normally used in Oracle. This geometry contains all the points and triangles stored in the TIN given as input. Only now can other spatial operations be applied on the TIN. It is not possible to do an intersection or other spatial operation directly on the TIN, it has to be decoded first to normal geometry.

The Oracle SDO_TIN type does not store topological relationships for triangles or points, although each bucket forms a complete and connected patch. Updating the data structure locally is not possible.

	edge_id	start_node	end_node
type	integer	integer	integer
value	1	1	2

abs_next_left_edge	next_right_edge	abs_next_right_edge	
integer	integer	integer	
2	3	3	

next_left_edge	right_face	left_face	geom
integer	integer	integer	geometry
2	1	2	LINESTRING

Table 11: PostGIS Topology edge table.

	face_id	geom
type	integer	geometry
value	1	POLYGON

Table 12: PostGIS Topology face table.

3.2.3 Postgis Topology

Since version 2.0 PostGIS has officially[8] implemented topology types that contain topological objects such as nodes, edges and faces. Although this is not a specific TIN type, it does store topology based geometry. A new object `TopoGeometry` is introduced, which stores references to a specific defined topology with an Spatial Reference IDentifier (srid), the layer id, the id in the layer and the type of geometry.

This information is stored in a topology schema, in which layer is stored, referring to the topology table, which refers to a unique schema for each layer. This is the metadata structure for PostGIS topology. The actual topology is distributed over four tables: An edge_data (Table 11), face(Table 12), node (Table 13) and relation table.

PostGIS uses the winged edge (see Section 2.6.2) structure to store topology. The fourth relation table relates the topology to defined layers, but is not of importance for this structure.

Although a main advantage of topology should be a decrease in storage size (as intended for PostGIS Santilli [2011]), because no overlapping edges and points are stored multiple times, this is oddly enough not visible in the PostGIS data structure. Each table contains a normal geometry type, meaning that the edge and faces are not (only) stored as references to nodes. Each

8 Topology existed in prior version but was not documented.

	node_id	containing_face	geom
type	integer	integer	geometry
value	1	1	POINT

Table 13: PostGIS Topology node table.

face contains a Polygon geometry and each edge a LineString geometry, storing many duplicate node coordinates. These geometries are indexed as well, creating more overhead. Each topological object is indexed by an Rtree and most ids are indexed by a binary tree, which leads to very large indexes.

Most functions are prescribed by the ISO SQL/MM standard, but PostGIS has many extra functions for accessing and creating topology objects. However, to spatially query the topology, it has to be converted back to geometry. This topological structure is often used for networks such as roads and rivers. Indeed the TIGER dataset[9] has a special loader in PostGIS. The structure is not primarily used for storing geometry, more often used as a in between step for simplifying geometry by simplifying the edges.

Other databases such as Oracle also support topology, either by implementing it manually in version 9i [Quak et al., 2003], while from version 10g a topology model not unlike the winged edge structure has been implemented Penninga [2004].

3.2.4 Academic implementations

IMPLICIT TIN As discussed in Chapter 2 for a given finite point set with some constraints the DT is unique. Thus only the point set can be used to model a TIN. This is the approach taken by Jones et al. [1994], which stores only a point set, together with optional constraining edges. A quadtree on both the points and the edges is used to efficiently find the right points and edges needed for an on the fly CDT.

This approach is very small and also fast on small queries. However, the larger the query window is, the more points that need to be triangulated, the longer such a query will take. The assumption of the model is that a complete detailed TIN is a (too) big load on memory and therefore only small portions should be triangulated at a time. However, Jones et al. [1994] states: "How long it is worth retaining [TIN] ... depends on the time taken to create it." and examples for big datasets that are too large to store are described as a few million Kidner et al. [2000]. This big dataset as a maximum (be it in 2000) is a not even a minimum for parts of the massive datasets currently used. Indeed, the time it takes to triangulate a massive dataset is one of the main reasons to store a TIN in a DBMS.

The Implict TIN can also be used in multiscale terrains. These multiscale of different level of detail terrains are often used for visualization, but can speed up operations where a detailed triangulation is not needed, or when it is used as a first pass over a dataset [Kidner et al., 2000]. By using a quadtree with points stored on each level, a multiscale terrain can be generated by incrementally generating a new DT for every level.

PGTIN The pgTIN extension for postgreSQL is a prototype by Ledoux [2013], for storing a TIN with a star based structure in postgreSQL, based on the approach by Blandford et al. [2005] and seen in 2.6. A 3D version have been implemented as well by Ledoux and Meijers [2013]. It only uses one nodes table, which means the database storage can also be very similar to existing point cloud solutions and will be compact. Nevertheless, since each point will be indexed, although only with a binary tree, the index will be large.

The structure in the database contains a basic id, the point coordinates and an array of references to vertices in the star of the node as seen in Table 14. A

9 https://www.census.gov/geo/maps-data/data/tiger.html

id	x	y	z	star[]	constraints[]
1	23.0	63.2	1.2	[0,2,3]	[False,False,True]

Table 14: pgTIN row structure in PostgreSQL.

separate table is used with references to points, which are regularly placed on the dataset, creating *virtual* buckets, each which stores one pointer to a node.

Speed is gained by the ability to walk through a dataset (see Figure 22 and Section 3.4). The jump and walk algorithm by Mücke et al. [1999] has been implemented, which uses the virtual buckets. Triangles have to be calculated on the fly, based on the topological relationships explicitly stored. Compared to the OGC Polygon, the structure is much smaller and as fast in basic queries such as point location (Section 3.4.1). However, comparisons to the OGC TIN type or the MultiPolygon type have not been done. I perform these comparisons in Section 5.5 and Section 5.7.

The star structure lacks attributes for triangles, but these can be implemented in an extra table. It can store information about edges e.g. constraints in a CDT by creating another link array as seen in Table 14.

3.3 SUBDIVISION USING BUCKETS

In order to keep massive datasets usable, points are partitioned into non-overlapping regions, also called tiling as seen in the beginning of this chapter. The same can be done for databases, which stores multiple objects not in rows but in a region, enabling a smaller number of rows. For massive dataset the `blocks` storage model is more efficient than storing them in a `flat` table van Oosterom et al. [2015]. This also solves the memory problems that massive datasets represent by implementing tiling at the database level.

Both Oracle and the point cloud extension in postgreSQL use such a method to handle point cloud datasets. If the dataset is split into multiple patches or buckets[10] problems occur with the connectedness of the TIN seen in Figure 18b. Point clouds can be easily subdivided this way, see Figure 18a, but in a TIN all these nodes are connected. When a triangle consists of nodes between different patches, the triangle is broken, unless pointers to the correct patch are also stored. This is actually what Oracle does in the SDO_TIN type, which not only refers to nodes in the nodes table, but also the correct patch in which the nodes are stored. For other implementations of the TIN structure this is a problem that has yet to be solved.

3.3.1 Pairing functions

When using references to other buckets we can store these explicitly as separate fields. To do this, however, means that all points are pairs, while only the triangles on the borders of the buckets use those pairs. For a given n points in a gridlike pattern, roughly $4 * \sqrt{n} - 4$ are on the convex hull and need specific bucket ids. When using relative ids there is an even clearer waste of space as seen in Table 15. Enter pairing functions which encode two natural numbers into one unique natural number, reversibly mapping

10 Terms are interchangeably used by Oracle and other database extensions.

	patch	point	patch	point	patch	point
absolute	154	1	154	2	155	1
relative	0	1	0	2	1	1
paired		2323		2324		3224

Table 15: Triangle nodes in bucket 154 of the multistar. Storing patch ids is overhead, with pairing this is prevented.

	patch	point	paired	unpaired
number	1	2	6	(1,2)
binary	0001	0010	0110	0001,0010

Table 16: Pairing using binary shifts of 4bit numbers.

$\mathbb{Z}_{\neq 0} \times \mathbb{Z}_{\neq 0} \text{onto} \mathbb{Z}_{\neq 0}$. This can be used to reversibly fold a vector into one number i.e. storing the bucketid and pointid as one unique number, which can be unpaired later into the bucketid and pointid.

Pairing functions are closely related to space filling curves, which continuously map an interval $[0, 1]$ in the space $[0, 1]^2$ Sagan [1994]. The morton key shown in Figure 17 is such a pairing function closely connected to the space filling curve, which interleaves the bits of the two natural numbers. Variants of interleaving the binary representation have been described by Pigeon [2001]. Other pairing functions exist, yet these are much more costly in pairing and unpairing and often require multiple exponential operations.

Based on the bit interleaving, a more simple one would be adding two binary representations together by bit shifting the first value by the length of the second e.g. placing them next to each other, doubling the storage as seen in Table 16. This is by no means a space filling curve anymore, but it is a very simple pairing function that is very fast to pair and to unpair.

A variation on the interleaved Morton key is the *spatial location code* by van Oosterom and Vijlbrief [1996], which not only stores the location of the id, but also stores the size of its bounding box, creating not only a spatial index for points, but for objects as well.

3.4 EFFICIENT ACCESS TO TINS

Many indexes exist for efficient access to the elements of a TIN. However, with massive point clouds, most tree access structures need disk access to follow pointers stored in for example a quadtree. A solution is to use bucket methods [Samet, 2006], which either aggregates data in the underlying space (bottom up) or decompositions the underlying space. This is thus another argument to use buckets.

3.4.1 Using topology

Point location is the problem in which region of a subdivision a given point is contained [van Kreveld, 1997] e.g. the ability to find a point in a TIN. If the TIN is created by a DT and the spatial extent is thus convex, a point can be found by *walking* through a TIN as seen in Figure 22 based on orientation tests with stored topological relationships. With random starting

A interactive demo of point location strategies is available here by de Castro, C and Devillers O. [2009]

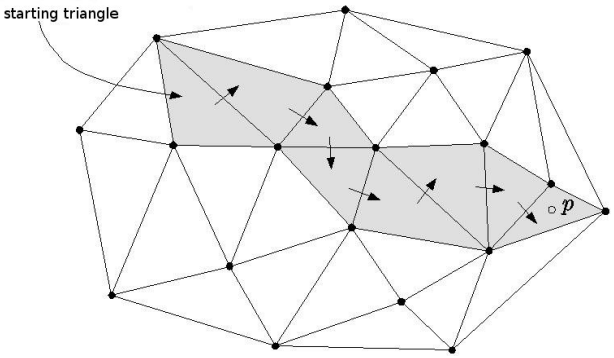


Figure 22: Walking in a TIN by orientation tests [by Ledoux H].

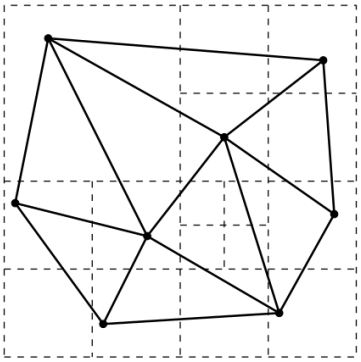


Figure 23: The PM2T quadtree. Notice the smallest subdivision. [De Floriani et al., 2008].

points in a Delaunay Triangulation a considerable speed can be achieved by *jumping and walking* [Mücke et al., 1999]. Many other methods exist for finding starting points and walking by a line Devroye et al. [2004]. A summary is given by Soukal et al. [2012].

3.4.2 Auxiliary spatial indexes

When not using topological relationships, an spatial index is needed on the points, edges or triangles of the TIN. An index especially for triangle meshes is the PM2T quadtree by De Floriani et al. [2008] as seen in Figure 23. It enables effective point location - for queries - instead of randomly choosing a starting point and then walking. The index does not use buckets and may grow very large on a a massive dataset. The maximum number of vertices used was 500.000, still far removed from massive datasets.

Another method similar to using Morton keys is the GeoHash tree by Sabo et al. [2014], which can be used in the point cloud extension in PostgreSQL.

There can be hybrid indexes, using the combination of two indexes: one for buckets and the second for the points in those buckets e.g. a combination of a quadtree with a local Rtree in 3d demonstrated by Yang and Huang [2014]. However, in the previous chapter it has been established that using spatial indexes on individual elements of a massive TIN will result in very large indexes, unfit for storing massive TINs.

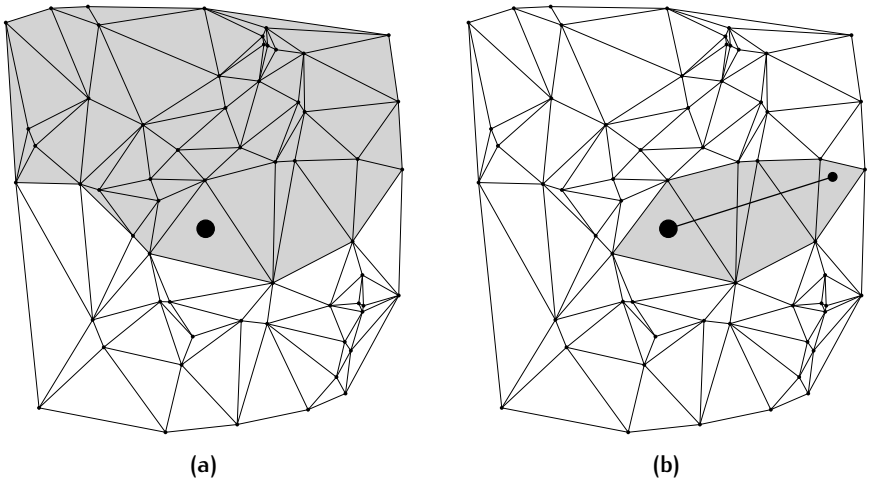


Figure 24: Brute force versus walking (a) A TIN is iterated over until the right triangle is found, resulting on average on iterating over half the dataset ($O(n)$) (b) Walking from a random point to the right triangle, requiring considerably less iterating since the complexity is $O(\sqrt{n})$ [Mücke et al., 1999]

Thus the alternative is using brute force, iterating through a complete bucket or subset until the specified criteria is met as seen in Figure 24a. Walking in a TIN is much more time efficient as seen in Figure 24b.

3.4.3 Compression

In order to make the data structures for TINs more compact compression can be used. Compression can be applied in the data structure itself, in the represented entities in the data structure and in the binary storage on disk. Compression can be either lossy or lossless i.e. with loss or without loss of the original data on decompressing. The pairing functions described in Section 4.3.2 can be seen as lossless compression of two numbers. Here only the data structure and entities are considered, since these can be influenced on loading the data, while the storage itself is handled by the database.

Taubin and Rossignac [1998] provides an overview of compression of triangular meshes using topology. The implicit TIN is another way of compression by [Kim et al., 1999].

Examples in the previous section Section 3.4.2 such as the GeoHashTree are in many ways also compression techniques. The Morton key can be used as a compression technique by pairing the x and y coordinate, storing one value for two (or even three) coordinates.

3.5 SUMMARY OF RELATED WORK

Generating massive TINs present a problem which is most elegantly solved by using *streaming* algorithms. Making use of a quadtree, the problem is solved by regular subdivision. This can also be used in database, as seen by Oracle. Access to a massive dataset is better achieved by using topology, to walk a triangulation, than to brute force the geometry. While an index

Implementation	Nodes	Topological relationships	Buckets	Atomic functions
OGC SF TINZ			•	
PostGIS Topology	•	•		
Oracle SDO_TIN	•		•	
pgTIN	•	•		

Table 17: Comparison of current implementations. A bullet indicates the implementation satisfies the criteria.

on each simplex in a TIN is not feasible, an index on each bucket might be, combining a normal spatial index *on* the bucket level with walking using explicitly stored topological relationships *inside* the bucket. The following additional criteria for storing massive TINs are thus defined:

- Buckets must be used
- Spatial index only on the bucket level
- Topology is used to solve point location

An overview of the current implementations and the criteria, together with the criteria found in Chapter 2 is presented in Table 17. Of only two spatial TIN types existing in commercial DBMS the Oracle implementation is mature, but it lacks topological relationships. The OGC Simple Features also lacks topology and is less suited to store massive TINs because it does not use a node table, resulting in a large storage size. These existing solutions also do not have atomic functions. PostGIS topology does store topology explicitly but requires several large tables and indexes and does not use buckets, nor does it have atomic functions. PostGIS topology is large in size and has to be converted to geometry before it can be used for spatial queries, making it unfit for storing TINs. The pgTIN prototype also does not use buckets, but is very small and requires only two tables.

4 STORING TINS WITH THE MULTISTAR APPROACH

Based on the criteria and the examples in the previous chapter, I can define what I refer to as a multistar structure. This structure takes from Blandford et al. [2005] and the database implementation of it by Ledoux [2013]. For massive datasets, however, we need a secondary structure to find our buckets as used by Oracle [2015] and the point cloud extension of PostgreSQL as shown in 3.

I use a streaming approach for construction and a tiling approach within the database to solve the problem massive TINs represent.

4.1 MOTIVATION

Based on the criteria in Chapter 2 and the research in Chapter 3 an efficient TIN structure has the following properties:

- Stores nodes explicitly
- Uses atomic functions
- Uses buckets for subdivision
  - to solve the massive data problem
  - with spatial index on the extent of the bucket
- Stores topological relationships
  - for specific TIN functions/queries
  - for traversing inside the bucket

Two possibilities stand out: the triangle array as used by Oracle in section Section 3.2.2 and the star based structure used by pgTIN, see section Section 3.2.4. The first structure lacks topological relationships and the second lacks buckets. I have chosen the star structure as the best option due to its smallest size and the fact that I can work with a prototype, while the Oracle technology is closed source.

The star structure has certain drawbacks that should be fixed, such as the large number of rows, which I hope to fix with buckets. Some drawbacks that are inherent to the structure are the inefficient storage of attributes for triangles, which can be mitigated by storing attributes on the bucket level.

4.2 OUTLINE OF THE MULTISTAR STRUCTURE

The required elements in the star structure are a point identifier or id, coordinates describing the point itself and an array of neighbours, referring to other other neighbouring points as described in the star structure Section 2.6.1 and seen in the table Table 18.

By using buckets, multiple rows and thus points are stored in one field, which will therefore be a binary value that has to be packed and unpacked

id	x	y	z	link[]
1	123.0	456.0	78.0	[2,3,4,5]

Table 18: Star based structure.

according to certain rules. Using buckets also requires a method to find the right bucket, thus the bounding box of the bucket is also stored.

If we pack all the coordinates into a structure, using the order as an id, unpacking them is straightforward. Using the id and the size of the x,y and z coordinates together, the right offset is found and the next values will be the requested coordinates. This logic is based on the fact that the coordinates are always represented as x, y and z with a fixed size.

Coordinates are often stored as *double* or *float* taking 8 or 4 bytes. One point thus takes 3 * 8 = 24 bytes in storing x,y,z coordinates.

However, the array of neighbors has no fixed size, although it is on average six [Okabe et al., 2009]. Instead it can be as high as 50 or as low as 2 (see section Section 5.8). If we pack them all together they cannot be retrieved as point coordinates can. Therefore, a structure is needed that somehow defines the end or starting point of the array. This can be done in several ways:

- Stating the length of the array at the beginning of every array.
- A special marker at the beginning or end of an array.
- An auxiliary array with offsets.

I have chosen for the external array, because it offers two distinct advantages:

1. The external array can be accessed directly in $O(1)$, while the internal methods require looping over every item in $O(n)$.
2. Using an external array offers some extra fields and attributes, such as storing the total number of points.

The points and the stars are stored separately to enable fast access to point coordinates, without the request being slowed down by skipping over stars.

This completes our structure. We have a bucket as a row in a database with an id, a bounding box (using the PostGIS box type), an array describing the offset of the stars, a packed points binary value and a packed stars binary value.

4.2.1 Included metadata

This structure alone is not enough to actually retrieve all the points and stars. For example, while the offset of the last star is known, its length is not. The total number of pointers is the star is unknown, as well as the number or id of points lying either on the convex hull or referring to other buckets. Therefore, some metadata is included in the data structure.

An example of this data structure, including a small TIN for demonstration is presented in Figure 25. The example is split into two buckets in order to demonstrate the method to reference other buckets. The bounding box is omitted from this example.

Because 0 is a reserved number for identifying the convex hull in a star (see also pgtin in Section 3.2.4) and negative numbers are reserved if we use difference encoding (see Section 4.3.2), the point ids start with 1. Since the

array positions (starting with zero) should conform with the point ids, the first array item is free and can be the number of points stored in the bucket.

Using another small hack, some numbers in the offset array are negative. This is done to mark either this bucket or certain ids. When parsing these numbers, the absolute value should be used.

- If the first item in the offset array, thus the total number of points, is negative, the bucket has points that lie on the convex hull.
- If other offsets are negative, the referenced star contains either a reference to another bucket or is zero.

In this way the points on the convex hull or those connected to other patches are quickly found for use in certain queries (see Section 4.4). Buckets containing such points can also quickly be found in the same manner.

4.3 BUCKETS

For the creation of a massive TIN from the AHN2 dataset, I use lastools as discussed in Section 3.1.2, which uses a quadtree. Hence, the buckets of the multistar structure use quadtree cells as seen in Figure 26. Since the implementation in Section 5.2 it has become theoretically possible to create other buckets forms and sizes, but here I will discuss the multistar using quadtree cells.

The level of the quadtree used is user input when using lastools and one can expect an average number of points for each bucket based on the total number of points ie:
average number of points per buckets = total number of points $/4^{(level)}$.
The number is an average because the filtered AHN2 is a irregular dataset and the quadtree divides the bounding box into even regions.

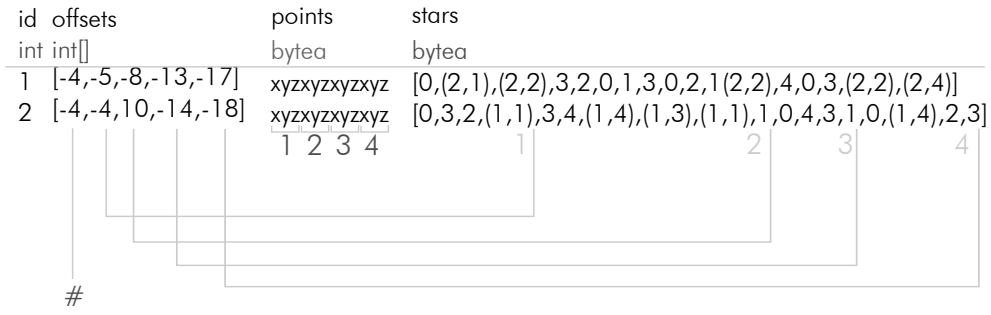
As seen in Figure 26 the buckets can have varying size. The user input level is taken as the deepest level, but when a bucket contains less than the average expected number of points divided by four, the bucket is merged with its three neighbours. If the merged buckets together still contain less points than the previously defined threshold, they will be merged again with its three neighbours, creating a bucket 16 times as large as the original bucket on the lowest level.

The buckets are numbered with a Morton key, but with the addition of all possible ids from the lower levels e.g. the upper right corner on level 2 is normally assigned the Morton code of 15, but 4 is added four the number of ids used on level 1. Since the buckets are bounding boxes, they can be easily indexed by the available spatial index, such as an Rtree.

4.3.1 Referencing other buckets

In the original implementation by Ledoux [2013], the only special attribute in a star is the handling of the convex hull of the TIN. With buckets however, points can have neighbors that lie in the other bucket. This depends on the dataset used. I distinguish between three options of using buckets.

ISLANDS When using buckets, they can be *islands*, not referring at all to other buckets (Figure 27a). Each bucket becomes an object on its own. Storing buckets without references is the fastest, but also the most destructive



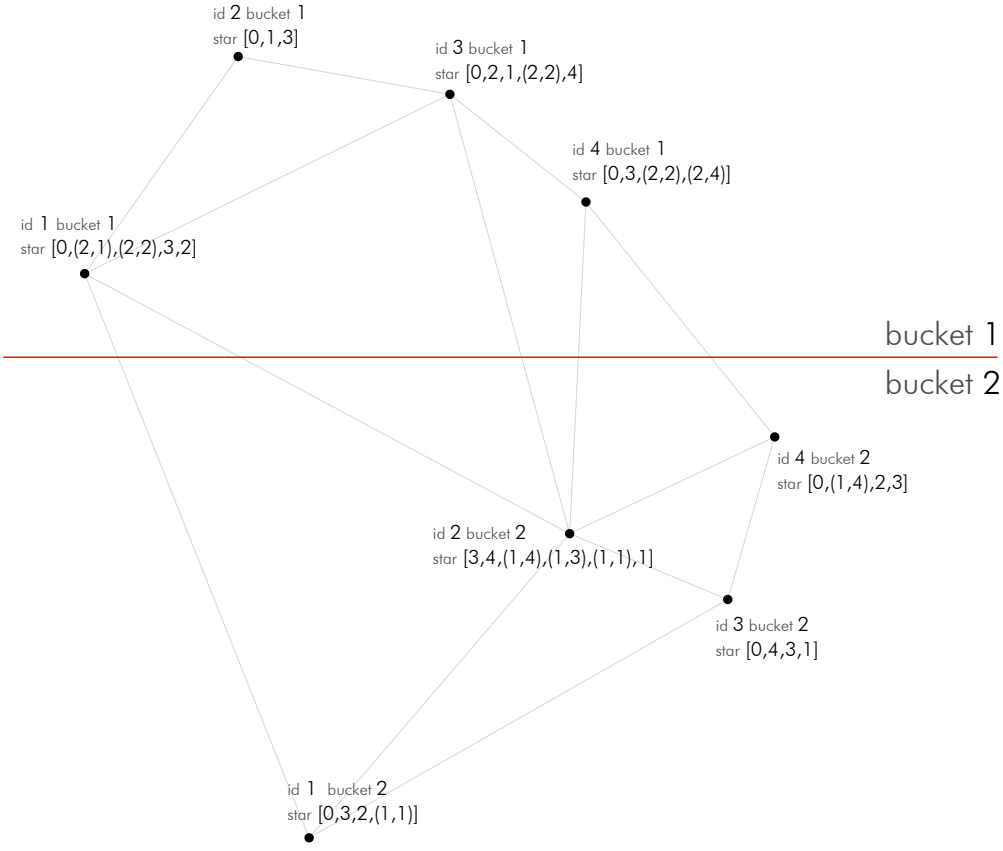


Figure 25: Multistar structure as a database row.

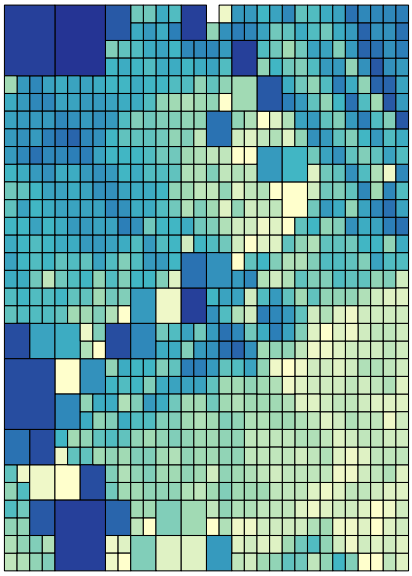


Figure 26: Quadtree used for buckets. Colored by number of points from yellow to blue. Dataset demo.las

method, in the sense that the information which connected all the buckets together in one big field is lost. A dataset such as the AHN2 thus does lose valuable information, where buckets need to be stitched together in order to do specific TIN operations on them such as watershed modelling. On data structures that cannot or do not store topological relationships, such as the SF, this is the only option.

If buckets are islands, the original question of what happens with a TIN when split appears again. Discarding all the triangles on the intersection with the boundary of a bucket will create gaps, so specific rules are necessary. I have chosen the following simple rule:

- When a triangle intersects the boundary of a bucket, it is assigned to the bucket with the lowest id.

Because the ids of the bucket are assigned Morton keys, most buckets will "eat" to the right and upper edge as seen in Figure 27a. This can also be seen in the implementation of the MultiPolygonZ in Chapter 5, Figure 32b. It also shifts the bounding box of each bucket as seen in Figure 28 because of now overlapping points, which makes queries on the edges of buckets slower.

LOOSE BUCKETS If buckets do refer to each other, the nodes need to store some information when referring to another bucket, such as a flag. This would create a sort of *dangling* reference, which only indicates a node has to be searched in a different bucket, which has to be searched for (Figure 27b). Only marking a reference to another bucket needs a spatial search for the bucket, which would be slow.

- Mark the id without referring to an actual patch. For example, by making the id negative as seen in Table 19.

node id	link[]
1	[2,3,4,5,-1]

Table 19: Referencing other buckets by a marker: storing only the node id and a flag (negative sign) if the node is in another bucket than the current bucket.

node id	link[]
1	[(a,2),(a,3),(a,4),(a,5),(b,12)]

Table 20: Storing direct references to buckets (a and b) and node ids.

DIRECT REFERENCES The last option is directly referring to other buckets, resulting in storing bucket ids for every point as well (Figure 27c).

Referencing to a point id in another bucket thus needs a special attribute, in order to locate the right point in another bucket and not in its own. In 3 pairing functions are discussed as is the approach taken by Oracle [2015]. Some options that are thus available for referencing to other buckets:

- Pair the bucket id and the point id together using a pairing function.
- Store the bucket id and point id separately.
- Store absolute bucket ids or relative to the current bucket

Pairing each point with its bucket id is one option (as seen in Table 20), but requires an unpairing function on access. Marking the points which need unpacking requires a check on access. Depending on the pairing function, these options can be both fast, but, depending which algorithm is used, the pairing function is normally slower.

When only references to other buckets are paired with the bucket id, the pairing function should then produce a number that is higher than the number of points (and thus pointids) in the bucket in order to avoid collisions.

I have chosen to store references directly by pairing every node with its bucket id. In the next section (Section 4.3.2) it becomes clear that storing absolute references or relative references does not matter. The example of the data structure in Figure 25 shows the abstract pairing of the bucket with the point when needed.

4.3.2 Pairing functions and space filling curves

Interleaving these bits would be a real pairing function.

The pairing function used, is simply packing two small integers of 2 bytes together to create one integer of 4 bytes. A consequence is that pointids in a patch cannot exceed the smallint size, which is defined by database as a signed small integer, thus 32767 points. Since the size of the buckets can be controlled, this should pose no problems. Using C as programming language however we can use unsigned small integers, resulting in 65365 points. The bucket ids also cannot exceed this value if we store the absolute value, resulting in a multistar with a maximum capacity of $32,767 \times 32,767 = 1,073,676,289$, which is slightly less than half of the maximum number stored in a integer. A billion points is a good size, but will fit at most around four tiles of the AHN2 dataset. On datasets with small multistars the limit of 32767 bucket ids will be reached quickly and the

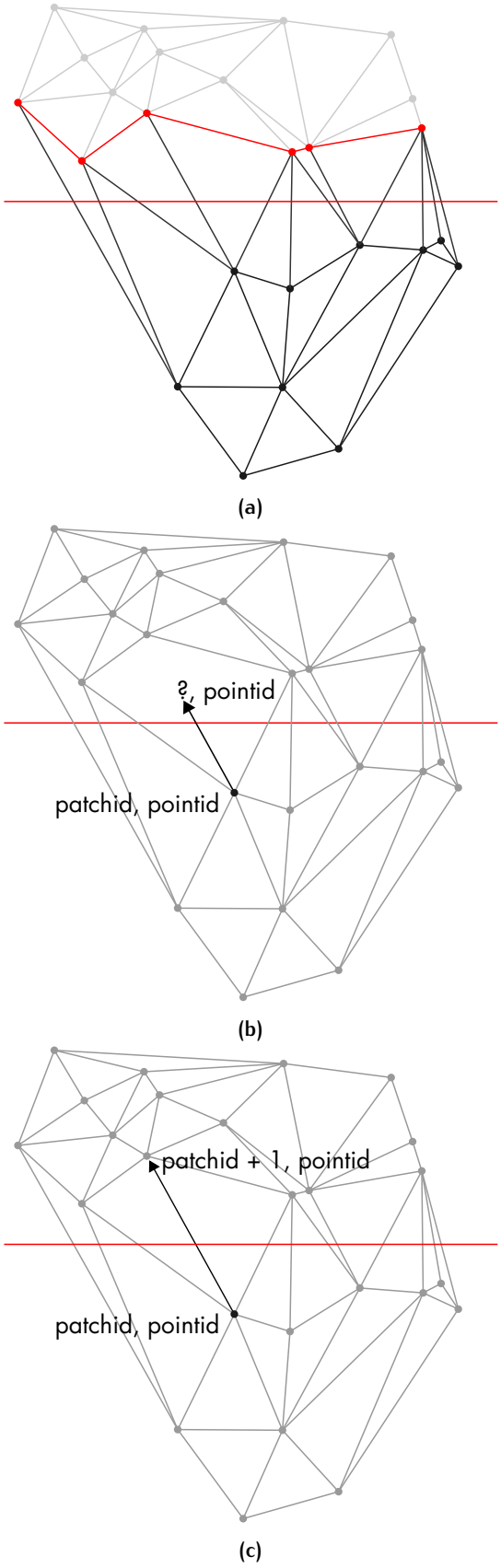


Figure 27: a) Not referring to other buckets by dividing triangles, resulting in overlapping points. The black triangulation and the gray triangulation on top are two different objects now. (b) Storing a flag for each neighbour in another bucket. (c) Storing explicit patch ids, relatively or absolutely to another patch.

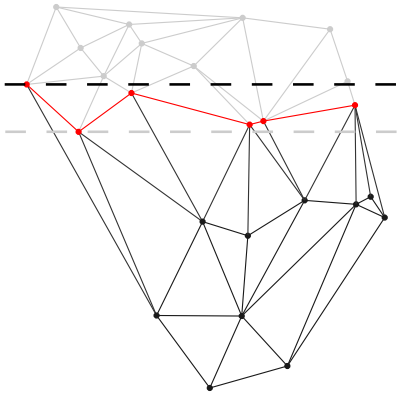


Figure 28: Bucket division line redefined by triangles with the new overlapping bounding boxes.

storage should switch to bigint, allowing for far greater ranges, but also increasing storing size by roughly 50%. The absolute maximum in database is the current `bytea` limit at 1GB [Douglas and Douglas, 2003] but this would require tens of millions of points in a bucket.

It is also possible to only store the id relatively to the current bucket, increasing the maximum number of ids. This is called *difference* encoding, used by Blandford et al. [2005] in their star based data structure. However, since I use a Morton key (explained in Chapter 3) on my buckets the expected advantage of the relative bucket id disappears. The maximum difference between bucket ids happens in the middle of an iteration of the z-order curve. The maximum difference, with the assumption that references to other buckets happen only to each buckets 8 neighbors (thus assuming that triangles are never greater than their buckets), is $\frac{n}{2} + 1$. This means that a relative storing of bucket ids from a quadtree using morton keys will never be (a bit) smaller. Other methods for ordering ids, such as simpler space filling curves as the row ordering will have a maximum id difference of one $row + 1$, or in a quadtree $\sqrt{n} + 1$. Other more complex space filling curves will have a much greater maximum distance.

The possibility of any advantage using difference encoding by row ordering however dissapears with using a quadtree. Buckets are now variable in size as seen in Figure 26, thus making the maximum difference between bucket ids, whatever the ordering used, unknown with a maximum limit of $n - 1$. In that case there is no difference with storing absolute bucket ids. Therefore the multistar implementation uses absolute references to other buckets.

4.3.3 Parallel SQL queries

The multistar can be queried from a starting bucket, from which the algorithm finds its own way by walking through the dataset and hence through other buckets. Another method is selecting all the buckets in our zone of interest by using the spatial index and running an algorithm confined to each bucket on each of these buckets. The latter works from the index on the buckets, while the first uses the stored topological relationships of the TIN itself to traverse the buckets. Both methods work, but only the latter is easily used in parallel processing. By using parallel queries each bucket

is seen as a separate object, which also solves eventual memory problems with the walking approach. Two options can be chosen:

- Subdivide the query to the buckets and introduce a ruleset to fix edge cases.
- Let the buckets be islands, not stitched together.

Both options create extra overhead and problems. Subdividing range queries into buckets create walking problems, but these are solved in Algorithm 4.1, instead of using the convex hull of a bucket, the items referring to are buckets are used. The ruleset then needs to adapt for the gaps in the range query i.e. walking backwards on the edge of buckets.

Buckets as islands can only work when the buckets are convex. This is only the case when using *regular* tessellations and even then the border is not convex. Only when there is overlap i.e. each bucket has a triangulation exceeding the size of the bucket, can the area inside the bucket be considered convex. This creates always overlapping triangles on bucket edges, which require a new ruleset to fix.

4.3.4 Height information

Two dimensional spatial queries do not require the minimum and maximum height of points in a bucket to be stored. However, viewshed operations on a TIN require queries based on height information. Therefore, the bounding boxes of the buckets are defined by their x, y and z coordinates (stored as the PostGIS type `BOX3D`). This enables specific height based queries e.g. selecting all buckets lower than 0m for quick visualisation of flood modeling.

4.4 INDEXING AND SORTING

Indexing is, just as it is done with the star based structure not done by a spatial tree, but by using the explicit stored topology. Several algorithms are used from the prototype developed by Ledoux [2013], but are adapted to work with the new bucket structure. Although indexing is also used to store items closer together on disk, the internal use of the multistar will load the complete bucket in memory, negating sorting advantages. These algorithms include:

- jump and walk by Mücke et al. [1999]
- straight walk
- window query by Zhu [2000]

It is important to note here that all the spatial functions implemented here start with the same simple algorithm, the random jump and walk by Mücke et al. [1999] for point location. For a straight walk, the first point of the query line is used in the point location algorithm. The triangle returned is then used as the starting triangle for a straight walk. This also applies to window queries, as well as other atomic operations, such as degree and slope, which build upon these spatial functions.

With a line intersection query, the first point of the line could be outside the TIN (see Figure 29a), which means that walking algorithms will fail to

find a starting triangle. A relative simple intersection algorithm is proposed to find the starting points, shown in Algorithm 4.1. The two returned points are part of the starting triangle where the line enters the TIN. The algorithm selects a random point on the convex hull, easily found because this information is stored in the offset array. An orientation test is done for the starting point from which is decided in which direction the next point on the convex hull is going to be checked. Looping over points on the convex hull until our orientation test flips the output, which means the current and the previous point form the last edge of our starting triangle as demonstrated in Figure 29b.

Algorithm 4.1: The convex_hull_intersection algorithm.

Input: a line a, b and a star based data structure s.
Output: if an intersection is found, two starting points

```
1  find_random_point_on_convex returns a random point on the convex hull of s.
2  startingid = find_random_point_on_convex(s)
3  orient returns line orientation: 1 for left or 0 for right of line
4  orientation = orient(startingid,a,b)
5  if orientation == 1 then
6      startingid left of the line, looping counter-clockwise
7      while true do
8          get_next_in_star returns next point on convex hull
9          i ← get_next_in_star(startingid)
10         o ← orient(i,a,b)
11         if o ≠ orientation then
12             return i, startingid
13         startingid ← i
14 else
15     startingid right of the line, looping clockwise
16     while true do
17         get_previous_in_star returns previous point on convex hull
18         i ← get_previous_in_star(startingid)
19         o ← orient(i,a,b)
20         if o ≠ orientation then
21             return  startingid, i
22         startingid ← i
```

Another problem is the fact that a constrained Delaunay Triangulations can cause a walking algorithm to loop indefinitely around the same point Devillers et al. [2002]. To combat this problem, instead of the same edge being chosen for an orientation test in walking, a random choice has been introduced, to be able to break free from such a loop.

4.5 STORAGE SIZE OPTIMIZATIONS

There are many ways to further optimize the storage size of this structure as seen in Section 3.4.3. Those compression techniques are only applied in the

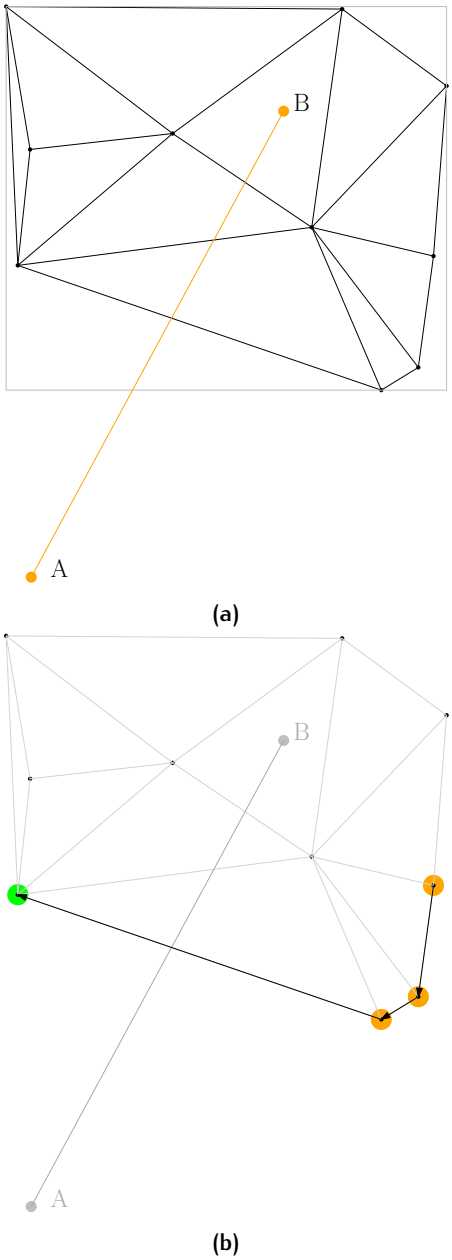


Figure 29: Line intersection with starting point outside the TIN. (a) The line AB intersects the TIN, but A lies outside, preventing the jump and walk algorithm from completing. (b) The convex hull intersection algorithm (Algorithm 4.1) returns two starting points (green point and the previous point) which are part of the triangle in which the line AB enters the TIN. The algorithm traverses the convex hull until an intersection is found.

data structure, as described previously. This includes relative referencing and pairing of numbers.

A simple way to optimize the coordinates storage size would be to scale and cast them to integers, e.g. used by van Oosterom et al. [2015]. This is possible because of the limited decimal places of the AHN2 dataset (up to two). 448512.49 (double precision) would become 44851249 (integer), which halves the storage space requirement.

Although this would be possible and probably advantageous, this has not been implemented. Coordinates are the only entity shared across all the different implementations and thus enable a fair comparison in storage size, because only then the data structure itself can be compared. Although the comparison is done on *total* storage size for each data structure, scaling coordinates can be done for *every* structure, meaning it is not an advantage for any *specific* data structure.

4.6 INTEGRITY AND VALIDITY CHECKS

It is not easy to check for errors in the multistar data structure since triangles are implicitly stored and the data itself is hidden in binary form. Methods to check the validity and integrity automatically are therefore needed.

Several simple checks have been written in Python, see appendix A. These are mainly focused on the topological structure and thus the validity. These tools check for every node if its star is ordered counter-clockwise and if all the pointers in its star results in nodes with stars that also point back to the node. This ensures a data structure that is both connected is usable by the implemented algorithms.

There are also simple integrity checks to make sure that all the necessary fields and tables are there with the correct attributes, as explained in Figure 25. Such integrity checks consist of checking every star for the minimum of two pointers and if a zero occurs in a star if it is put in front.

4.7 ATOMIC TIN FUNCTIONS

In order to make a TIN structure useful, several atomic functions should be implemented. In Chapter 2 these have been defined as:

- Slope
- Aspect
- Local minimum
- Local maximum
- Degree

All these functions have been implemented in the Multistar. Local minimum and local maximum are boolean functions comparing the height of the requested point to the height of the vertices in its star. Slope and aspect are standard functions for planes in a DTM, adapted from Moore et al. [1991]. The degree is the number of vertices in the star of a given point, giving information on the complexity of the triangulation at that point.

4.7.1 Thinning and simplification using the implicit TIN

A simple thinning function has also been implemented, that for every point, takes the sum of difference in height with the vertices in its star and compares this sum to a user defined threshold. This is by no means a very intelligent filter, but it enables to filter distinct points from less relevant (flat) points. Similar methods are applied by the drop heuristic method by Lee [1989], which also determines the relevance of a node by its neighbours. The relevance is determined by the difference in height of the node and the triangulated terrain without the node.

With this non-random sample much of the terrain is still represented by generating a DT from it on the fly, which is not unlike the pyramid used in the Multiscale Implicit TIN of Kidner et al. [2000].

An array such as the offsets now stored can be used in the same way to represent levels of detail, storing an extra attribute of significance for each point. Items of special significance, such as those present in the constraints of a CDT can always be requested at each LoD. However, the CDT should be a *conforming* DT.

For each bucket a different level of detail can be requested and triangulated on the fly, enabling visualization purposes.

4.8 STORING EXTRA ATTRIBUTES

Using the Multistar only permits storing of attributes at the bucket. A database table using multistars requires the fields as described in Section 4.2. The user is free, however, to add extra fields and hence extra attributes. This could be done for the TOP10NL3D dataset, which has multiple attributes connected to each bucket.

Storing attributes at point level is more difficult in the multistar. It would require adding these attributes to the binary points field. The binary field would then consist of $n \times xyzabc$, where a b and c are attributes instead of $n \times xyz$. These attributes must have a fixed size for this to work. The algorithm used should dynamically determine the number of attributes and thus the size of each point. The point cloud extension in PostgreSQL uses a separate table with point cloud schemas to do determine this exact thing.

At an edge or triangle level no attributes can be easily stored using this structure. Edges and triangles are implicit in this structure, which is an advantage in storage size, but disadvantageous for storing attributes at such levels. Edge attributes can be stored twice (one for each point of the edge), using a structure such as pgTIN e.g. to specify the constraints of a CDT.

4.9 DRAWBACKS

Each data structure has its disadvantages, which should be known in advance in order to decide whether it should be used or not. The disadvantages follow from the star based structure, which stores triangles indirectly by using a variable length star. The indirect storage of triangles makes a complete reconstruction of all triangles slower than explicitly stored triangles and it makes it difficult to store attributes for each individual triangle.

The variable size length of each star results in the need for an offset array in order to lookup the right star in the multistar data structure, mainly increasing the size of the data structure. Although updates are out of scope, replacing a star within the multistar data structure is a costly operation, because a complete multistar has to be re allocated and a part of all the offsets replaced.

- Triangles stored indirectly, resulting in slower reconstruction.
- Variable length, making updates slower.
- Needs offsets array, increasing the storage size.

5 IMPLEMENTATION, EXPERIMENTS AND COMPARISON

The main research question asks for an *efficient* approach to storing massive TINs. Efficient is defined by the performance of the following items and these will be compared in this chapter for the different existing approaches.

- Loading time of the TIN, including construction
- Storage size of data structure
- Storage size of index
- Time of spatial queries
- Time of atomic functions

In Chapter 2 criteria for efficient approaches were defined, by which current implementations were compared in theory in Chapter 3. A prototype based on these criteria was proposed in Chapter 4. This chapter will describe the workflow for setting up the practical comparison, the implementation of the developed prototype and a practical comparison of approaches for storing massive TINs in a DBMS.

The hypothesis that the criteria found in Chapter 2 and implemented in the multistar prototype are both faster in spatial queries and smaller in storage size than existing solutions.

5.1 WORKFLOW

Several extracts from the AHN2 filtered dataset, ranging in size and number of points, are used. All datasets are publicly available for download, with hyperlinks in the Appendix B. For retrieving the region of interest from the Actual elevation information of the Netherlands (AHN2) one must lookup the tiled datasets and find the required tiles and if necessary, merge them[1]. The following subsections are used:

- `demo.las` A small point cloud dataset of around 5.2 million points
- `rijswijk.las` A very small point cloud dataset of around 320 thousand points
- `g37en2.laz` A massive point cloud dataset of 367 million points, only the filtered ground points.

With a subsection of the AHN2 dataset, lastools is used to create a TIN. Most of the data structures and its variants have been created using custom Python code, which uses the output of lastools. These workflows can be seen in Figure 30 for single type data structures and in Figure 31 for multitype data structures, such as the multistar.

1 A web tool to automate this tedious process is available at MATAHN

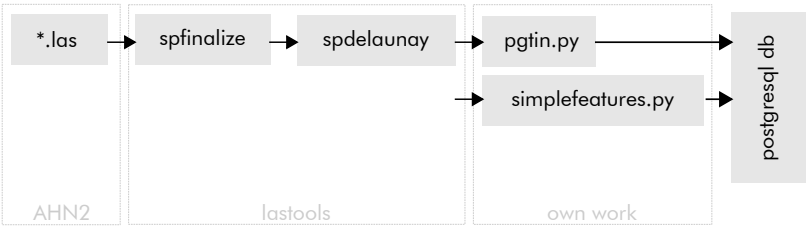


Figure 30: Workflow scheme for single type.

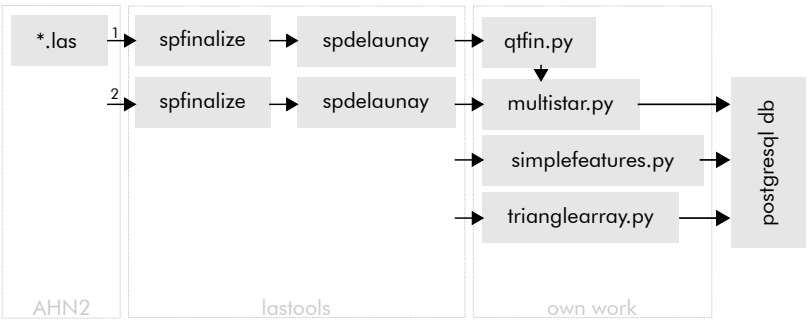


Figure 31: Workflow scheme for bucketed types.

For every TIN structure, a spatial index was created on the spatial field, or if not applicable, on the bounding box field. The exact structures can be found in the Appendix E. The following TIN structures are available for testing, where single type indicate a simplex per row in a database e.g. a Triangle in each row and multi type indicates several simplexes per row e.g. a MultiPolygon in each row.

SINGLE TYPE

- OGC PolygonZ
- OGC Triangle
- pgTIN, as implemented by [Ledoux, 2013]

MULTI TYPE

- OGC MultiPolygonZ
- OGC TINZ
- Triangle array, my own implementation discussed in Section 5.3
- Multistar, my own implementation discussed in Section 5.2

The PostGIS topology is not used for experiments as it is not fit for storing TINs (Section 3.2.3) and the construction never completed (see Section 5.4). Both of the implementations done as part of this thesis are discussed in Section 5.2 for the Multistar and Section 5.3 for the triangle array. The Triangle Array has been implemented to mimick the SDO_TIN type by Oracle.

The experiments are run on a workstation with an Intel i7 2600K, 16GB of main memory running Ubuntu Linux 15.04 and PostgreSQL 9.3 with the database on a 7200 RPM SATA hard disk. Complete specifications for running the experiments in the PostgreSQL database can be found in Appendices B to D.

5.2 MULTISTAR IMPLEMENTATION

The multistar has been implemented through two programs developed in C. One program can decode the multistar to multiple rows with points and stars. The second program is based on the pgtin prototype by Ledoux [2013], which has been adapted to work with the first program and the nature of the multistar. Main functionality added or changed is described in Chapter 4. Other functions are adapted to work with multiple buckets and are described in Appendix B.

The multistar structure uses a spatial index on the bucketlevel and for spatial queries a starting bucket has to be found first. Within the starting bucket topology can then be used by the implemented code. A typical sql query in PostgreSQL for creating a profile by line intersection is as follows:

```
/* Line is defined as l(a,b) with a{12.0,34.0} and b{56.0,78.0}
    */
SELECT profile(12.0,34.0,56.0,78.0,id)
FROM multistar
WHERE bbox && ST_Point(12.0,34.0);
```

The function profile takes in the coordinates of the line, as well as the id of the bucket in which the first point of the line lies. This id of the bucket is the result of the intersection of the bucket's bounding box with the first point of the line.

Integers (32 byte) are used for the offset array and the star. Thus, stitching functions use two small integers (16 byte). Before big integers are needed, it would be optimal if both small integers are used to their capacity, thus having as many buckets as there are points in each bucket on average.

Two buckets at are cached in memory as a queue, as soon as a bucket is requested that is not cached, the requested bucket will replace the oldest cached bucket. When only caching one bucket at a time, a worst case scenario requests bucket a, then b, then a again and then b again, which happens when crossing buckets. This is prevented when caching two buckets at a time. At the intersection of three or four buckets, caching two buckets might not be enough, but these cases are much rarer (four occurences at the corners for each bucket).

5.3 ALTERNATIVE IMPLEMENTATION

Practical comparisons are limited to PostgreSQL in this thesis, thus the triangle array type used by Oracle in Section 3.2.2 is not available. Comparing two different data structures in two different databases cannot be precise. Also, the loading of data into Oracle presented problems (see Section 5.4). Therefore a similar data structure, here called triangle array, has been implemented in PostgreSQL, which is presented in detail in Appendix C.

Compared with the multistar data structure seen in Figure 25 instead of storing an array of stars, the triangle array stores an array of triangles, where each triangle consists of three pointers to points. Since the triangle always has three vertices, each triangle in the array has a fixed size, hence an offset array used by the multistar structure is not needed. Just like the points, the id of the triangles is derived from their respective index in the array.

The pairing function used by the multistar structure is used here as well, stitching the bucket id and the point id together when a triangle points to

a point in another bucket. The placement of triangles is the same used as by the construction of the TINZ and MultiPolygonZ structures, placing each triangle in the bucket with the lowest id. When one bucket is reconstructed, the output is exactly the same as the MultiPolygonZ or TINZ types stored by PostGIS.

The only functions currently available return a `trianglez` or `TINZ` in WKB, not unlike the Oracle `SDO_TIN`, which only casts back to geometry. However, the Oracle approach also indexes the triangles inside each bucket, thus enabling clipping inside each bucket, whereas the triangle array prototype implemented here only clips on the bucket level, by using a spatial index on the box3d field. The same line intersection query used in Section 5.2 is as follows:

```
/* Line is defined as l(a,b) with a{12.0,34.0} and b{56.0,78.0}
    */
SELECT ST_Intersection(tinz_bytea(numt,points,triangles,id)::
    geometry,
ST_MakeLine(ST_Point(12.0,34.0),ST_Point(56.0,78.0))
FROM trianglearray
WHERE bbox && ST_MakeLine(ST_Point(12.0,34.0),ST_Point(56.0,78.0)
    );
```

In this query, all the buckets intersecting with the bounding box of the line are selected and each bucket is decoded to WKB cast to PostGIS geometry, which is intersected with the line. In this example tinz_bytea returns a MultiPolygonZ in order to use the standard PostGIS functions.

The triangle array can be expected to be much smaller than the OGC Simple Feature types, and because it does not store topology nor requires an offset table, the structure should also be smaller than the multistar prototype. Nevertheless, the performance of spatial queries can be expected to be a bit slower than the OGC Simple Features, since the geometry used is exactly the same, but has to be constructed first.

The Oracle implementation, although not benchmarked here, is expected to be larger in storage size, since it also stores an index, and faster because it can use the same index to return a smaller geometry, which takes less time to operate on.

5.4 CONSTRUCTION AND LOADING PERFORMANCE

For constructing a TIN from a point cloud internal algorithms in the database and external algorithms outside the database can be used. The upside of using internal construction is that no tools are needed anymore to construct the TIN outside the database and then load it into the database. However, many internal are actually external TIN constructors, using external libraries. This is the case of PostgreSQL, which uses GEOS as its back end to do such calculations.

INTERNAL CONSTRUCTION

- Oracle SDO TIN
- PostgreSQL

EXTERNAL CONSTRUCTION

- Lastools

software	rijswijk.las 50K	demo.las 5M	g37 en2.laz 367M
PostgreSQL	15	3600	-
Oracle	5	!	-
Lastools	0.67	10	888
Triangle	1.8	31	!

Table 21: Comparison of performance in creating a DT. Time in seconds. ! did not complete, - did not test.

- Triangle
- CGAL
- many others...

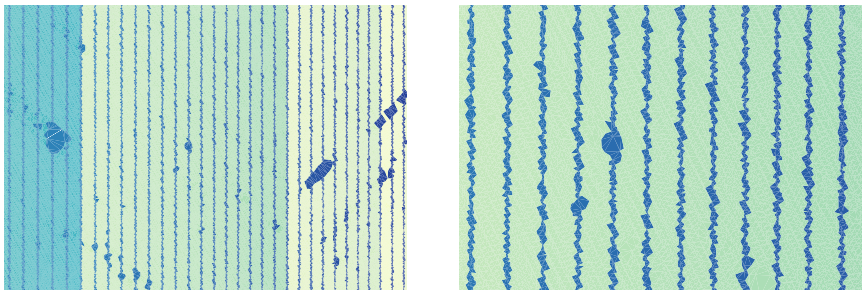
Four programs for constructing TINs are tested: two internal algorithms and two external. Many more programs have been researched in Chapter 3 but only lastools can do the computation streaming. Having `Triangle` [Shewchuk, 1996] here is mainly for a speed comparison. The results are shown in Table 21.

The external algorithms work very well on smaller datasets. The database implementations start out promising as well, but their performance drops considerably on a relatively small dataset with 5 million points. Oracle did not complete on 5 million points and was killed after more than 24 hours. PostgreSQL, on the other hand, did run the same dataset in an hour. However, extrapolating this to a dataset of 367 million points - even when assuming linear increase in runtime - would result in a construction of 72 hours. Thus database constructors do not scale very well and are *not fit* for massive TINs. The construction has to be achieved with an external algorithm.

The external algorithms perform better, reporting very fast times. However, when running the 367 million dataset through Triangle, it cannot run because of out of memory errors. Lastools is the only software which runs on massive datasets. Internal algorithms will also run out of memory, which is a problem for full scale terrains used by the implicit TIN Kidner et al. [2000].

The output of these internal constructors has been discussed in Chapter 3. Nonetheless, the results Oracle produced were unexpected. Where PostgreSQL outputs everything in one TIN geometry, Oracle creates buckets, which is great on paper. The results as seen in Figure 32 are the normal output of a point cloud that is roughly twice as wide as it is high. The buckets created, however, are completely vertical and each bucket vertical is separated by a very small bucket with the width of only one triangle. As explained in Chapter 2 buckets should be close to a square in order to equalize performance for all queries. It is unknown what causes this but at the time the triangulation inside Oracle has to be considered broken - both in terms of running time as in its output.

The TINs are transferred into the database with the Python scripts seen in Figure 31 and described in detail in Appendix B. These scripts convert the output of lastools into the appropiate data structure and transfer it to the database. The buckets created by the Python scripts are shown in Figure 33. In Figure 33a the bucket extents for the Multistar data structure are shown when run on demo.las. By using the same quadtree, the MultiPolygonZ buckets are presented in Figure 33b.



(a) (b)

Figure 32: Buckets of the SDO TIN type in Oracle. (a) Zoomed out. Skinny vertical buckets. (b) Zoomed in. Close up of vertical buckets.

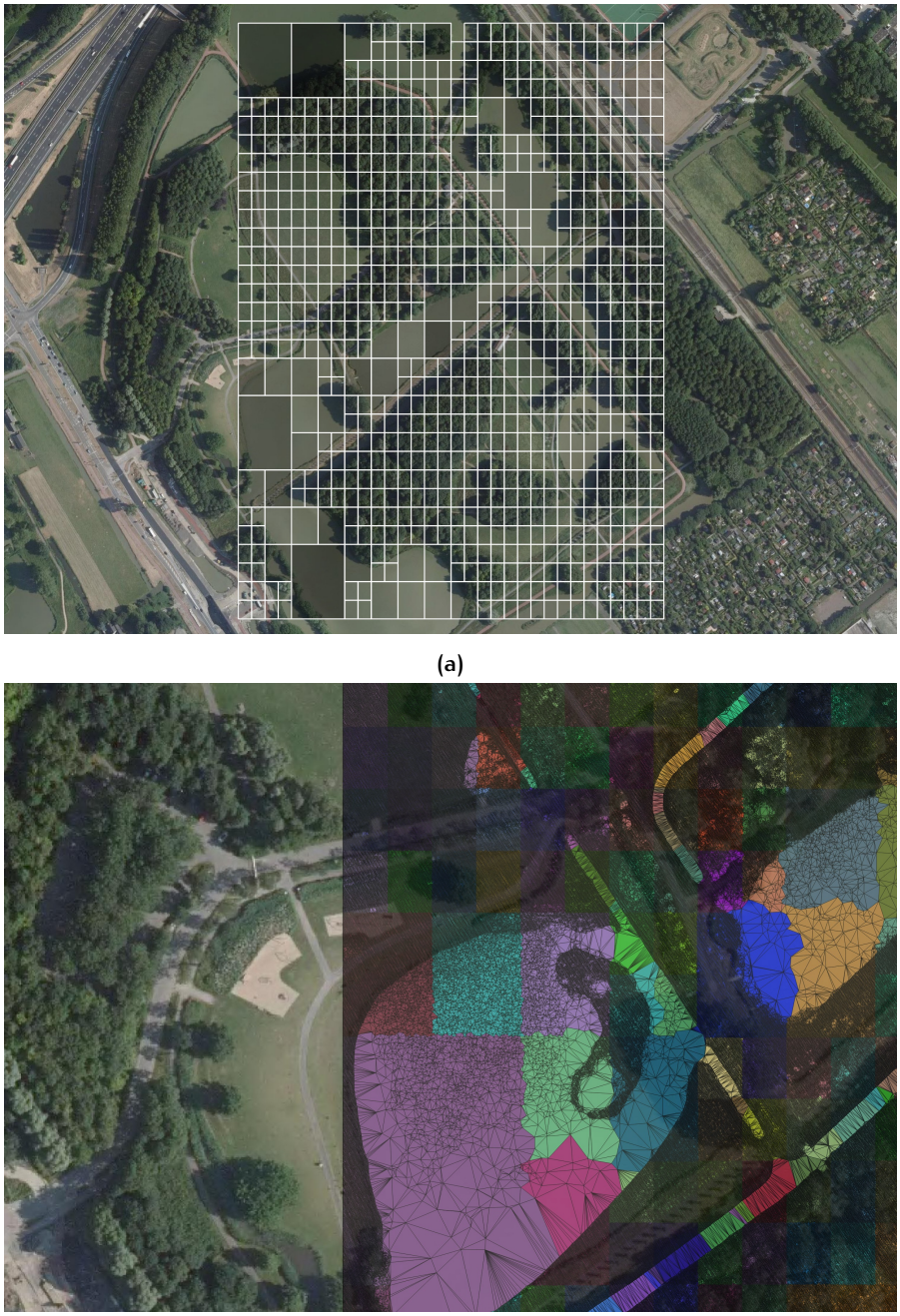
Data structure	Generation and conversion	Creating indexes	Total
Multistar	171	7	178
TINZ*	209	4	213
Triangle Array	160	4	164
pgTIN	73	11	84
TriangleZ*	119	172	291

Table 22: Comparison of performance in converting the output of lastools to database, using different Python scripts. Dataset demo.las. Times in seconds, generation and conversion includes generation of TIN. *TINZ and MultiPolygonZ, Trianglez and PolygonZ have identical performance, differing only in a few bytes.

The various data structures are quite different in complexity, e.g. the `multistar` requires both buckets and stars to be calculated, while the `pgTIN` requires no conversion at all. So, different data structures result in different loading times, as seen in Table 22. Single types are faster in generation and conversion than multi types, but they do take a longer time to create indexes. Note that the creation of an index on every TriangleZ takes a long time, while the creation of a binary tree on the same number of features as used by pgTIN takes only eleven seconds. PostGIS topology took several hours to process only a handful of MultiPolgyonZ geometries, projected to take weeks. It is therefore omitted from this thesis.

When using a massive dataset, the time it takes to convert the TIN into the database amounts to several hours as seen in Table 23. The TriangleZ type has not been used, since the expected size would exceed 100GB. The TINZ type conversion did not complete because the database consistently ran out of memory during vacuuming[2], which is probably caused by large size of the data structure. The other data structures did convert completely and without errors. Although these data structures take a long time to load, it should be noted that this step is only done once. The creation of an index on the pgTIN data structure, which does not use buckets took up a significant part of the total construction time.

2 Even with autovacuum off, the database became unresponsive after several thousand rows



(a)

(b)

Figure 33: Buckets of the multistar and TINz. (a) Demo.las subdivided using quadtree using the Multistar. (b) Zoomed in demo.las. TINz divided in the same way with randomly colored buckets. Original quadtree levels are still there but more fuzzy. On water there are much less return points, creating bigger triangles.

Data structure	Generation and conversion	Creating indexes	Total
Multistar	265	0	265
Triangle Array	243	0	243
pgTIN	95	21	116

Table 23: Comparison of performance in converting the output of lastools to database, using different Python scripts. Dataset g37en2.laz. Times in minutes, generation and conversion includes generation of TIN. Simplefeatures did not complete loading because of the database running out of memory.

Polygons	Table size	TOAST size	Indexes size	Total size
10240	0	14.0	0.02	14.02
2560	0.02	13.6	0.02	13.64
640	0.46	13.0	0.10	13.56
160	14.4	0.064	0.34	14.82
40	15.5	0.01	1.26	16.94
10	67.1	0.01	4.90	71.91

Table 24: Comparison of storage size of MultiPolygonZ using different numbers of polygons in each bucket. Dataset rijswijk.las. Size in MB.

5.5 STORAGE SIZES

The size of the different data structures and their indexes are compared in this section. In theory (Chapter 2) the SF TINZ is expected to be largest structure, while both the Multistar and Triangle array are expected to be small. Data structures using buckets are also expected to have an smaller index than data structures that do not use buckets.

5.5.1 Different bucket sizes

Different bucket sizes have been tested by splitting the dataset into different bucket levels. Each bucket size has an average number of Polygons stored. In Table 24 it becomes clear that using small bucket sizes is not preferable at all, as the size increases significantly for smaller buckets. Around the 500 polygons in a MultiPolygon we see the transition from normal table space to TOAST (Section 5.6.2). A stable region seems to be between 50 and 500 polygons, where the storage space stays relatively stable and it is not using TOAST yet.

The comparison in storage size for each TIN data structure can be seen in Table 25 and Figure 34a. There is a clear distinction between single and multi data types. Multi-type structure (when stored in TOAST) are roughly *5 to 6 times smaller*. The smallest dataset is the Triangle array, followed by the Multistar structure. The increased storage size of the single type data structures is in part due to the large indexes. These indexes are ten to hundred times larger than the indexes on the multi type data structures, confirming the notion that not every element of a TIN should be indexed. The spatial indexes on the single type geometries such as PolygonZ are two to three times as large as the Triangle Array of Multistar data structure.

When looking at the size of the data structures when used for a massive dataset in Table 26 and Figure 34b, the same patterns are visible. The

Structure	Table size	TOAST size	Indexes size	Total size
PolygonZ	1744	0.08	534	2278
TriangleZ	1673	0.08	534	2207
pgTIN	555	0	112	667
OGC MultiPolygonZ	0.98	209	0.31	210.30
OGC TINZ	1.0	203	0.288	204.29
Triangle array	0.82	101	0.30	102.11
Multistar	1.08	130	0.31	131.39

Table 25: Comparison of storage size of the terrain of a demo.las sample with a quadtree of level 6. Size in MB.

Structure	Table size	TOAST size	Indexes size	Total size
Multistar	12	9390	5	9407
pgTIN	38303	0	7841	48841
Triangle array	52	7396	18	7466

Table 26: Size of datasets for the g37en2.laz dataset. Size in MB.

triangle array is again the smallest data structure, followed closely by the Multistar data structure. The single type pgTIN is five to six times larger, in part due to the very large index. Although a binary tree is used, which is smaller than spatial indexes, the size of the index *alone* is larger than the complete Triangle array structure.

5.6 POSTGRESQL SPECIFICS

The storage sizes of the single type structures seem excessive compared to the multi type structures, two specific PostgreSQL properties are responsible: the first is row overhead and the second is compression by TOAST.

5.6.1 PostgreSQL row overhead

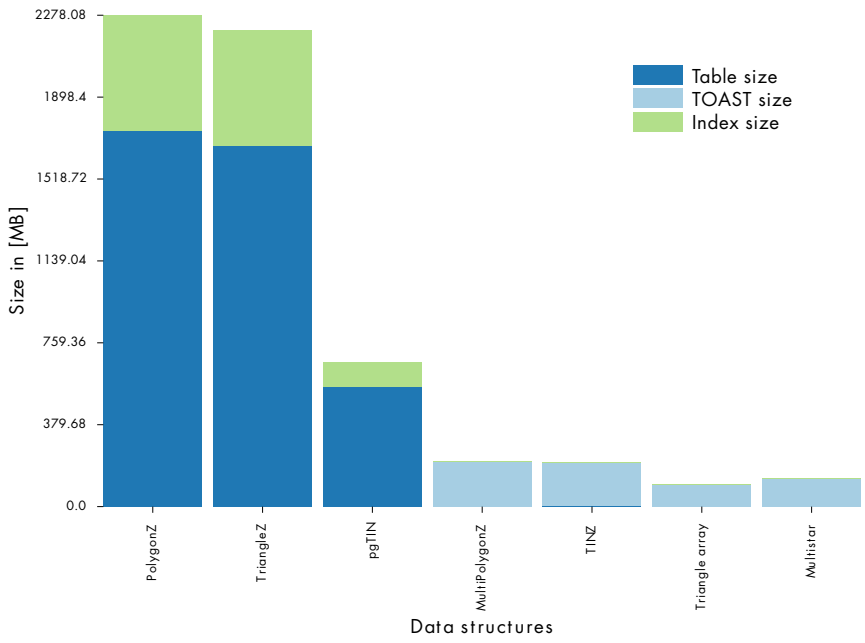
The main explanation for large single types is *overhead*. For every row in the database, PostgreSQL stores some information about it, so called meta-data, in a object called a `HeapTupleHeader`. The size of this object is 23bytes. With padding, in order to align values to 8 bytes, this becomes 24 bytes.

Now it is clear why single type structures are large. For every x, y and z coordinate that is stored in a row, totaling 24 bytes, the total size is *doubled* by the `HeapTupleHeader`.

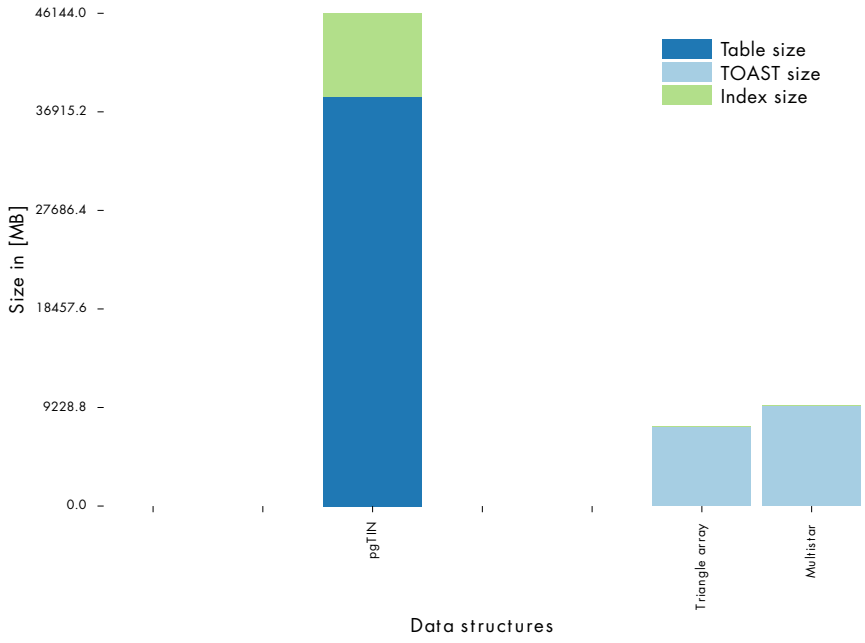
5.6.2 Compression by TOAST

The page size (see Chapter 3) in PostgreSQL is fixed at 8kb[3] Douglas and Douglas [2003]. When large geometries are stored e.g. MultiPolygons or Multistars, these can grow larger than the fixed page size. PostgreSQL does not allow tuples (database rows) to span multiple pages. This problem is solved internally by first compressing the data and if the data is still

3 Changing this requires recompiling PostgreSQL



(a)



(b)

Figure 34: Graph showing the size of different data structures (a) Size of data structures for the demo.las dataset. (b) Size of data structures for the g37en2.laz dataset.

Structure	Compressed size	Uncompressed size	Increase
Multistar	131	206	57%
OGC MultiPolygonZ	205	1164	567%

Table 27: Comparison of storage size with and *without compression* of the terrain of a demo.las sample with a quadtree of level 6. Size in MB.

	float	double	difference
Triangle array	182	187	3%
Multistar	130	135	4%

Table 28: Comparison of storage size using floats or doubles. Dataset demo.las. Size in MB.

larger than the defined threshold (normally 2kb) it is split over multiple pages. This process takes place completely invisible from the user and cannot be disabled completely. The compression, however, can be disabled on columns with the command `ALTER TABLE SET STORAGE EXTERNAL`, but only on creation. This results in a significant increase in the disk space required for data structures that repeat data, such as the MultiPolygonZ as seen in Table 27. The Multistar also increases in size, but less so, demonstrating a more efficient data structure.

TOAST becomes obvious when comparing Triangle arrays and Multistars with either float or double storage of coordinates. Theoretically, we should see an increase around 50 percent, yet the picture is different as we see in Table 28. Compression amounts to a difference of only a few percent. The tables that were tested were completely stored in TOAST.

This is the reason why optimizations described in Chapter 4 are hard to detect and only make the storage size results more opaque. The main reason why multistar works with floats is the output of lastools and AHN2, which do not require more precision.

5.7 QUERY PERFORMANCE

Using different bucketsizes has an effect on the storage size, and it is expected to also have an effect on the query times. The main question to resolve is how algorithms scale with smaller and larger buckets. Larger buckets require less processing because there are less buckets to process in total, but point location performance decreases with larger buckets. Larger buckets will cover more points for a region of interest than smaller ones on diagonal queries. Smaller buckets, however, have a larger proportion of points on the edge of the bucket. Thus they often require more processing of stitched pointers and requests of new buckets. It is expected that the solutions using topology, such as the multistar, are faster than solutions using brute force, such as Simple Features as discussed in Section 3.4.2 and in Chapter 3. Because of the ability to walk it is expected that for data structures using topological relationships the bucket sizes only influence the initial point location.

Bucketsize	Query performance
10240	15484
2560	7505
640	4017
160	2957
40	2502
10	2058

Table 29: Comparison of query performance of MultiPolygonZ using different buckets. Dataset rijswijk.las. Time in ms.

type	l4		l5		l6		l7	
	cold	hot	cold	hot	cold	hot	cold	hot
MultiPolygonZ	37303	36482	5150	4142	526	428	123	76
Triangle array	86762	84418	6649	4696	574	476	345	92
Multistar	123	12	33	12	52	32	52	21
TINZ	327	256	63	52	33	22	14	13
pgTIN	44	43	45	43	43	42	53	43
PolygonZ*	50	12						
Trianglez*	53	12						

Table 30: Comparison of point location by bucket size. Time in ms. Dataset demo.las. * indicates single type, levels do not apply.

Queries are run multiple times, once after a database restart as the *cold* query and then once more, called a *hot* query, since the previous query is probably cached by the database, resulting in better performance.

In Table 29 it becomes clear that using bigger buckets on the MultiPolygonZ type slows down the query. This is expected, since the results are brute forced, the bigger the bucket, the longer the intersection takes. Another thing to notice is the lack of a clear performance hit around the bucketsize of 500. In Table 24 there was a change in storage size when TOAST set in, but this does not seem to have an affect to the performance, if any effect at all.

Take note that with larger bucketsizes, the PostgreSQL query planner can decide to stop using the spatial index, instead using a slower sequential scan. By using the `EXPLAIN` SQL command this can be checked. It is prevented by using the following SQL: `SET enable_seqscan TO off;`.

Bucketsize is defined by the depth of the quadtree, level 5 represents 4^5 buckets thus a bucketsize of around 5000 in the demo.las dataset which holds five million points. pgTIN does not use the same construction algorithm. Instead it uses the same number of virtual buckets, to make the comparison as fair as possible.

Table 30 shows that the bucketsize clearly influences the performance of the query for point location queries, most clearly demonstrated by the MultiPolygonZ which becomes unusable on larger bucketsizes and the speed of single types. The point location function finds a triangle in which the requested point lies.

The performance of the TINZ data structure is remarkable, being the fastest on small buckets and scaling well on larger bucket sizes. The backend SFCGAL is expected to be responsible, making use of fact that the TINZ

	l4		l5		l6		l7	
	cold	hot	cold	hot	cold	hot	cold	hot
MultiPolygonZ	91988	92130	16409	16409	4642	4489	2212	2072
Triangle array	131444	108356	20981	20813	6062	5849	3253	2887
Multistar	113	82	114	85	114	87	113	80
TINZ	1119	1110	612	580	387	397	336	335
pgTIN	283	60	55	43	42	42	42	42
PolygonZ*	450	366						
TriangleZ*	346	325						

Table 31: Comparison of query performance for each bucketlevel. Dataset demo.las. In ms. Horizontal query with 1358 intersections. * indicates single type where levels do not apply.

	l4		l5		l6		l7	
	cold	hot	cold	hot	cold	hot	cold	hot
MultiPolygonZ	273543	265546	43965	43771	21486	21382	7002	6275
Multistar	163	153	185	141	143	122	233	125

Table 32: Comparison of query performance for each bucketlevel. Dataset demo.las. In ms. Horizontal query on edges of buckets with 2043 intersections. * indicates single type where levels do not apply.

type is a connected patch. Once a triangle is found, no other triangles will match, while the normal MultiPolygonZ could have overlapping triangles, requiring checking all triangles. This does not explain the complete performance gain shown here, which requires further research. The queries used are described in Appendix E. For intersecting a 2D line with a TINZ or MULTIPOLYGONZ, both ST_Intersection and ST_3DIntersection return 2 dimensional results, without heights. SFCGAL is able to intersect 3D geometry, but that would require the intersection of a plane with the TIN.

The performance of the star based structures show that the size of the bucket does not influence the performance of walking much. Especially the pgTIN prototype is not influenced by the virtual bucketsize, which could be due to well placed virtual buckets. The multistar prototype shows the same pattern, but has a peak at the largest bucketlevel. The performance of the multistar and the pgTIN prototypes is comparable, but pgTIN is often faster.

The hot queries are faster than the cold queries overall, as the hot queries have the most effect on smaller bucketsizes and single type geometries, since they have the smallest sizes and thus fit most easily in cache.

The same pattern observer for point location holds true for a line intersection query. The bigger the buckets, the slower the queries become, as seen in Table 31. The star based structures that use walking instead of brute forcing outperform the other data structures for each bucketlevel. The pgTIN prototype is marginally faster than the Multistar prototype.

A diagonal intersection is a very optimal way of crossing buckets, as is a horizontal or vertical one on buckets which are mostly square since each bucket can be expected to be traversed only once. An almost horizontal intersection, exactly on the edge of buckets is a worse case scenario, alternately requesting points from two buckets.

	l4	l5	l6	l7	l8
# avg points	20490	6113	1319	338	88
# avg edgepoints	544	314	142	71	34
% points	2.7	5.1	10.8	21.0	39.2

Table 33: Bucket edge statistics: average number of points in buckets and average number of of points that have a pointer to another bucket in its star in a bucket, for different levels. Dataset demo.las with a total of 5245548 points.

type	l8	
	cold	hot
Multistar	5916	460
pgTIN	23878	313
Triangle array 1	8min	8min
Triangle array 2	8min	8min

Table 34: Comparison of query performance for bucketlevel 8 on 367M points. Dataset g37en2.laz. In ms. Diagonal intersection with 41291 intersections. 1 Uses MultiPolygonZ geometry, 2 uses TINZ geometry.

This is demonstrated in Table 32 for the multistar and simple feature types. The intersection crosses 50% more points, but the performance of the MultiPolygonZ type is more than halved in both hot and cold queries. The bounding boxes for each bucket of the MultiPolygonZ types overlap slightly at the edges, which doubles the number of buckets that need to be checked at those edges. The performance of the multistar prototype does not suffer, since the bounding boxes of points do not overlap, and multiple buckets are cached once used.

The performance gain for smaller buckets is not linear for each level. Indeed, the performance gain noticed becomes smaller with each increasing level. It is expected that this is caused by the overhead of processing points on the edge of buckets and the overhead of requests for new buckets. The increase of points on the edge of each bucket can be seen in Table 33. On smaller levels, even caching complete buckets becomes less useful, because new buckets are requested more often, and more points in proportion are on the edge of the bucket, adding to the requests for other buckets.

In Table 34 the performance of a diagonal query is presented on the massive g37en2.laz dataset. As discussed in Section 5.4, none of the SF data structures are available. Nevertheless, the triangle array can cast to such geometry, yet it is unusable as the intersection takes several minutes. The star based data structures are the only structures that still perform well on larger queries. The multistar prototype is for the first time noticeably faster on cold queries, but is slightly slower on hot queries. The ability to cache complete buckets during queries of the Multistar prototype is likely responsible for this, but PostgreSQL is not able to cache these buckets as efficiently as the stars in the pgTIN prototype for hot queries.

Larger queries with massive extents cannot not be benchmarked, because window queries for example request massive amounts of data from the database. Point location returns one triangle and range queries will return a line or a collection of triangles, roughly up to $\sqrt{n}$ points (the diagonal of the dataset), but only a window query can cover the complete dataset.

query	none	slope	aspect	local min	local max	degree
point location	52	52	52	53	53	52

Table 35: Performance of atomic functions. None is the timing of the basic query. Time in ms.

	rijswijk.las	demo.las	g37_en2.laz
degree	5.9996	5.99993	5.999997

Table 36: Average degree.

In such a case we encounter the problem of massive datasets again, as the database will run out of memory. PostgreSQL will report it is unable to allocate memory for the output buffer. This is a problem that cannot be solved from the inside, such as by the multistar extension, because PostgreSQL waits for all operations to complete before returning a result. Although the database response can be limited and offsetted, skipped rows are still computed. Instead the client should be aware of memory constraints and split up queries based on those constraints, but at the moment this is an open problem.

5.8 ATOMIC FUNCTIONS

None of the existing approaches have specific atomic functions for TINs. As implementation of the theoretical efficient approach, the Multistar data structure does have atomic functions implemented. If atomic TIN functions can be used inside the database, there is no need for extracting large subsections from the database and running those functions outside the database, saving both time and bandwidth. In Chapter 2 the following atomic functions were defined:

- Slope, the level of steepness of a triangle.
- Aspect, the direction of the slope of the triangle.
- Local minimum, whether a node is lower than its surrounding nodes.
- Local maximum, whether a node is higher than its surrounding nodes.
- Degree, the number of neighbours of node.

These functions are benchmarked to give an impression of the complexity of the operation, as well as the time it would take on top of a point location e.g. the slope of the triangle of the query point. The results in Table 35 show that the cost of calculating these atomic functions is a fraction of the spatial query used, often even undetectable.

The average degree of each node in a DT is expected to be 6 [Okabe et al., 2009], on which the theoretical sizes of TIN data structures are based in Chapter 2. This is confirmed by the degree function run on three different datasets as seen in Table 36. As the datasets increase in number of points, the average degree converges on the number 6.

5.9 SUMMARY

This chapter discussed the practical comparison of TIN data structures, based on the definition of an efficient approach. The performance of items that define an efficient approach are summarized here for the different data structures:

- *Loading time of the TIN, including construction* is defined by the complexity of the data structure. Bucketed data structures take longer to construct and load into the database than data structures that do not use buckets. On massive datasets the database fails to load the SF geometries, such as the TINZ. The SF TriangleZ would exceed the 100GB when loaded into a database, making it unfit for massive datasets. The performance of loading the data structure is of lesser importance as this step is only ran once.
- *Storage size of data structures* present a clear distinction between data structures using buckets and those that do not, data structure using buckets being on average several times smaller. The Triangle Array is the smallest data structure in these experiments, followed by the Multistar data structure. Results are influenced by PostgreSQL TOAST compression and row overhead. The SF geometries are the largest data structures.
- *Storage size of index* is defined by the use of buckets. A large proportion of the storage space taken by non bucketed data structure is used by the (spatial) index. These indexes are often bigger than a complete data structure that does use buckets. On massive datasets the same pattern is visible, where the index of the pgTIN data structure alone is larger than the Triangle array data structure.
- *Time of spatial queries* is influenced by the size of buckets for data structures using brute force methods. These data structures, such as the Triangle Array and SF TINZ become unusable when larger buckets are used, while the performance of data structures that use walking are largely unaffected. On massive datasets the use of buckets in the Multistar results in faster cold queries, but in slower hot queries compared to the non bucketed approach of pgTIN.
- *Time of atomic functions* is negligible when applied during spatial queries. The operations required are small and fast, often able to use the cached data structure that is used for the spatial query itself.

The experiments validate the criteria for an efficient approach for storing massive TINs in a DBMS. The Multistar prototype performs well each item tested, except for the construction and loading. On massive datasets only the pgTIN and Multistar prototype perform well in spatial queries, demonstrating the need for topological relationships. The need for buckets is demonstrated by the size of the pgTIN index, which is almost as large as the complete Multistar structure.

6 CONCLUSION AND DISCUSSION

This thesis dealt with finding an efficient approach to store massive TINs in a DBMS. The criteria found in theory for such an efficient approach has been realized in the Multistar data structure, which has been implemented in a database prototype. Conclusions on the efficient approach for storing massive TINs are presented in Section 6.1. A discussion on both the criteria as well as the prototype is given in Section 6.2. Finally, in Section 6.3 recommendations for future work are given on storing massive TINs in DBMS.

6.1 CONCLUSION

The main objective of this thesis was to compare different approaches to store massive TINs in a DBMS - both theoretically and practically - thereby answering the question: *What are efficient approaches to store massive TINs in a DBMS?*. The main problem is to combine a TIN data structure with solutions for massive datasets. By both theoretical and practical comparisons this thesis has shown that an efficient approach for storing massive TINs in a DBMS includes the following criteria.

- *Nodes* of a TIN are explicitly stored only once, preventing duplicate coordinates. TIN data structures using nodes are also the smallest data structures possible. Node storage thus minimizes the disk space required to store TINs. This improves the handling of massive datasets. By storing nodes a pointcloud is stored as well, using the disk space more efficiently.
- *Atomic functions* such as those that calculate slope and aspect are implemented. Other functions include the degree of a node, local minimum and maximum as well as interpolation algorithms. Many applications of TINs, such as a DTM, drainage network, analysis of pointclouds are enabled by these functions. By implementing atomic functions inside the database, the transfer of massive amounts of data between database and client is prevented, saving both time and bandwith.
- *Topological relationships* of a TIN are explicitly stored. Although this increases the size of the data structure, topology can be used for spatial operations as well as to access a TIN. Atomic functions such as local minimum require knowledge about neighbours of node, which can be accessed in direct time if topological relationships are stored. Topology can also be used for *walking*. This solves point location in $O(\sqrt{n})$ time by traversing the TIN instead of brute forcing in $O(n)$ time. Similar algorithms such as *marching* are used to solve other spatial queries. Spatial indexes on simplexes of a TIN are not required for these algorithms.
- *Buckets* are used to split the TIN into non-massive sections, which can be processed and analysed one at a time. Topological relationships

should be able to point to neighbours in other buckets, making it still possible to traverse across buckets. A spatial index is only used on the spatial extent of the bucket, as a spatial index on each individual node, edge or triangle would become too large on massive datasets. This enables quick selections of buckets in a region of interest. These buckets can be traversed by topological algorithms, since these algorithms only require a starting point.

At the moment only two TIN data structures are implemented in spatial databases. One is the SDO_TIN type used in the Oracle Spatial and Graph component of the Oracle Database. The other is the SF TIN type described in the latest SF standard by OGC, implemented by databases such as PostgreSQL with the PostGIS extension.

- *SDO_TIN* encodes the TIN in a data structure that can be spatially clipped and cast to Oracle Geometry. The data structure is similar to the triangle array structure, storing nodes and triangles pointing to those nodes. The data structure also uses buckets, which results in triangles that store pointers to points in specific buckets. However, atomic functions are not available, nor are topological relationships stored or used. A SDO_TIN object can be generated from a pointcloud stored in the same Oracle Database, but this only works on small datasets. The buckets generated are thin vertical sections of the complete datasets, resulting in inefficient access.

- *Simple Feature (Access) (SF)* encodes the TIN as a patch of triangles - in which each triangle is stored as a ring of four coordinates. Thus the coordinates of one node are duplicated. Further duplications exist because each node is present in almost six triangles on average, each triangle storing the coordinates of the node. Buckets can be used, but each bucket exists on its own. Topological relationships are not stored. In the PostGIS implementation of the SF TIN type, the SFCGAL backend is required to spatially operate on the TIN, but atomic functions described are lacking.

A practical comparison between TIN types is done in PostgreSQL. In order to compare these types in the same database, a data structure, named Triangle Array, similar to the Oracle SDO_TIN structure has been implemented in PostgreSQL. The pgTIN prototype by Ledoux [2013] using a star based data structure has also been used. A data structure called the Multistar, based on the criteria discussed, is one of the main works in this thesis. It has been developed and implemented in PostgreSQL as to validate the criteria for an efficient approach of storing massive TINs in DBMS.

- *pgTIN* is a prototype by Ledoux [2013] storing massive TINs with a star based approach, using topological relationships to solve spatial queries. By storing only nodes and pointers to its neighbouring nodes the storage size is very small. Several atomic functions are implemented, but the data structure does not use buckets. Individual nodes are indexed with a binary tree.

- *Multistar* builds upon the pgTIN prototype by implementing buckets in both the data structure as the topological functions as walking and marching. Several more atomic functions are implemented. Multiple nodes and stars are stored in each database row, omitting identifiers

for each node, but requiring an offset array. An index is only used on the spatial extent of each bucket.

The construction of the TIN from one tile of the AHN2 dataset is done by the *streaming* algorithm of lastools. Several Python scripts have been implemented to transform the lastools output into the different data structures. Existing approaches and the prototypes have been compared on requirements for an efficient approach, which is the performance of the following items:

- *Loading time of the TIN.* The time taken to construct a TIN from a pointcloud and transfer the data structure into the database is very much dependent on the complexity of the data structure. The Multistar together with the Triangle Array takes the longest time to load, while the pgTIN structure and single SF types are much faster. However, on a massive datasets the SF fail to load, leaving only the three prototypes of which the pgTIN is the fastest.
- *Storage size of data structure.* Of all the data structures the Triangle Array is the smallest, followed closely by the Multistar data structure. The SF TIN type is twice as big as the Triangle Array. However, this is the compressed size, uncompressed it would be five times as large. Row overhead in PostgreSQL causes the pgTIN type to be the largest data structure, but the data structure itself is as large as the Multistar.
- *Storage size of index.* Indexes on individual nodes, edges or triangles become very large on massive datasets. Even non spatial indexes such as a binary tree become several gigabytes on massive datasets. Thus, buckets are required for keeping the spatial index small.
- *Performance of spatial queries.* The performance of spatial queries is dependent on the size of buckets used, where smaller buckets yield better performance. The prototypes using topological relationships scale much better with the size of the buckets and outperform any brute forcing method. Indeed, spatial queries using PostGIS functions ran for hours on massive datasets, compared to the seconds needed by the pgTIN and Multistar prototype.

The pgTIN and Multistar prototypes outperform other data structures in both size and query performance. When the data structures are benchmarked on massive datasets these are the only two data structures that remain viable. However, the index of pgTIN is almost as large as the complete Multistar data structure, showing the need for buckets. The criteria presented for an efficient approach for storing massive TINs in a database are validated in the Multistar prototype. Currently implemented data structures for TINs such as the SF TIN type and the SDO_TIN type are unfit for storing massive TINs in a DBMS.

6.2 DISCUSSION

An efficient approach to storing massive TINs has been presented in theory as well as in practice. However, one should not conclude that the criteria or the implementation of those criteria in the Multistar prototype are the only ways to an efficient approach. Indeed, many prototypes can be build

on the criteria presented and the Multistar prototype is not without drawbacks. The criteria presented are valid for the scope of this thesis, but can be changed for other applications. Several other points should be made in the context of this thesis:

PostgreSQL specifics such as TOAST and the overhead for each row obfuscated the storage sizes. Since TOAST compression is on by default and seems to have little to no performance impact, the results include TOASTed data structures. Yet, this does exaggerate differences between data structures. The difference between the Multistar and pgTIN prototypes is completely due to row overhead and the size of the indexes used. The data structure itself is identical.

One of the criteria for data structures has been defined as having atomic TIN functions, which were lacking in all existing approaches. One could argue that these functions are so specific they do not belong at the database level, but in the middle ware or client. Nevertheless, for an efficient approach, such atomic functions should be able to make use of the topological data structure inside the database. Although simple analysis such as aspect and slope are lacking for SF, they are available for rasters in PostGIS, showing that these functions are common.

The multistar prototype is a method for constructing, loading, storing and accessing massive TINs in a DBMS, solving the problem of the massive dataset in each aspect. However, requesting a massive section of the dataset in the database for output will still result in the database being unable to locate enough memory to comply with the request, as stated in Section 5.7. This is an open problem, which cannot be solved inside the database - and is thereby out of reach for the multistar prototype.

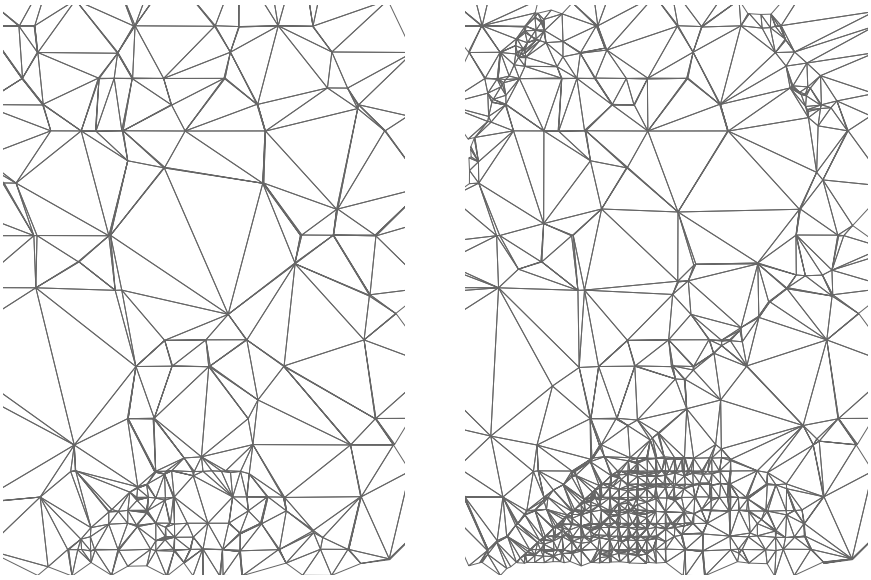
6.3 FUTURE WORK

The Multistar prototype, as well as the Triangle Array prototype are promising. More research should be done on this topic, which expands the scope set for the research in this thesis. Future work would include updates to the TIN data structures and implementing the triangle array+ data structure in a prototype. These topics and other future work are presented here, split in sections of the expected time and effort it will take to research them.

6.3.1 In a month

- *Thinning.* Explicitly stored topological relationships lend itself for thinning in more ways than thinning every nth point or random thinning. A prototype function (see Appendix E) has been implemented that uses the ST_DelaunayTriangles to retriangulate a thinned subset of the multistar prototype, not unlike the Implicit TIN (Section 4.7.1). The selection criteria is the sum of the height differences of a point with its neighbours. Two examples can be seen in Figure 35.

- *Multiresolution TIN.* Thinning of the multistar data structure results in new stars for selected relevant points. Instead of only retriangulating on the fly, the result could be decomposed into a star based structure and stored. Thus, several columns holding these stars could be added to the data structure, one for each level. This would also require multiple offset arrays, mostly empty, but would reuse the node coordinates



(a) (b)

Figure 35: Thinned datasets of demo.las (a) A thinned and retriangulated dataset using a filter of 5m total difference (b) A thinned and retriangulated dataset using a filter of 3m total difference.

stored. The offset arrays required for the multistar prototype are not required for the triangle array with explicit topological relationships stored.

- *Dynamic tables.* The current implementations of the multistar and triangle array prototypes have hardcoded tablenames in their source code, for looking up data outside the requested bucket. To make benchmarking and using multiple tables easier, the table name should be used as an SQL input. Care should be taken to prevent SQL injection, as the input code is executed again.

6.3.2 In a year

- *3D* Ledoux and Meijers [2013] have shown that it is possible to store 3D topography in database with the star based approach. When eventually moving to 3D instead of 2.5D the buckets can be defined not by a 2D regular tessellation, but a 3D one, effectively storing buckets as voxels in an octree. Both thinning and multiresolution that have been proposed in the previous section also apply to 3D. However, the walking algorithms used in the multistar prototype will not work anymore, as they are based on 2D orientation or intersection tests.

- *Compression* PostgreSQL uses TOAST to compress large rows, but it applies a generic compression algorithm. By using specific compression algorithms for point coordinates, the size of the massive TIN will be reduced, touching upon the main problem these massive datasets present. Other tricks, such as discussed in Section 4.5 are also possible. This research would not need to start from scratch as the open source pointcloud extension for PostgreSQL already uses several compression

techniques [1]. One of these techniques is dimensional compression, which compresses at between 3:1 and 5:1 efficiency on samples with a high spatial coherence, as buckets have.

- *Updates* Updating the TIN in the database is outside the scope of this thesis, but is expected of (spatial) database structures. As discussed in Section 4.9 because of the variable length of stars and thus of the multistar, updating becomes slower. Indeed, the advantage of an insertion of one point requiring only three updates to stars Ledoux [2013] is somewhat negated by the multistar. For an insertion of a point, the expensive operation is to update the offset array, since every offset from the inserted point is shifted. On edges of buckets, this can become more expensive, updating several buckets at a time.

  Local updates not violating the Delaunay criteria should also be researched, which requires a function to compute a DT inside the database. For PostgreSQL such a function is present, but it should be adapted in such a way that each update results in incrementally updating the TIN until the TIN is a valid DT again. This would result in several recalculations of the offset array in a bucket. The triangle array with topological relationships is a better alternative in this respect, since the sizes of this data structure are fixed, not requiring an offset arrays.

- *Parallelization.* Some operations can be run in parallel at the same time, thereby increasing performance. It is, for example, possible to process independent subsections of dataset several at a time, instead of processing them one by one. The simple features approach, which cannot refer to other buckets (islands as discussed in Section 4.3.1) is fit for such an approach. A parallel client is used in a paper by van Oosterom et al. [2015] for accessing pointclouds.

  The multistar algorithm is not very fit for parallelized queries as discussed in Section 4.3.3. Indeed, its speed is derived from not having to calculate a starting point for each bucket, since it is referenced from the previous processed bucket. However, a combination would be possible by dividing a line intersection into two lines, starting two walking algorithms to the common center of the line from the endpoints of the line. It is only expected to be efficient when the time taken for the main function, such as marching for a line intersection greatly outweighs the time required to find a starting point for marching.

- *Bucket shapes.* Currently a quadtree is used to determine the shape of buckets, resulting in differently sized rectangular buckets. Especially on very thin or very wide datasets, this could decrease performance, which could be solved by implementing square buckets. Other regular tessellations are possible, but it is unknown how hexagonal buckets would influence performance.

- *Hierarchies* Future work such as thinning, multiresolution TINs and bucket shapes can be combined with concepts as hierarchical subdivided TINs [De Floriani and Puppo, 1995; Bertolotto et al., 1995]. An overview of such simplification methods is given by Heckbert and Garland [1997], but research is needed for discovering possibilities of applying these methods in a database.

1 explained at its github site

6.3.3 With a team of developers

- *Loader performance* The current construction and loading algorithms are written in Python. By implementing these algorithms in a programming language such as C, loading and construction times can be sped up. It would also lower the memory footprint on these algorithms. The code by lastools [2] could be used to implement a more elegant solution to combine the output of the streaming algorithm. It would also be preferable to implement an open source streaming algorithm or similar construction methods that are able to handle massive TINs, as the streaming solution used in this thesis is neither open source nor free.

- *Mature prototype* Although the current prototype works and is publicly available, it can be become more mature in its functioning an programming. Future works includes the handling of dynamic tablenames i.e. not hardcoding them in the prototype. However, this shows a flaw in accessing the multistar prototype in a relational database. Other problems are the many fields needed for each function in the multistar prototype. A solution would be storing the table name in each multistar, but this would be a non normalized approach. Oracle has solved this exact problem by implementing the SDO_TIN object, with all the relevant metadata stored as discussed in Section 3.2.2. When implemented in PostgreSQL, Multistar buckets are only accessed and retrieved by interfacing with this object stored in another table.

- *Triangle array+* Although the Multistar prototype has been shown to work and to be both fast and small, it has certain drawbacks. A good alternative, based on the same criteria for an efficient approach described in this thesis, would be storing the triangle array data structure with explicitly stored topological relationships. It is expected to have a larger size on disk, but would improve upon aspects such as updating and accessing.

2 found at https://github.com/LAStools/LAStools/

BIBLIOGRAPHY

Agarwal, P., Arge, L., and Danner, A. (2006). From point cloud to grid DEM: A scalable approach. In Riedl, A., Kainz, W., and Elmes, G., editors, *Progress in Spatial Data Handling*, pages 771–788. Springer Berlin Heidelberg.

Agarwal, P., De Berg, M., Bose, P., Dobrint, K., Van Kreveld, M., Overmars, M., De Groot, M., Roos, T., Snoeyink, J., and Yu, S. (1996). The complexity of rivers in triangulated terrains. In *Proceedings of the Canadian Conference on Computational Geometry 8th, CCCG'96, August 12-15, Carleton University, Ottawa*, pages 325–330. Citeseer.

Agarwal, P. K., Arge, L., and Yi, K. (2005). I/o-efficient construction of constrained delaunay triangulations. In *Algorithms–ESA 2005*, pages 355–366. Springer.

Baumgart, B. G. (1975). A polyhedron representation for computer vision. In *Proceedings of the May 19-22, 1975, national computer conference and exposition*, pages 589–596. ACM.

Berg de, M., Van Kreveld, M., Overmars, M., and Schwarzkopf, O. C. (2000). *Computational geometry*. Springer.

Bertolotto, M., De Floriani, L., and Marzano, P. (1995). Pyramidal simplicial complexes. In *Proceedings of the Third ACM Symposium on Solid Modeling and Applications*, SMA '95, pages 153–162, New York, NY, USA. ACM.

Blandford, D. K., Blelloch, G. E., Cardoze, D. E., and Kadow, C. (2005). Compact representations of simplicial meshes in two and three dimensions. *International journal of computational geometry & applications*, 15(01):3–24.

Burrough, P. A. and McDonnell, R. (1998). *Principles of geographical information systems*, volume 333. Oxford university press Oxford.

Chen, M.-B., Chuang, T.-R., and Wu, J.-J. (2006). Parallel divide-and-conquer scheme for 2d delaunay triangulation. *Concurrency and Computation: Practice and Experience*, 18(12):1595–1612.

De Floriani, L., Facinoli, M., Magillo, P., and Dimitri, D. (2008). A hierarchical spatial index for triangulated surfaces. In *GRAPP*, pages 86–91.

De Floriani, L. and Puppo, E. (1995). Hierarchical triangulation for multiresolution surface description. *ACM Trans. Graph.*, 14(4):363–411.

Devillers, O., Pion, S., and Teillaud, M. (2002). Walking in a triangulation. *International Journal of Foundations of Computer Science*, 13(02):181–199.

Devroye, L., Lemaire, C., and Moreau, J.-M. (2004). Expected time analysis for delaunay point location. *Computational Geometry*, 29(2):61 – 89.

Djinevski, L., Stojanova, S., and Trajanov, D. (2014). Optimizing durkins propagation model based on TIN terrain structures. In Trajkovik, V. and Anastas, M., editors, *ICT Innovations 2013*, volume 231 of *Advances in Intelligent Systems and Computing*, pages 263–272. Springer International Publishing.

Douglas, K. and Douglas, S. (2003). *PostgreSQL: a comprehensive guide to building, programming, and administering PostgreSQL databases*. SAMS publishing.

Edelsbrunner, H. (2001). *Geometry and topology for mesh generation*. Cambridge University Press.

Egenhofer, M. J. and Franzosa, R. D. (1991). Point-set topological spatial relations. *International Journal of Geographical Information System*, 5(2):161–174.

Elmasri, R. and Navathe, S. B. (2006). *Fundamentals of Database Systems (5th Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Fritsch, D. (1996). Three-dimensional geographic information systems-status and prospects. *International Archives of Photogrammetry and Remote Sensing*, 31:215–221.

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

Heckbert, P. S. and Garland, M. (1997). Survey of polygonal surface simplification algorithms. Technical report, DTIC Document.

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

Isenburg, M., Liu, Y., Shewchuk, J. R., 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.

Jones, C. B., Kidner, D. B., and Ware, J. M. (1994). The implicit triangulated irregular network and multiscale spatial databases. *The Computer Journal*, 37(1):43–57.

Kettner, L. (1999). Using generic programming for designing a data structure for polyhedral surfaces. *Computational Geometry*, 13(1):65–90.

Kidner, D. B., Ware, J. M., Sparkes, A. J., and Jones, C. B. (2000). Multiscale terrain and topographic modelling with the implicit TIN. *Transactions in GIS*, 4(4):361–378.

Kim, S., Cho, M., and Cho, H.-g. (1999). A geometric compression algorithm for massive terrain data using delaunay triangulation. In *Proc. of WSCG '99*, pages 124–131. WSCG.

Kumler, M. P. (1994). An intensive comparison of triangulated irregular networks (tins) and digital elevation models (dems). *Cartographica: The International Journal for Geographic Information and Geovisualization*, 31(2):1–99.

Ledoux, H. (2013). Storing and analysing massive tins in a DBMS with a star-based data structure. Unpublished report.

Ledoux, H. and Meijers, M. (2013). A star-based data structure to store efficiently 3d topography in a database. *Geo-spatial Information Science*, 16(4):256–266.

Lee, J. (1989). A drop heuristic conversion method for extracting irregular networks for digital elevation models. In *Proceedings GIS/LIS '89*, pages 30–39, Orlando, USA.

Li, Z., Zhu, C., and Gold, C. (2010). *Digital terrain modeling: principles and methodology*. CRC press.

Lyon, J. G. (2003). *GIS for Water Resource and Watershed Management*. CRC Press.

Moore, I. D., Grayson, R., and Ladson, A. (1991). Digital terrain modelling: a review of hydrological, geomorphological, and biological applications. *Hydrological processes*, 5(1):3–30.

Mücke, E. P., Saias, I., and Zhu, B. (1999). Fast randomized point location without preprocessing in two-and three-dimensional delaunay triangulations. *Computational Geometry*, 12(1):63–83.

OGC (2006). Opengis implementation specification for geographic information - simple feature access - part 1: Common architecture. Technical report, OGC. version 06-103r4.

Okabe, A., Boots, B., Sugihara, K., and Chiu, S. N. (2009). *Spatial tessellations: concepts and applications of Voronoi diagrams*, volume 501. John Wiley & Sons.

Oracle (2015). Sdo_TIN_pkg package (tins). https://docs.oracle.com/cd/ Accessed on 22 December 2014.

Oude Elberink, S., Stoter, J., Ledoux, H., and Commandeur, T. (2013). Generation and dissemination of a national virtual 3D city and landscape model for the Netherlands. *Photogrammetric Engineering and Remote Sensing*, 79(2):147–158.

Peckham, R. J. and Gyozo, J. (2007). *Digital terrain modelling, Development and Applications in a Policy Support Environment*. Lecture Notes in Geoinformation and Cartography. Springer.

Penninga, F. (2004). Oracle 10g topology. Technical report, TU Delft.

Peucker, T. K., Fowler, R. J., Little, J. J., and Mark, D. M. (1978). The triangulated irregular network. In *Amer. Soc. Photogrammetry Proc. Digital Terrain Models Symposium*, volume 516, page 532.

Pigeon, S. (2001). *Contributionsa la compression de données*. PhD thesis, Ph. d. thesis, Université de Montréal, Montréal.

Quak, W., Stoter, J., and Tijssen, T. (2003). Topology in spatial dbmss. In *The 3rd international symposium on digital earth, Brno, September, 2003*.

Ravada, S., Kazar, B., and Kothuri, R. (2009). Query processing in 3d spatial databases: Experiences with oracle spatial 11g. In Lee, J. and Zlatanova, S., editors, *3D Geo-Information Sciences*, Lecture Notes in Geoinformation and Cartography, pages 153–173. Springer Berlin Heidelberg.

Sabo, N., Beaulieu, A; Bélanger, D., Belzile, Y., and Piché, B. (2014). The geohashtree: a multi-resolution data structure for the management of point clouds. serial Technical Note 4E, Geomatics Canada.

Sagan, H. (1994). *Space-filling curves*, volume 18. Springer-Verlag New York.

Samet, H. (2006). *Foundations of multidimensional and metric data structures*. Morgan Kaufmann.

Santilli, S. (2011). Topology with postgis 2.0. Presentation.

Shewchuk, J. R. (1996). Triangle: Engineering a 2d quality mesh generator and delaunay triangulator. In *Applied computational geometry towards geometric engineering*, pages 203–222. Springer.

Soukal, R., Málková, M., and Kolingerová, I. (2012). Walking algorithms for point location in TIN models. *Computational Geosciences*, 16(4):853–869.

Taubin, G. and Rossignac, J. (1998). Geometric compression through topological surgery. *ACM Transactions on Graphics (TOG)*, 17(2):84–115.

van Kreveld, M. (1997). Digital elevation models and TIN algorithms. In *Algorithmic foundations of geographic information systems*, pages 37–78. Springer.

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. *Computers & Graphics*, 49(0):92 – 125.

van Oosterom, P., Quak, W., and Tijssen, T. (2005). About invalid, valid and clean polygons. In *Developments In Spatial Data Handling*, pages 1–16. Springer.

van Oosterom, P., Stoter, J., Quak, W., and Zlatanova, S. (2002). The balance between geometry and topology. In *Advances in Spatial Data Handling*, pages 209–224. Springer.

van Oosterom, P. and Vijlbrief, T. (1996). The spatial location code. In *Proceedings of the 7th International Symposium on Spatial Data Handling, Delft, The Netherlands*.

Vitter, J. S. (2001). External memory algorithms and data structures: Dealing with massive data. *ACM Computing surveys (CsUR)*, 33(2):209–271.

Weibel, R. and Heller, M. (1993). *Digital terrain modelling*. Oxford University Press.

Wilson, J. P. and Gallant, J. C. (2000). *Terrain analysis: principles and applications*. John Wiley & Sons.

Worboys, M. F. and Duckham, M. (2004). *GIS: a computing perspective*. CRC press.

Yang, J. and Huang, X. (2014). A hybrid spatial index for massive point cloud data management and visualization. *Transactions in GIS*, 18:97–108.

Yu, S., Van Kreveld, M., and Snoeyink, J. (1996). Drainage queries in TINs: from local to global and back again. In *Proc. 7th Int. Symp. on Spatial Data Handling, pages A*, volume 13. Citeseer.

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

Zlatanova, S., Rahman, A. A., and Shi, W. (2004). Topological models and frameworks for 3d spatial objects. *Computers & Geosciences*, 30(4):419–428.

A REFLECTION

An efficient approach for storing massive TINs in a DBMS is proposed in this thesis. The process took 8 months, as initially planned. The planning proposed included learning the programming language C, as required for implementing a fast extensions in PostgreSQL. However, extending PostgreSQL and in general code by others such as the prototype by Ledoux [2013] is not easily done with a newly acquired language. Indeed, this required several more weeks, in order to understand the structure and logic behind database extensions. This resulted in a tight schedule at the end of the thesis. I am however quite happy with the result, learning a new programming language was one of my main motivations and I would gladly do it again. Indeed, the long list of future work show my interest in this topic. This has been a disadvantage at times as well, when research is done outside of the scope of the thesis.

The research initially was focused on expanding the PostgreSQL pointcloud extension with a star based structure, but this proved to be difficult as the pointcloud extension uses fixed size. Storing variable length stars would require rewriting most of the existing code, which is why stand alone prototypes has been developed. This led to a change in focus on a general efficient approach in theory, of which the multistar approach is a possible implementation, but certainly not the only one. The results show the validity of the requirements for an efficient approach proposed in this thesis, as well as the shortcomings of current approaches. However, this does certainly not exclude other requirements and implementations from being valid as well.

The field of Geomatics is concerned with the analysis, acquisition, management and visualisastion of geographic data. Storing geographic data such as TINs in databases as well as analysing TINs is a clear part of the research undertaken within the field of Geomatics. Indeed, much of the knowledge applied was gained in the core courses of the Master of Geomatics.

The methodology used in this thesis is well known in the Master Geomatics. The approach proposed in this thesis, as well as the prototype implemented, requires both topology and geometry linked in a DBMS which is an open area of research in the field of Geomatics [Zlatanova et al., 2004]. TINs have many applications, one being terrain modeling as in the TOP10 NL 3D dataset [Oude Elberink et al., 2013]. Practical comparisons on spatial performance in databases, called benchmarking is done often, most recently during this thesis [van Oosterom et al., 2015]. The prototype developed reuses elements from the implementation by [Ledoux, 2013], which has also been implemented in 3D by Ledoux and Meijers [2013], showing the relation between this thesis and the other research undertaken in the Geomatics.

Many practical applications of the massive point cloud data require easy access and analysis of such point clouds. These operations often require an intermediary TIN. Accessible database storage of such massive TINs enables easier access, analysis and distribution for those who work with point cloud data. Pointcloud and TIN analysis used in a scientific setting is very relevant

when used for flood modeling. Indeed, flood prevention is relevant for the general public in the Netherlands.

The prototypes developed can still be improved, which is why the code can be found on GitHub, as it is explained in detail in the other appendices. It is expected that a Geomatics PhD research is undertaken on the same topic, using my thesis and prototypes as a starting point. This again underlines the scientific relevance of this topic and thesis as well as the relation of this thesis with the research undertaken in the field of Geomatics.

B CONSTRUCTION AND LOADING TOOLS

Construction and loading of TIN structures into the database is done by lastools which is piped to custom Python tools as described in Section 5.1. A distinction is made between single type loaders, which output only one triangle or star at the time, while multi type loaders output collections of geometries in buckets, such as MultiPolygons. All these loaders are available and further document at https://github.com/evetion/thesis_tools.

A typical way of constructing the the multistar data structure and loading it into the database is demonstrated in Listing 1. Both the `spfinalize` and `spdelaunay2d` are lastools programs that use streaming algorithms. The first run uses a Python script `qtfin.py` that collects statistics about the number of points for each bucket. This information is reused by all the multi type data structures to create the buckets in the database. The second step uses a Python postgresql client to load the data into the database. These script could be sped up by using COPY instead of INSERT statements.

Listing 1: Loading of multistar type.

```
/* Construct quadtree buckets information */
spfinalize -i rijswijk.laz -ilas -o -v -level 5 -ospb |
    spdelaunay2d -ispb -osmb | python qtfin.py 5
/* Create data structure based on the previous run */
spfinalize -i rijswijk.laz -ilas -o -v -level 5 -ospb |
    spdelaunay2d -ispb -osmb | python multistar.py 5
```

Single type geometries such as TriangleZ but also pgTIN, do not require the construction step of a quadtree and output their information to a named pipe as seen in Listing 2.

Listing 2: Loading of single type.

```
/* Create pgTIN data structure */
spfinalize -i rijswijk.laz -ilas -o -v -level 5 -ospb |
    spdelaunay2d -ispb -osmb | python pgtin.py > pipe

/* Create TriangleZ data structure */
spfinalize -i rijswijk.laz -ilas -o -v -level 5 -ospb |
    spdelaunay2d -ispb -osmb | python simplefeatures.py > pipe
```

From this named pipe the data can be imported in the database after creating the relevant table as seen in Listing 3 and Listing 4.

Listing 3: pgTIN table creation and loading.

```
CREATE TABLE points (id  int, x  double precision, y  double
    precision, z  double precision, star integer[]);
COPY points from 'pipe';
ALTER table points add primary key (id);
```

Listing 4: TriangleZ table creation and loading.

```
CREATE TABLE trianglez();
SELECT AddGeometryColumn('public', 'trianglez', 'geom', -1, '
    TRIANGLEZ', 3);
COPY trianglez FROM 'pipe';
CREATE INDEX idx_trianglez USING gist (geom);
```

When constructing and loading massive datasets or using a large number of buckets, the bucket id and point id cannot be paired into an unsigned 32bit integer anymore. In order to switch to unsigned 64 bit integers, three statements should be changed in typical loader scripts:

- The SQL statements for creating the table should state bigint
- The writebin function should use the 'Q' structure instead of 'I'
- The stitch function needs to be changed to 64bit integers.

Relevant comments on changing these things are present in the python files.

C MULTISTAR

The Multistar data structure is described extensively in Chapter 4 and in Section 5.3. In here the different functions are described. The source code can be found in the Github repository at https://github.com/evetion/multistar. The prototype is currently divided into two C source files, one which decodes the data structure and one which contains the spatial functions. At the moment the tablename is hardcoded in the source code, the prototype thus requires recompiling in order to use different tables.

The Listing 5 the table structure and the relevant indexes and views are created. This is normally done by the Python script in the construction procedure as seen in Listing 6. An index is created on both the unique id as the geometry. A view is created for use in QGIS to show the extents of the buckets.

Listing 5: Multistar table creation.

```
CREATE TABLE multistar (id int, bbox box3d, offsets int[], points
     bytea, stars bytea);

ALTER TABLE multistar ADD PRIMARY KEY (id);
CREATE INDEX idx_multistar ON {} using GIST(ST_FORCE_3DZ(bbox));
CREATE OR REPLACE VIEW multistar_all AS SELECT id,
                    st_force_3dz(bbox::geometry) AS st_force_3dz
                    FROM multistar;
```

Listing 6: Loading of multi type.

```
/* Construct quadtree buckets information */
spfinalize -i rijswijk.laz -ilas -o -v -level 5 -ospb |
    spdelaunay2d -ispb -osmb | python qtfin.py 5
/* Create data structure based on the previous run */
spfinalize -i rijswijk.laz -ilas -o -v -level 5 -ospb |
    spdelaunay2d -ispb -osmb | python multistar.py 5
```

Three spatial queries are adapted from the pgTIN prototype to work with the multistar data structure as seen in Listing 7

Listing 7: Spatial queries.

```
/* Point location, returning TriangleZ */
CREATE OR REPLACE FUNCTION pl(double precision, double precision,
     integer)
  RETURNS bytea AS
'/usr/lib/postgresql/9.3/lib/multistar.so', 'pl'
  LANGUAGE c IMMUTABLE STRICT

/* Profile, returning LineStringZ */
```

```
CREATE OR REPLACE FUNCTION public.profile(double precision,
    double precision, double precision, double precision, integer
    )
  RETURNS bytea AS
'/usr/lib/postgresql/9.3/lib/multistar.so', '
    profile_count_intersections'
  LANGUAGE c IMMUTABLE STRICT

/* Range query returning true if all points are found */
CREATE OR REPLACE FUNCTION range_query(double precision, double
    precision, double precision, double precision, integer,
    integer)
    RETURNS boolean
    AS '/usr/lib/postgresql/9.3/lib/multistar.so', 'range_query'
    LANGUAGE C STRICT IMMUTABLE;
```

Spatial functions cache database rows, which can be requested internally in PostgreSQL, calling one of the functions in Listing 8. The gettin_convex is for example used in Algorithm 4.1.

Listing 8: Explode Multistar to stars.

```
/* Dump one multistar row as a set with all the points stored.
    Input is offsets, points and stars */
CREATE OR REPLACE FUNCTION gettin(IN integer[], IN bytea, IN
    bytea,
        OUT id integer, OUT x double precision, OUT y double
            precision, OUT z double precision, OUT star integer
            [])
  RETURNS SETOF record
  AS '/usr/lib/postgresql/9.3/lib/multistar.so', 'gettin'
  LANGUAGE c IMMUTABLE STRICT  ;

/* Dump for one multistar row only the points on the convex hull,
     does return NULL rows. Input is offsets, points and stars */
  CREATE OR REPLACE FUNCTION gettin_convex(IN integer[], IN bytea
      , IN bytea,
        OUT id integer, OUT x double precision, OUT y double
            precision, OUT z double precision, OUT star integer
            [])
  RETURNS SETOF record
  AS '/usr/lib/postgresql/9.3/lib/multistar.so', 'gettin_convex'
  LANGUAGE c IMMUTABLE STRICT  ;
```

The decoder functions are exposed to PostgreSQL and can be used to create a view on the pointcloud stored as seen in Listing 9. A simple simplifying algorithm has been implemented, as discussed in Section 4.7.1, which returns significant points. These points can again be triangulated into a TIN as seen in Listing 10. Other useful functions such as stating the complete number of points stored, or the selection of buckets on the convex hull do not require an extensions, but are retrieved in pure SQL as seen in Listing 11.

Listing 9: View creation for PostGIS point geometry.

```
/* A view on all the points in one bucket from the gettin
    function */
```

```
CREATE OR REPLACE VIEW points AS SELECT id, ST_MakePoint(x,y,z
    ,28992) FROM ( SELECT (gettin(tree,points,stars)).* FROM
    multistar_l6 WHERE id =117  ) as f
```

Listing 10: Simplify as shown in the future work (Section 6.3).

```
SELECT ST_DELAUNAYTRIANGLES(ST_UNION(geom),0)
FROM (SELECT ST_MAKEPOINT(x,y,z) as geom FROM
        (SELECT (simplify(tree,points,stars)).* FROM
            multistar_l6_adaptive) as f
WHERE @dif > 5) as x
```

Listing 11: Meta functions not requiring the PostgreSQL extension.

```
/* Total number of points in table */
SELECT SUM(@tree[1]) FROM multistar_l4
/* Bucket ids that contain points that are on the convex hull */
SELECT id FROM multistar_l4 WHERE tree[1] < 0
```

D TRIANGLE ARRAY

The Triangle Array implementation is described in Section 5.3 to mimic the Oracle SDO_TIN structure. In here the different functions are described. The source code can be found in the Github repository at https://github.com/evetion/trianglearray.

The Listing 12 the table structure and the relevant indexes and views are created. This is normally done by the Python script in the construction procedere as seen in Listing 15.

Listing 12: Triangle array creation.

```
CREATE TABLE triangle_array (id int, bbox box3d, nump int, numt
    int, points bytea, triangles bytea);

ALTER TABLE triangle_array ADD PRIMARY KEY (id);
CREATE INDEX idx_triangle_array ON {} using GIST(ST_FORCE_3DZ(
    bbox));
CREATE OR REPLACE VIEW triangle_array_all AS SELECT id,
                  st_force_3dz(bbox::geometry) AS st_force_3dz
                  FROM triangle_array;
```

Only three functions are exposed to PostgreSQL, the `tinz_bytea` function and the `trianglez_bytea` function, as well as a helper function used internally. After compiling the triangle_array.c file with the provided Makefile, these functions can be loaded into PostgreSQL as seen in Listing 13.

Listing 13: Triangle array SQL function loading.

```
/* Returns one bytea trianglez for given triangle id */
CREATE OR REPLACE FUNCTION trianglez_bytea(IN integer, IN bytea,
    IN bytea, IN integer,
    OUT geom bytea)
  RETURNS bytea
  AS  'triangle_array.so', 'trianglez_bytea'
  LANGUAGE c IMMUTABLE STRICT;

/* Returns bytea tinz for bucket id */
CREATE OR REPLACE FUNCTION tinz_bytea(IN integer, IN bytea, IN
    bytea, IN integer,
    OUT geom bytea)
  RETURNS bytea
  AS  'triangle_array.so', 'tinz_bytea'
  LANGUAGE c IMMUTABLE STRICT;

/* Helper function. Returns one point, used if outside of bucket
    */
CREATE OR REPLACE FUNCTION trianglexyz(IN integer, IN bytea,
    OUT x double precision, OUT y double precision, OUT z double
        precision)
```

```
  RETURNS SETOF record
  AS  'triangle_array.so', 'trianglexyz'
  LANGUAGE c IMMUTABLE STRICT;
```

Afterwards the two main functions can be called as seen in Listing 14. The output is WKB which can be cast into PostGIS geometry.

Listing 14: Triangle array SQL function calling.

```
/* 91 is the bucket id for each row */
SELECT ST_AsTEXT(tinz_bytea(numt,points,triangles,91)::geometry)
    FROM triangle_array WHERE id = 91
/* 500 is the number of a triangle requested */
SELECT ST_AsTEXT(trianglez_bytea(500,points,triangles,91)::
    geometry) FROM triangle_array WHERE id = 91
```

At the moment the tablename is hardcoded once in the source code. As shown in Chapter 5 the TINZ en TriangleZ types in PostGIS are not very usable yet, but their close neighbours MultiPolygonZ and TriangleZ types are. By changing two lines in the source code (seen in Listing 15), this can be accomplished, demonstrating the difference between MultiPolygonZ and TINZ, PolygonZ and TriangleZ is only one integer.

Listing 15: Changing WKB output.

```
/* in tinz_bytea */
uint32_t wkbtype = 1006; /*  MULTIPOLYGONZ | WKB TINZ is code
    1016 */
/* in trianglez_to_geometry_wkb */
uint32_t wkbtype = 1003; /* POLYGONZ | WKB TRIANGLEZ is code 1017
     */
```

E QUERIES

The results presented in Chapter 5 should be reproducable. In the previous appendices, the data structures and PostgreSQL extensions are described. This appendix is divided into three sections: The commands used for timing the construction, commands used for retrieving the size of database tables and the SQL queries used in benchmarking.

E.1 LOADING TIME

To measure actual construction and loading time, the utility time on Linux is used in the shell.

Listing 16: Timing function.

```
time spfinalize -i rijswijk.laz -ilas -o -v -level 5 -ospb |
    spdelaunay2d -ispb -osmb | python ~/streamingstars/qtfin.py 5
/* output */
real    0m2.170s
user    0m2.584s
sys     0m0.048s
```

The real time is used, because waiting for databases and disk should be included in the performance. In python the time.time() module and function is used, for measuring the creation of indexes.

E.2 DATABASE STORAGE SIZE

For measuring the storage size of the various data structures the statistics page for each table in pgAdminIII, the PostgreSQL client, is used. When this proves to be inaccurate e.g. reporting only sizes in gigabytes, one of the following SQL query is used:

Listing 17: Disk space taken query

```
/* Table pretty size */
SELECT pg_size_pretty(pg_total_relation_size('
    multistar_l10_delft_a'))

/* Table size in bytes */
SELECT pg_total_relation_size('multistar_l10_delft_a')

/* Complete database pretty size */
SELECT pg_size_pretty(pg_database_size( current_database() ))

/* Complete database size in bytes */
SELECT pg_database_size( current_database() )
```

E.3 SPATIAL QUERIES

The following queries are documented for each benchmark. For each benchmark the queries for all the data structures are given, with their names as comments. Take note that most of these queries require specific extensions and backends:

- TINZ (ST_3DIntersection) requires the SFCGAL backend to PostGIS. See their website https://oslandia.github.io/SFCGAL/
- Multistar structures requires the multistar extension, see Appendix C.
- Triangle array structures (TINZ_bytea) requires the triangle array structure, see Appendix D.
- pgTIN functions require the pgtin extension by Ledoux [2013].
- PostGIS is required for all of these functions, except pgTIN.

For large bucketsizes, resulting in few rows, use the following SQL: `SET enable_seqscan TO off;` to prevent the PostgreSQL planner from not using the spatial index.

Listing 18: Point location benchmark of the demo.las dataset.

```
/* point 82419.172,449005.397 demo.laz */

/* Multistar */
SELECT pl(82419.172,449005.397,id) FROM multistar_l4 WHERE bbox
    && ST_POINT(82419.172 449005.397);

/* MultiPolgyonZ, same for single type PolygonZ */
SELECT ST_Intersection(geom,ST_POINT(82419.172 449005.397)) FROM
    multipoly_l4 WHERE geom && ST_POINT(82419.172 449005.397);

/* TINZ, same for single type TriangleZ */
SELECT ST_Intersection(geom,ST_POINT(82419.172 449005.397)) FROM
    multitinz_l4 WHERE geom && ST_POINT(82419.172 449005.397);

/* Triangle Array */
SELECT ST_Intersection(tinz_bytea(numt,points,triangles,id)::
    geometry,ST_POINT(82419.172,449005.397)) FROM multitin_l5_td
    WHERE bbox && ST_POINT(82419.172,449005.397)

/* pgTIN */
SELECT pl(82419.172,449005.397)
```

Listing 19: Line intersection of demo.las

```
/* line from 82506.264,448925.855 to 82639.791,449024.138 in demo
    .las */

/* Multistar */
```

```
SELECT profile_count_intersections
    (82506.264,448925.855,82639.791,449024.138,id) FROM
    multistar_l6 WHERE bbox && ST_POINT(82506.264,448925.855)

/* TINZ, but needs ST_FORCE2D, making it useless */
SELECT ST_ASTEXT(ST_3DIntersection(ST_FORCE2D(geom), ST_MakeLine(
    ST_POINT(82506.264,448925.855),ST_POINT(82639.791,449024.138)
    ))) FROM multitinz_l7 WHERE ST_INTERSECTS(geom,ST_MakeLine(
    ST_POINT(82506.264,448925.855),ST_POINT(82639.791,449024.138)
    ))

/* MultiPolygonZ */
SELECT ST_ASTEXT(ST_Intersection(geom, ST_SETSRID(ST_MakeLine(
    ST_POINT(82506.264,448925.855),ST_POINT(82639.791,449024.138)
    ),28992))) FROM multipoly_l5 WHERE ST_INTERSECTS(geom,
    ST_MakeLine(ST_POINT(82506.264,448925.855),ST_POINT
    (82639.791,449024.138)))

/* Triangle Array */
SELECT ST_Intersection(tinz_bytea(numt,points,triangles,id)::
    geometry,ST_MakeLine(ST_POINT(82506.264,448925.855),ST_POINT
    (82639.791,449024.138))) FROM multitin_l4_td WHERE
    ST_INTERSECTS(bbox,ST_MakeLine(ST_POINT(82506.264,448925.855)
    ,ST_POINT(82639.791,449024.138)))

/* pgTIN */
SELECT profile_count_intersections(82506.264,448925.855,
    2686.674,448926.463)
```

Listing 20: Horizontal line intersection of demo.las

```
/* line from 82348.405,449248.327,82556.899,449248.503 in demo.
    las */

/* Multistar */
SELECT profile_count_intersections
    (82348.405,449248.327,82556.899,449248.503,id) FROM
    multistar_l6 WHERE bbox && ST_POINT(82348.405,449248.327)

/* MultiPolygonZ */
SELECT ST_ASTEXT(ST_Intersection(geom, ST_SETSRID(ST_MakeLine(
    ST_POINT(82348.405,449248.327),ST_POINT(82556.899,449248.503)
    ),28992))) FROM multipoly_l5 WHERE ST_INTERSECTS(geom,
    ST_MakeLine(ST_POINT(82348.405,449248.327),ST_POINT
    (82556.899,449248.503)))
```

Listing 21: Diagonal line intersection of g37en2.laz.

```
/* line 85243.601,443932.968,89657.005,449718.544 g37en2.laz */

/* Multistar */
SELECT profile_count_intersections
    (85243.601,443932.968,89657.005,449718.544,id) FROM
    multistar_g37_l8 WHERE bbox && ST_POINT(85243.601 443932.968)
    ;
```

```
/* pgTIN */
SELECT profile_count_intersections
    (85243.601,443932.968,89657.005,449718.544);

/* Triangle Array */
SELECT ST_Intersection(tinz_bytea(numt,points,triangles,id)::
    geometry,ST_MakeLine(ST_POINT(85243.601,443932.968),ST_POINT
    (89657.005,449718.544))) FROM multitin_g37_l8 WHERE
    ST_INTERSECTS(bbox,ST_MakeLine(ST_POINT(85243.601,443932.968)
    ,ST_POINT(89657.005,449718.544))
```

Listing 22: Atomic functions. Each mode is one of the atomic functions, returned only as log messages.

```
/* basic point location */
SELECT pl2(82416.692,449007.626,id,0) FROM multistar_l7 WHERE
    bbox && ST_POINT(82416.692,449007.626);

/* aspect */
SELECT pl2(82416.692,449007.626,id,1) FROM multistar_l7 WHERE
    bbox && ST_POINT(82416.692,449007.626);

/* slope */
SELECT pl2(82416.692,449007.626,id,2) FROM multistar_l7 WHERE
    bbox && ST_POINT(82416.692,449007.626);

/* local minimum */
SELECT pl2(82416.692,449007.626,id,3) FROM multistar_l7 WHERE
    bbox && ST_POINT(82416.692,449007.626);

/* local maximum */
SELECT pl2(82416.692,449007.626,id,4) FROM multistar_l7 WHERE
    bbox && ST_POINT(82416.692,449007.626);

/* degree */
SELECT pl2(82416.692,449007.626,id,5) FROM multistar_l7 WHERE
    bbox && ST_POINT(82416.692,449007.626);
```

COLOPHON

This document was typeset using LaTeX. The document layout was generated using the `arsclassica` package by Lorenzo Pantieri, which is an adaption of the original `classicthesis` package from André Miede.

This thesis uses and has contributed elements from and to the MSc Geomatics Thesis Template found at https://github.com/tudelft3d/MScGeomaticsThesisTemplate.

The figures and diagrams were mostly drawn using IPE, Inkscape and QGIS. The cover was generated with Rhino and Grasshopper.