

Implementation and testing of variable scale topological data structures

Experiences with the GAP-face tree and GAP-edge forest

MSc Geomatics thesis by B.M. Meijers

June 2006



Delft University of Technology
Section GIS Technology
The Netherlands

Implementation and testing of variable scale
topological data structures
Experiences with the GAP-face tree and GAP-edge
forest

Thesis

submitted in partial fulfillment of the
requirements for the degree of

Master of Science

in

Geomatics

by

B.M. Meijers
born in Delft, The Netherlands

Professor: Prof. dr. ir. P.J.M. van Oosterom
Supervisor: Drs. C.W. Quak

Delft University of Technology
Section GIS Technology
The Netherlands

Abstract

With the increase of the availability of large-scale geographic data set and the rise of widespread computer networks, such as the Internet, the need has arisen to be able to transfer this data by means of these networks. The networks form the basis for a Geographic Information Infrastructure (GII), in which data users, data providers and data producers are connected with each other.

There also exists the need to offer this data on several scales to end users, for example to get an overview of an area first. Because the geographical information are now sent by the computer networks and large-scale geographic information brings many data, data reduction must take place. This is to prevent that sending of the information takes too much time. Generalization of geographical information is a possible means to let this reduction take place.

Generalization is the selection and simplification of detail appropriate to the scale and or purpose of the map. The appliance of generalization demands that choices must be made with respect to which geographical objects are selected and simplified and how this selection and simplification must take place. Moreover, also the surroundings of the objects to be generalized, are often taken into account in the generalization process, which makes that the complete process even requires more time. This way, the complete process can not be carried out in real time.

Earlier, reactive data structures, in which geographical information is stored in the computer with several levels of detail, have been proposed as a solution to allow the use of generalized large-scale geographical information in real time. So far, these data structures were using redundancy with respect to geometry. For this reason a new conceptual model has been developed, where a number of existing data structures have been combined into two new data structures, namely, the GAP face tree and the GAP edge forest (described in Van Oosterom, 2005). The complete structure is termed tGAP structure, in which tGAP stands for topological Generalized Area Partitioning.

The tGAP structure has not been theoretically verified, nor implemented or tested. Therefore, the objective of this research is to theoretically verify the data structures and to test the data structures considering requirements such as loading time and storage capacity.

To reach the objective literature study has been performed in the field of generalization,

database management systems and the data structures. Moreover, a prototype has been built, with which the data structures have been implemented in a mainstream database management system (DBMS) with spatial data types.

Literature study has shown that generalization is a key issue in the complete process of obtaining and processing geo-information and that using reactive data structures is a suitable option to offer the results of generalization within a GII in real time.

The implementation of a prototype has shown, that it is possible to implement the data structures in a mainstream DBMS. The data structures are implemented in Oracle Spatial, the DBMS, and by means of Apache, a web server, opened up to Google Earth, a geographical viewer. The data structures in the prototype make it possible to view the geographical data interactively within the viewer, i.e. within 15 seconds, independent from the size of the area to be loaded.

The final conclusion must be, that with some workarounds and with some changes to the proposed conceptual model, it is possible to implement the model as described in (Van Oosterom, 2005). With an implementation it becomes possible to show geographical data on a variable number of detail levels and the implementation shows that the data structures can provide the desired data reduction within a GII in real time.

Samenvatting

Met de groei van het aanbod van grootschalige geografische data sets en het beschikbaar komen van wijdverspreide computernetwerken, zoals het Internet, is er de behoefte ontstaan om de data te kunnen verspreiden via deze netwerken. Deze netwerken vormen de basis voor een Geografische Informatie Infrastructuur (GII), waarbij data gebruikers, data aanbieders en data producenten met elkaar verbonden zijn.

Er bestaat ook de behoefte om deze data op meerdere schalen aan eindgebruikers te kunnen aanbieden, bijvoorbeeld om eerst een overzicht van een gebied te krijgen. Echter, doordat de data nu per computernetwerk verstuurd wordt en de geografische data sets veel gegevens met zich mee brengen, zal data reductie plaats moeten vinden. Dit om te voorkomen dat het oversturen van de informatie erg lang duurt. Generalizatie van geografische informatie is een mogelijk middel om deze reductie plaats te laten vinden.

Generalizatie is het selecteren en simplificeren van geografische informatie, zodat dit is aangepast op het doelgebruik en de schaal van een kaart. Het toepassen van generalizatie vergt echter dat er ingewikkelde keuzes gemaakt moeten worden met betrekking tot welke geografische objecten er geselecteerd en gesimplificeerd moet worden en hoe deze selectie en simplificatie moet gebeuren. Daarnaast wordt vaak ook nog de omgeving van te generalizeren objecten meegenomen in het generalizatie proces, wat de hele onderneming nog tijdsintensiever maakt. Dit maakt dat het hele proces niet in real time kan worden uitgevoerd.

Eerder al zijn reactieve data structuren, waarbij geografische informatie met meerdere detailniveaus wordt opgeslagen in de computer, aangedragen als oplossing om toch in real time generalizaties van de grootschalige geografische informatie te kunnen aanbieden. Deze structuren gebruikten tot nu toe redundantie voor wat betreft het opslaan van geometrie. Daarom is er een nieuw conceptueel model uitgewerkt, waarbij een aantal bestaande data structuren gecombineerd worden in twee nieuwe data structuren, te weten de GAP face boom en het GAP edge bos (beschreven in Van Oosterom, 2005). De complete structuur is de tGAP structuur gedoopt, waarbij tGAP staat voor topologisch gegeneralizeerde vlakken partitie, in het Engels topological Generalized Area Partitioning.

De tGAP structuur is noch theoretisch doorgelicht, noch geïmplementeerd of getest. Het doel van dit onderzoek is om deze data structuren theoretisch door te lichten en te

testen op vereisten zoals benodigde laadtijd en opslagcapaciteit.

Om dit doel te bereiken is literatuurstudie verricht op het gebied van generalizatie, database management systemen en de data structuren. Daarnaast is een prototype gebouwd, waarbij de data structuren geïmplementeerd zijn in een mainstream database management systeem (DBMS) met ruimtelijke data typen.

De literatuurstudie heeft laten zien dat generalizatie een sleutelbegrip is in het hele proces van inwinnen en verwerken van geo-informatie en dat het gebruiken van reactieve data structuren een geschikte optie is om de resultaten van generalizatie binnen een Geografische Informatie Infrastructuur in real time aan te kunnen bieden.

De implementatie van een prototype heeft aangetoond, dat het mogelijk is om de data structuren in een mainstream DBMS te implementeren. De data structuren zijn geïmplementeerd in Oracle Spatial, het DBMS, en via Apache, een webserver, ontsloten naar Google Earth, een geografische viewer. De data structuren in het prototype maken het mogelijk om binnen de viewer interactief, i.e. binnen 15 seconden, elke willekeurige uitsnede van het ingeladen gebied te bekijken.

De slotconclusie moet zijn, dat het met wat omwegen en veranderingen toch goed mogelijk is het conceptuele model beschreven in (Van Oosterom, 2005) te implementeren. Met een implementatie wordt het mogelijk om geografische informatie op een variabel aantal detailniveaus in real time te tonen en de implementatie laat zien dat deze data structuren daarmee de gewenste data reductie in een GII kunnen leveren.

Acknowledgements

This thesis is the result of my graduation project and marks the end of my Master of Science study at Delft University of Technology. It describes the research project that has been performed at the OTB Research Institute for Housing, Urban and Mobility Studies from November 2005 to June 2006. During this time, a lot of people have contributed, directly or indirectly, to this project.

First of all, I thank Peter van Oosterom for publishing his ideas, which form the basis for this research and for all valuable comments made during our discussions. I also like to thank Marian de Vries for the discussions I had with her on how to improve and work with the data structures. Theo Tijssen always was available when I screwed up with the database (again) or had some other question related to Oracle. Jeroen de Vries, from the municipality of Amsterdam, supplied test data for the research project. Arta Dilo gave useful comments on my thesis. Last but not least, I would like to thank Wilko Quak, for helping me out with programming issues, brainstorming and giving guidance on where to go next. Thanks all, for the open, professional and inspiring atmosphere to work in.

Moreover, I would like to thank Fieke, for her love and support, even when I could not free my mind from generalization issues in my spare time. Above all, thanks goes to my parents, Koos and Jaap, for their continuous support and encouragement.

Thank you.

Naaldwijk, 19th June 2006

Martijn Meijers

Contents

Abstract	v
Samenvatting	vii
Acknowledgements	ix
Contents	xi
List of Figures	xv
List of Tables	xvii
List of Listings	xix
List of Terms	xxi
1 Introduction	1
1.1 Problem statement	2
1.2 Research objective	2
1.3 Research framework	2
1.4 Research issues	3
1.5 Structure of the thesis	3
2 On automated generalization	5
2.1 Storing the world in a computer system	5
2.1.1 A short overview of GIS and DBMS	6
2.1.2 Planar area partitions	8
2.2 Generalization of geographical data	9
2.2.1 A rough classification of generalization	10
2.2.2 How generalization takes place	12
2.2.3 Additional needs for model generalization	12
2.3 Topology	15
2.3.1 Left right topology without edge references	16
2.3.2 Left right topology with edge references	18

2.3.3	Winged edge topology	18
2.4	Reactive data structures for generalization	19
2.4.1	Generalized Area Partition face tree	20
2.4.2	Edge simplification: Binary Line Generalization tree	21
2.4.3	Generalized Area Partition edge forest	23
2.5	Advantages the data structures offer	25
2.5.1	Increasing the use of framework data	25
2.5.2	Real time need for generalization	25
2.5.3	Continuous range of levels of detail	25
2.5.4	Non-redundant topological storage	26
2.6	Prerequisites for the data structures	26
2.6.1	Requirements with respect to the face information	26
2.6.2	Requirements with respect to the edge information	27
2.7	Available DBMSs suitable for implementation	27
2.8	Final remarks	28
3	Methodological justification	29
3.1	Need for experiments	29
3.2	Implementation experiments	30
3.3	Benchmarks	31
3.4	Final remarks	31
4	Implementation and testing of the tGAP structure	33
4.1	Implementation	33
4.1.1	Topological testing	34
4.1.2	Geographic data selection	37
4.1.3	Data models	39
4.1.4	Binary Line Generalization (BLG) tree data type	42
4.1.5	Generalized Area Partition (GAP) face tree	45
4.1.6	GAP edge forest	47
4.1.7	Architecture to visualize the content of the tGAP structure	52
4.1.8	Lessons learnt from implementing	54
4.2	Testing	55
4.2.1	Removing redundant geometrical storage	56
4.2.2	Size of tGAP structure, in comparison with the source topology	56
4.2.3	Loading speed	59
4.2.4	Tuning loading speed	59
4.2.5	Visualization speed	62
4.2.6	Building a 3D functional index	62
4.2.7	Problems encountered with mapping scale to importance	66
4.2.8	Lessons learnt from testing	68
4.3	Possible theoretical improvements to the data structure	68
4.3.1	Changing the data model with respect to the GAP edge forest	68
4.3.2	Self-intersections of edges due to the Douglas-Peucker algorithm	71

4.3.3	Smooth zooming, gradual changes in display	71
4.3.4	Collapse of area objects to lines and points	72
4.3.5	Simplification of Geographic Shape by Discrete Curve Evolution .	73
4.4	Final remarks	73
5	Conclusions, recommendations and future research	75
5.1	Conclusions	75
5.2	Recommendations	76
5.3	Future research	76
A	Example of request–response sequence of Google Earth, Apache and Oracle	79
B	Size of indices on the source topology and tGAP tables	85
	Bibliography	89

List of Figures

1.1	Research framework	3
2.1	Gradual changes in GIS architecture	8
2.2	Why generalization is needed	10
2.3	A valid polygon	16
2.4	Left right topology without start edge references	17
2.5	Left right topology with start edge references	18
2.6	Winged Edge topology	19
2.7	Merge steps for construction of the GAP face tree	20
2.8	Edge and its coordinates	22
2.9	First two iterations for the Douglas-Peucker algorithm	23
2.10	BLG tree	24
4.1	Overview of constructing the tGAP structure	34
4.2	Performance of topological face reconstruction	35
4.3	Test data used for reconstruction of face geometry	36
4.4	Artificial dataset	37
4.5	UML diagram for left right topology	39
4.6	UML diagram for the tGAP structure	41
4.7	Steps for construction of the GAP face tree	46
4.8	Edges in the tGAP structure: the GAP edge forest	48
4.9	Visualizing contents of the tGAP structure in Google Earth	53
4.10	Time needed to build the GAP face tree	60
4.11	Tuning building speed of the GAP face tree	61
4.12	Intersection of BLGs	72
A.1	Window of Google Earth after request-response sequence	83

List of Tables

2.1	Different generalization operators	13
4.1	Information in the source topological data sets	38
4.2	Binary Line Generalization data type, filled with data	44
4.3	tGAP face table after constructing the GAP face tree	47
4.4	tGAP edge table after constructing the GAP edge forest	49
4.5	tGAP BLG table after constructing the GAP edge forest	51
4.6	Size of geometrical version of the source topology tables	56
4.7	Size of source topology and tGAP tables	57
4.8	Example edge, showing that some fields stay the same in different versions of the same edge	58
B.1	Index size of indices on non-spatial attributes of source topology structure	85
B.2	Physical size of spatial (function based) indices	86
B.3	Index size of indices on non-spatial attributes of the tGAP structure	87

List of Listings

4.1	Source topology: physical data model	40
4.2	tGAP structure: physical data model	41
4.3	Creating the BLG tree object within Oracle	43
4.4	Updating spatial reference identifier for geometry in a table	54
4.5	Creating the 3D functional index	63
4.6	Query which will use a 2D and a 1D index	64
4.7	Query which will use the 3D functional index	64
4.8	Query plan of query using the 2D and a 1D index	65
4.9	Query plan of query using the 3D functional index	65
4.10	Proposed tGAP face table	70
4.11	Proposed tGAP edge table	70
4.12	Proposed tGAP original BLG table	70
4.13	Proposed tGAP joined BLG table	70
4.14	Proposed tGAP BLG view, so that it appears as one table	70
A.1	Request from Google Earth to Apache over HTTP	79
A.2	Database query fired from Apache to Oracle DBMS	80
A.3	GML document, as available in the memory of the webserver after transformation	80
A.4	GML to KML with XSLT	81
A.5	Apache response to Google Earth in KML	82

List of Terms

API	Application Programming Interface
BLG	Binary Line Generalization
DAG	Directed Acyclic Graph
DBMS	Database Management System
DCL	Data Control Language
DDL	Data Definition Language
DEM	Digital Elevation Model
DML	Data Manipulation Language
GAP	Generalized Area Partition
GBKN	Grootschalige Basiskaart Nederland
GDAL	Geospatial Data Abstraction Library
GII	Geographic Information Infrastructure
GIS	Geographic Information System
GML	Geography Markup Language
ICA	International Cartographic Association
KML	Keyhole Markup Language
LKI	Landmeetkundig Kartografisch Informatiesysteem
LOD	Level of detail
OGC	Open Geospatial Consortium
OO-DBMS	Object-Oriented Database Management System
OR-DBMS	Object-Relational Database Management System

R-DBMS	Relational Database Management System
RD	Rijks Driehoekstelsel
SOA	Service Oriented Architecture
SQL	Structured Query Language
tGAP	topological Generalized Area Partitioning
UML	Unified Modelling Language
W3C	World Wide Web Consortium
WGS84	World Geodetic System 1984
XML	eXtensible Markup Language
XSLT	eXtensible Stylesheet Language Transformations

Chapter 1

Introduction

A lot of large scale geographic data sets have been collected and are available nowadays, e.g. in The Netherlands some large scale datasets are: large scale base map of The Netherlands (GBKN), Top10NL and cadastral maps of the Netherlands (LKI). With the growth of the Internet, it is possible to transfer these large scale data sets from a server to a user, via a wired or even wireless network. In this network infrastructure, different clients, ranging from desktop Geographic Information Systems (GIS) to mobile handhelds can interact with the data sources.

Although the large scale data sets offer a lot of detail, many users want to have an overview of the geographical area first and thus require less detailed geographic data. Another issue concerning transfer of large scale data sets is the size of the data sets, which brings problems with transferring the data in real time. To prevent these problems, generalization of the geographic data is a possible solution. Generalization is “the selection and simplified representation of detail appropriate to the scale and or purpose of the map” (ICA, 1973).

Generalization of geographic data can be applied in different ways. One solution is to collect and store the geographic information at various levels of detail. This is sub-optimal, because this results in redundancy in information and may cause inconsistencies between the different levels, making it difficult to keep the information up to date.

Another approach could be to generalize the geographical data in real time. However, decisions related on how to apply generalization are computation intensive and require a lot of time. When generalizing, the surroundings of objects to be generalized can be taken into account, which requires even more time. This makes it impossible to fulfill the generalization process in real time. Therefore, reactive data structures have been proposed to store the results of the generalization process (Van Oosterom, 2005). According to (Van Oosterom and Schenkelaars, 1995), “[a] reactive data structure is defined as a *geometric* data structure with *detail levels*, intended to support the sessions of a user working in an interactive mode. It enables the information system to *react* promptly to user actions.” With using the data structures, browsing the geographic data sets at multiple levels of detail then can take place in real time.

1.1 Problem statement

Up to now reactive data structures for generalization were using redundancy with respect to geometry, using complete polygons for every area object (Van Oosterom and Schenkelaars, 1995) or, when topology was used (to avoid redundant storage of stored boundaries) this was limited: when edges are merged, again redundancy is introduced (Vermeij, 2003).

Based on the previous structures a new improved combination has been proposed by (Van Oosterom, 2005). A combination of an extended Generalized Area Partition tree (called the GAP face tree) and Binary Line Generalization trees (combined into a forest of BLG trees) structured in a topological way were proposed to solve the problems of the earlier attempts. This combination is termed the topological Generalized Area Partitioning structure, for short, the tGAP structure. However, this proposed solution has not been verified in theory and it has not been implemented and tested yet.

1.2 Research objective

The objective of this research is to verify the GAP-face tree and GAP-edge forest in theory and test the functionality and performance, in terms of time and storage requirements, of the tGAP structure for on-the-fly database map generalization. This is done by verification of the theory of the data structures via literature study and implementation and testing of the data structures, which make it possible to verify and possibly improve the theory (in case of discovered drawbacks). The implementation and tests also deliver information to allow comparison of these data structures with other approaches for storing geographic information with multiple levels of detail.

1.3 Research framework

To reach the objective of the research a framework is established, which guides the research. This framework consists of four parts: (a) the research objects, (b) the research perspective, (c) interrelation of research results and (d) the objective of the research. It is graphically depicted in figure 1.1.

The objective as formulated in section 1.2 is reached by following the path of conducting literature study on the proposed data structures in the context of automated map generalization theory and giving a theoretical description of an implementation in an Object Relational Database Management System (OR-DBMS). This theoretical background then provides enough points of interests for implementing and testing the data structures. Based on the implementation experiment and test results conclusions and recommendations regarding the theory and performance of the data structures can be given.

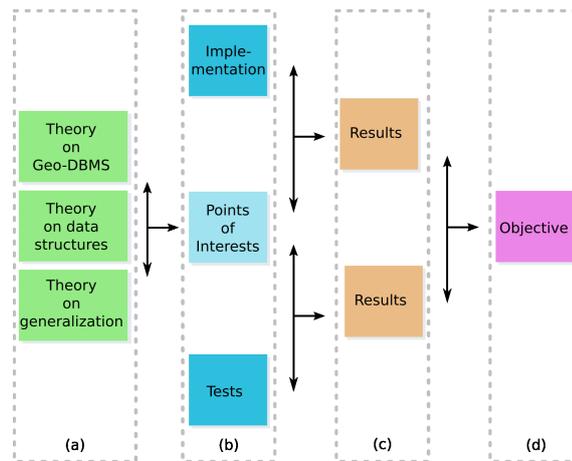


Figure 1.1: Research framework that guides this research. It is split in four parts: (a) research objects, (b) research perspective, (c) interrelation of research results and (d) the objective of the research.

1.4 Research issues

To help reaching the research objective, the following questions should be answered:

1. What are the good things about having these data structures inside the database for map generalization?
2. How can the GAP-face tree and GAP-edge forest be implemented in an object-relational database management system?
3. What problems occur when implementing the data structures inside the database?
4. How can we assess the performance of the implemented structures?
5. What do we learn from an implementation of the GAP-face tree and GAP-edge forest data structures in an object-relational database management system?

1.5 Structure of the thesis

The results of the research are described in the chapters of this thesis. Chapter 2 puts the work in context and summarizes backgrounds on GIS and topology. It describes the needs and problems related to generalization. After this, it introduces the solution of using topological reactive data structures for automated generalization and shows what the needs and advantages of these structures are. Chapter 3 justifies the methods of implementing and testing the data structures for reaching the research objective as described in section 1.2. In chapter 4 a description of the implementation and testing

of the GAP face tree and GAP edge forest is given and it shows the results of the experiments. Chapter 5 gives conclusions and recommendations and describes possible future research directions.

Chapter 2

On automated generalization of polygons within area partitions

Although a lot of research has been carried out in this field (e.g. McMaster and Shea, 1992; Müller et al., 1995; Weibel, 1997; Galanda, 2003; Stoter et al., 2004), the problem of automated generalization of geographic data is still not completely solved. This chapter presents background, needs and problems related to automated generalization. It also introduces a proposed solution to the needs and problems related to generalization: the use of reactive data structures within a DBMS.

The first section (2.1) contains theory on geographic information systems to set the thesis in context. Issues that are related to generalization are given in the second section, 2.2. Because the data structures heavily rely on topology, an introduction to this topic is given in section 2.3. After this section, the reactive data structures, i.e. Generalized Area Partition (GAP) face tree and GAP edge forest, for short the tGAP structure, are introduced in section 2.4. Questions on what problems are solved using the data structures, what is needed to implement the data structures and which DBMS is suitable for implementation are answered in sections 2.5–2.7. The chapter concludes with some final remarks in section 2.8.

2.1 Storing the world in a computer system

This section is largely based on information found in (Longley et al., 2005, Chapter 3). To capture and represent the real world inside a computer system abstractions have to be made. This is due to the fact that the world is infinitely complex, but computer systems are discrete, finite machines. Geographic concepts are found in the real world, on, or near, the earth surface. Two conceptual views exist to look at this surface:

Continuous field approach With this view, the real world is represented as a finite number of variables, each one defined at every possible position. The variables can represent different things, like height, type of vegetation, etcetera.

Discrete objects approach This view represents the geographic world as objects with well-defined boundaries. Objects are houses, roads and railways, for example.

These views do not solve the problem that the representation for a computer system should be finite. The amount of detail to be captured has to be limited. However, with both views, the world can still be represented as an infinite amount of information. Other methods of representing geographic data in digital computers are needed. Two representation methods exist:

Raster representation The real world is divided into arrays of cells (e.g. squares). To these cells attributes are assigned.

Vector representation Features in the real world are represented with primitive features, like points, lines and areas. To the points, lines and areas attributes can be assigned.

Both the conceptual views and computer representations can be related to each other. However, there is a strong association between raster and fields approach and between vector and discrete objects. Normally which conceptual view and which digital representation is taken, depends on the needs of the application. In this thesis the conceptual view will be that of 'discrete objects' with a 'vector representation' stored in the computer (see section 2.1.2).

2.1.1 A short overview of GIS and DBMS

A GIS is a computerized system that helps managing geographic information. As defined by Worboys (1995):

“[a] geographic information system (GIS) is a computer-based information system that enables capture, modelling, manipulation, retrieval, analysis and presentation of geographically referenced data.”

A geographic information system is centred around geographical data. This data is normally stored in a repository called a database. As described by Worboys (1995), a database is:

“a repository of data, logically unified but possibly physically distributed over several sites, and required to be accessed by many users. It is created and maintained using a general purpose piece of software called a database management system (DBMS).”

The mainstream form of database management systems found these days, has its roots in the relational model described by Codd (1970). These systems are termed relational database management systems (R-DBMS).

To store non-spatial data in a DBMS several (simple) data types are provided. For example, to store financial data, numeric data types, like integers and floating points, are available. To store texts, a string data type is available.

Spatial data is more complex to store, due to its multi-dimensional nature. However, spatial vector data types are not always available in a DBMS. To store a geographical object in two dimensions in a relational database management system, one could use the simple data types available in a normal relational DBMS. This is not an optimal choice, with respect to performance. For example, storing a line in a relation DBMS, several references have to be stored, because the line has a relationship with points and the points have coordinates in two dimensions. The x- and y-coordinate can be stored, as belonging to a point. When materializing the line, all references need to be used. A bigger problem that occurs is that the DBMS does not know that the line is a line object and can not check it for validity or apply operations on the line. Instead, a better solution would be to store geographic information with their own primitive data types, like points and lines, in the DBMS. This way, it is needed to have a way to extend the database with new data types, or the vendor of the database management system needs to provide the built-in spatial data types.

This calls for complete geographic objects inside the database. Another approach for creating a database, i.e. using an object oriented paradigm (OO-DBMS), is still less common, but would be more appropriate for the use of spatial data, due to its multi-dimensional nature. Therefore, a mixture of object-oriented techniques and relational databases have been developed, which makes it possible to store geographic information with their own spatial data types, like points, lines and areas inside the DBMS.

The adaptation of these spatial data types in mainstream DBMS technology meant gradual changes in the architecture of GIS systems. Different architectures have been adopted and are shown in figure 2.1. The architectures evolved since the second half of the 1980's from hybrid systems (dual architecture), to mid 1990's spatial cartridges (layered architecture) and, since the late 1990's and beginning of the twenty-first century, to an integrated architecture.

With a dual architecture, the representation of geographical objects is broken into two pieces. A spatial subsystem is used for storage and querying of spatial data, together with a separate DBMS for administrative data. The link between the two systems is maintained through the use of unique identifiers. This architecture means freedom to use efficient data structures and algorithms in the spatial subsystem. However, queries for such a system must be decomposed into two parts, so it is complex to do the query processing and no global query optimization is possible. Additionally, the two separate subsystems might show inconsistencies with respect to each other.

A layered architecture stores all data in a single DBMS with the spatial engine responsible for generating spatial knowledge contained in a middleware layer between the application and the DBMS, separate from the DBMS itself. Drawbacks here are that the DBMS itself still does not understand the semantics (i.e. meaning) of the spatial data and can not perform true query optimization.

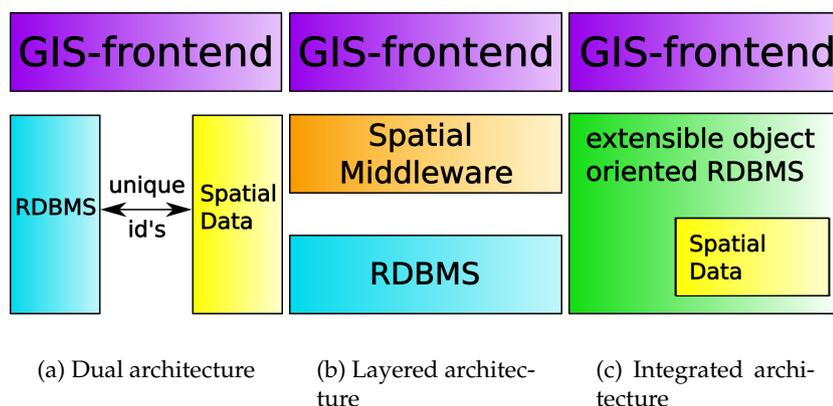


Figure 2.1: Gradual changes in GIS architecture: three different GIS architectures in which the integrated architecture is most advanced, taken from Van Oosterom (2001, p. 8)

An integrated spatial DBMS architecture means that spatial data types are integrated with a normal database system, which means that for both non-spatial *and* spatial data, data types are available. Operations for spatial data are embedded within the database. Indexing of both data types is possible, because for each type a suitable index is available. Several spatial indexing techniques are shown in (Van Oosterom and Vijlbrief, 1996). The database engine should make use of such indices. To optimize the speed of the system a query is mostly split in two parts: a filter operation, where a rough answer is found solely based on the index placed on top of the objects, and second, an operation based on the exact geometry of the objects selected by the index giving a more accurate answer.

Having the spatial data types available within a DBMS alone is not sufficient for accomplishing the goals set for a GIS (amongst others, modelling, updating, analyzing and retrieving geographical data). To set up models (structures) that use the data types effectively and to have a possibility to insert, retrieve, update and delete data to and from these created models, in theory, different languages are needed: A data definition language (DDL) is needed for creating the models. After creating the models, there is a need to manipulate the data (insertion, updating, deleting and retrieval), here a data manipulation language (DML) is needed. To arrange access to the data, also a data control language (DCL) is needed. In practice, these languages have been unified into one industry standard language which is called the structured query language (SQL).

2.1.2 Planar area partitions

For storing spatial data, it is necessary to make a choice on how to transform the real world into a model that can be stored inside a database in a computer system. The pro-

cess of modelling spatial data for storage inside a database normally consist of several phases, in which different models are defined (as outcome of the modelling process).

First, a conceptual model is created. The main objects (entities) for the application and how they should look like are defined. In this phase a global view of the data to be stored is given. Then, a logical model is constructed. This logical model consists of a precise description of the data used in the conceptual model. During this phase all entities, relationships and attributes are identified. This design is independent from which OR-DBMS will be used. At last, a physical model is defined. Here design takes place for implementing the logical model, that considers the OR-DBMS to be used. During this phase, tables, indices, and so on, will be described.

A possible choice for a conceptual model is to use a planar area partition for storing geographical data. In a planar area partition for a two dimensional domain, the whole area is covered by polygons (area objects). The coverage is created in such a way, that no gaps, nor overlaps, do exist within the mapped domain. This kind of conceptual model is often used in the geographic application domain to model a part of the real world. Examples of planar partitions are soil maps, cadastral maps and maps showing land use. This research focusses on the use of this type of conceptual model for storing and generalizing geographical data, because it is applicable within diverse applications.

2.2 Generalization of geographical data

Generalization is a key issue throughout the whole process of data collection and map making. It already starts when data is collected within the field. It also takes place when there is a need to make a map with a different level of detail than the data that already have been collected. A formal definition of generalization is given by ICA (1973). Generalization is:

“the selection and simplified representation of detail appropriate to the scale and or purpose of the map”.

For generalization, selection is done by taking away irrelevant detail or objects that are not relevant for a lower scale¹, so that a reduction of data takes place. Scale alone is not the only requirement for selecting which objects to show on a lower scale level. Also the intended use of a map influences which objects will be shown. Simplification is done by changing the geometric shapes of objects.

This process of selection and simplification should take place, because the physical constraints of the map (the size), limits the amount of objects to be shown at a certain scale (cf. Töpfer and Pillewizer, 1966). An example of this is shown in figure 2.2. The map on the left side is the original. The upper right map is scaled to half the size of the original. It can be observed, that it is not very legible any more. The map on the lower

¹Scale is the ratio between the size of an object in reality and its representation on a map. Small scale maps depict a mapped region on a small area, while large scale maps need a large area to depict the same region.

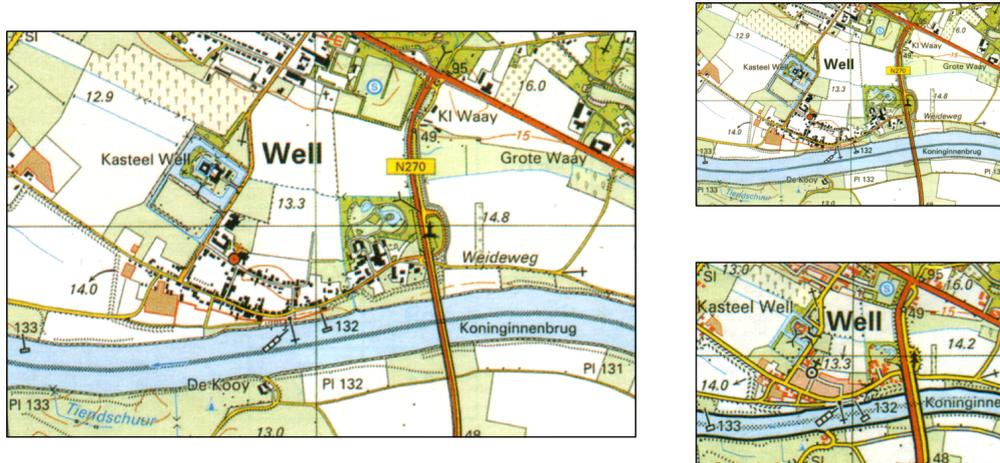


Figure 2.2: Why generalization is needed. Map on the upper right is scaled to half the size of the original map. Map on the lower left, is apart from scaled, also generalized, improving readability. Data source: © Dutch Topographic Survey

right side is also scaled, but it also has been generalized, which improves the readability of the map.

2.2.1 A rough classification of generalization

Nowadays, maps are shown more and more on computer screens, for which the physical constraints are not always the same (as opposed to the paper on which a map is printed). Often it is also possible for a user to interactively zoom in and zoom out, therefore scale is not a fixed parameter any more, so instead it is better to use the term 'level of detail' (LOD).

Although digital generalization has many differences with the traditional, manual generalization process, the main reasons to generalize are still the same, as observed by McMaster and Shea (1992), quoted in (Weibel, 1997):

“Digital generalization can be defined as the process of deriving from a data source, a symbolically or digitally-encoded cartographic data set through the application of spatial and attribute transformations. Objectives of this derivation are: to reduce in scope the amount, type and cartographic portrayal of the mapped or encoded data consistent with the chosen map purpose and intended audience; and to maintain clarity of presentation at the target scale”.

The need for having geographical data available at multiple levels of detail does still exist. Because GISs and Geo-DBMSs are used for managing geographical data, there is a need to use these for automation of the generalization process. As formulated by Müller et al. (1995, p. ix):

“Generalization is motivated by the need to provide multiple views of geographical data at various scales and levels of resolution. It is a tool which has been in use for many years, particularly in cartography for the production of maps at smaller scales. [...] The introduction of geographical information systems (GIS) for planning purposes and automated map production in mapping agencies provides a new *raison d’être* to generalization, namely as a tool to model data to support either spatial analysis or the automated production of maps at multiple scales.”

Generalization is thus embedded within a work flow. According to literature (Müller et al., 1995; Galanda, 2003), this process can be split up in three different parts:

Object generalization with data collection, objects are abstracted from the real world, but with a finite amount of observations and measurements. Objects are selected and simplified, i.e. generalized.

Model generalization performs a ‘controlled reduction of data’ (within the database), to generate a smaller amount of objects for the targeted level of detail.

Cartographic generalization the visual appearance of objects is changed (e.g. symbolization of a church and moving objects, so that these do not overlap on a smaller scale) so that the readability of maps is improved.

The focus within this thesis is on model generalization (reducing the amount of data within the database for another level of detail). Model generalization can be performed with different approaches according to Müller et al. (1995, chapter 1):

Batch generalization with this approach, input is a geographic dataset, based on algorithms, rules, or constraints and without human intervention, the computer returns an output dataset;

Interactive generalization this is a slightly different approach compared to batch generalization. The user interacts with the computer to produce the generalized output set.

The output of these two approaches can be stored in a Multi-representation Database (MRDB). A MRDB stores explicit relations between the objects at the different levels of details in multi-representation data structures. This way updates can be propagated via the links that are stored between the objects. However, from a managing point of view, it is more convenient to update only one level of geographic data, compared to having to update multiple levels of data, because inconsistencies between the different levels are difficult to cope with (for example, see Uitermark, 2001). Another disadvantage of multiple representations data structures is that they only support a limited number of levels of detail. Therefore, this list was extended by Van Oosterom (1995) with another approach:

On-the-fly generalization this approach does not duplicate geographic data, but stores information on the generalization process in a data structure, so that a temporary generalization can be derived on-the-fly from the information stored in the data structure. These data structures are termed reactive data structures (Van Oosterom, 1990; Van Oosterom and Schenkelaars, 1995). See page 1 for a definition of the term 'reactive data structure'.

2.2.2 How generalization takes place

Independent from the approach chosen, model generalization can be implemented by generalization operators (McMaster and Shea, 1992, chapter 3). Objects will have to be transformed to fit another level of detail. Therefore, transformations are needed. These transformations can be described with conceptual operators. These operators either work on the thematic, i.e. non-spatial, attributes of the objects or on the geometry of the objects. Example operators are shown in table 2.1. To make the operators work, an implementation is needed; this is a (mathematical) algorithm, that takes care of the required transformation. Some problems exist with these operators. The amount of time needed, especially with bigger datasets, is quite large:

1. to make decisions which conceptual operators to use, when and on what objects;
2. to perform the transformation itself. For example, transformation of geometry takes quite some processing time when there are a lot of coordinates stored with an object and also the surrounding objects need to be changed.

This makes it impossible to derive a lower level of detail in real time with current hardware.

2.2.3 Additional needs for model generalization

Scale, map purpose and intended use are still incentives for generalization. However, with all rapid Internet and (wireless) networking developments, the way people work with geographic data and GISs is changing and this brings some additional requirements for generalization. Focus is shifting from stand-alone stored datasets and stand-alone GIS workstations to an integrated Service Oriented Architecture with the advent of Geo-Information Infrastructures (GII), where data users, data brokers, and data producers are connected via computer networks (Van Oosterom et al., 2000; Thewessen and Brentjens, 2005).

In a GII, geographic data will still be stored in databases and managed with database management systems, but access to the datasets is also granted to remote clients via web services. To work in practice, these web services have to be standardized and interoperable (i.e. there exists a need for vendor independence). Then, such a network infrastructure makes it possible to combine data from different remote sources in real time,

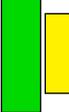
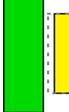
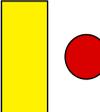
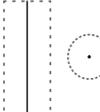
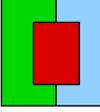
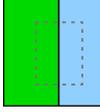
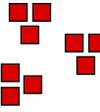
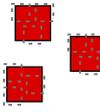
Reclassification	Changes the category an object belongs to and possibly combines it with neighbouring objects of the same class		
Aggregation	Combines an object with other objects of the same or a similar class to a new object		
Displacement	Denotes the movement of an entire object; its shape remains unchanged		
Enlargement	Denotes a global increase of an object		
Exaggeration	Defines a local increase of an object, its shape is distorted		
Collapse	Type of geometry is changed, possible changes are a polygon to a line or to a point and a line to a point		
Elimination	Removes an object from the data set, the freed space is assigned to other categories		
Simplification	Reduces the granularity of an outline		
Smoothing	Improves the visual appearance of an object's outline		
Typification	Reduces the complexity of a group of objects by removing, displacing, enlarging and aggregating single objects, maintaining the typical object arrangement		

Table 2.1: Different generalization operators, after Galanda (2003). Generalization operators take care of transforming objects from a source level of detail (LOD) to a target LOD.

improving availability, accessibility and usability of geographical data. Wireless networks will even make ubiquitous computing possible, making it possible for Location Based Services to become a success (cf. Porcino and Hirt, 2003; Greenfield, 2006).

In a network infrastructure it is convenient to have progressive data transfer. With progressive data transfer, it is possible to get a quick, coarse overview first, which can be refined later on. If accidentally a wrong area is requested, the user can quickly respond to his mistake and data to be send over the network is limited, because a coarse overview is sent first. Furthermore, progressive transfer allows re-use of already sent data. For raster data, several implementations of progressive transfer exist (e.g. based on raster pyramids or wavelet compression principles). For vector data, it is more difficult to obtain the effect of progressive transfer. Although multi-representation data structures store links together with the objects at different levels of detail, they can not help with progressive transfer, as for each level a (different) graphical representation of the objects may exist. If progressive transfer is to be realized with vector data, this calls for explicit relations between the graphical representations of the objects at different LODs.

Another requirement in a GII that calls for generalization, is the fact that each geographic data set might have a different LOD. Nowadays, within national mapping organisations for example, it is common practice to have some predefined levels of detail available, like 1:10,000, 1:25,000, 1:50,000 and 1: 250,000 (see Stoter et al., 2004). However, this limited number of levels of detail, may not be sufficient for the end user whose application is demanding a level of detail that is not available. The absence of a variable number of LODs causes problems for the user to combine different data sets for his application in real time. One possible approach to make it possible to integrate the datasets, is to harmonize the levels of detail of the different sets to the required level of detail: generalization could take place to harmonize the levels of detail. However, as stated in section 2.2.2, it is not possible to perform this generalization in real time with the existing generalization methods and hardware, due to complexity of the algorithms and decisions to be made on how to generalize.

From these observations, some requirements for having generalization available within a GII, near the data sources, can be derived:

1. Delivery of a generalized dataset has to be possible in real time;
2. It is convenient to have progressive data transfer for vector data available in a GII;
3. Availability of data sets with a variable number of LODs at a data source will prevent problems for a user to combine different data sets in real time.

A MRDB, filled with batch or interactive generalization techniques, is not the best viable alternative for use in a GII, as it cannot fulfill all three criteria. As the MRDB approach creates more data sets with different levels of detail, it can fulfill the first criterion of delivering a generalized dataset in real time: it will serve a different data set with another level of detail. In this way, a MRDB is already an improvement over only having source data available. However, progressive data transfer is not possible with multi-representation data structures, because independent graphic representations are

stored. This approach also fails to fulfill the third criterion, because no variable number of LODs is available.

On-the-fly generalization realized with reactive data structures, can fulfill all requirements, because the information is stored in such a way, that a temporal generalization with a variable level of detail can be derived from the stored information in real time. The stored information in a reactive data structure also enables progressively refinement of sent geographic data. To accommodate the three requirements above, this approach is thus best suited for use in a GII.

The reactive data structures have been the subject of earlier research. The data structures were either using redundancy with respect to geometry (see Van Oosterom and Schenkelaars, 1995), or when topology was used (to avoid redundancy of stored boundaries) this was limited: again redundancy was introduced when edges were merged (MSc thesis of Vermeij, 2003). A new idea was published by Van Oosterom (2005) to use a full topological data structure for storing the data, called the tGAP structure. This way redundancy with respect to geometry is avoided. Another thing introduced was the storage of generalization information for the edges within the data structure (with the use of Binary Line Generalization trees, see section 2.4.2), so that with reduction of the amount of objects (selection), the appearance of the objects can also be simplified (simplification).

2.3 Topology

The theory described in this section is loosely based on a description of topology given in (Quak et al., 2003). Vendors have integrated spatial data types in their DBMS products, more or less according to the specifications from (Open Geospatial Consortium, 2005). The Open Geospatial Consortium (OGC) is an international industry consortium of companies, government agencies, and universities participating in a consensus process to develop publicly available geo-processing specifications. According to the specifications, a spatial object can be represented by two different logical vector models within a DBMS:

1. with geometry (described in the simple feature specifications) or;
2. with topology (described in the complex feature specifications).

Both structures have their own strengths and weaknesses. Geometry provides direct access to coordinates of the objects, while topology provides information on spatial relationships. Because geometry provides direct access to coordinates, shared edges and shared nodes of polygons in a planar area partition are stored twice, which causes redundancy. The topological approach avoids redundancy by using references to uniquely identified edges and nodes.

An area object in a topological planar area partition is represented by a topological face, which consists of directed rings. Each ring consists of a set of 1 or more edges and 1 or more nodes. Each face has at least one outer ring (oriented counter clockwise) and

zero or more inner rings (oriented clockwise). The outer ring specifies the boundary of the face. Inner rings can be used to store holes in the area objects. For a topological face to be valid according to the OGC specifications for SQL (Open Geospatial Consortium, 2005) as described in (Van Oosterom et al., 2003), no point on the same ring can be equal to another point defining that ring. In addition, an inner ring is allowed to touch the outer ring in at most one point.

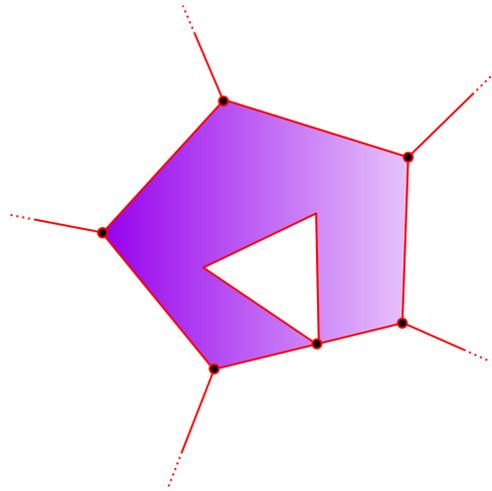


Figure 2.3: A valid polygon: the interior ring is allowed to touch the outer ring in one point

For storing topological information inside an object-relational DBMS some tables in which the information on the faces, edges and nodes will be stored, have to be created. At least three tables are needed in a relational implementation: a face table, an edge table and a node table. The exact information in the tables is dependent on which topological model is chosen, see section 2.3.1– 2.3.3. Because the geometry of the faces is implicitly stored via references, also a function has to be implemented in the database, to materialize the geometry of the faces, based on the information about the nodes, the edges and the references. This function is dependent on the information stored in the tables and the implementation of the function differs per model.

The remainder of this section describes three possible ways of storing and reconstructing topological faces with their advantages and disadvantages. These methods are also tested for their performance and difficulties for the use of the Generalized Area Partition (GAP) face tree, as described in section 4.1.1.

2.3.1 Left right topology without edge references

Left right topology is based on storing left and right information about faces for each directed edge in the planar area partition. For each directed edge in the structure is stored which face is on its left and which face is on its right side. Based on this information a collection of edges can be retrieved that belongs to one face. With this retrieved

subset, a face can be reconstructed. All edges are directed and each edge has a start and an end node. Based on this node information temporary rings can be reconstructed, by 'glueing' edges together. Searching is needed to derive which pair of edges can be glued.

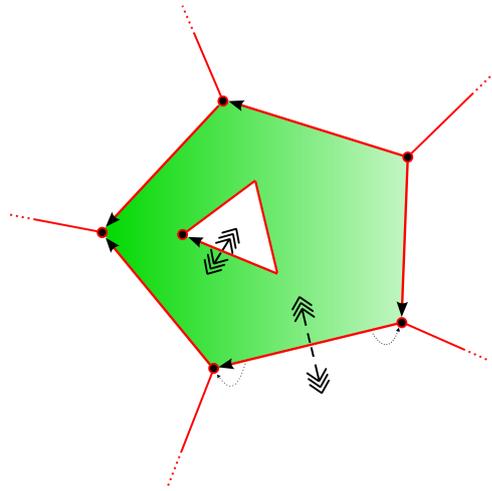


Figure 2.4: Left right topology without start edge references

When temporary rings are formed, the coordinates of the temporary rings also have to be oriented in the way the conventions prescribe, i.e. outer ring counter clockwise, inner rings clockwise. This orientation can be inferred from calculating the signed area per temporary ring. Given that each ring contains n points (x_i, y_i) , $i = 0, \dots, n$, with $x_0 = x_n$ and $y_0 = y_n$, this signed area value can be calculated with the following formula (via E.F. Glynn II, 2005):

$$A_{signed} = \frac{1}{2} \sum_{i=0}^{n-1} a_i \text{ where } a_i = x_i y_{i+1} - x_{i+1} y_i$$

After this, a decision has to be made, which temporary ring is appointed as the outer ring and which temporary rings are inner rings. This decision can be based on the size, i.e. the absolute value of the signed area, of the temporary rings. The largest ring must be the outer ring.

Polygons with an inner ring touching an outer ring as shown in figure 2.3 are automatically reconstructed correctly, because the reconstruction process will form a temporary ring of the inner ring, which is touching the outer ring.

An advantage of this method is that it does not store much references. A disadvantage of this method is that it is not possible to decide which temporary rings are inner rings without a geometrical calculation, the signed area value.

2.3.2 Left right topology with edge references

This kind of left right topology looks a lot like the model presented in the previous section. It is again based on storing left and right information about faces for each directed edge in the planar area partition.

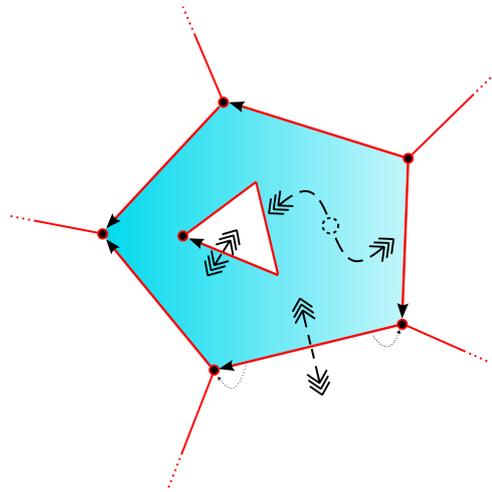


Figure 2.5: Left right topology with start edge references

Based on the left right information stored with the edges a collection of edges can be retrieved that belongs to one face. With this retrieved subset, a face can be reconstructed. All edges are directed and each edge has a start and an end node. Based on this node information temporary rings can be reconstructed.

In addition, for each face references to its outer ring and all its inner rings are stored explicitly. Information is thus available on which edge starts which temporary ring, outer or inner, and in which direction, therefore the references have a signed value. This way, the temporary rings can be oriented correctly and it is not needed to compute the signed area value for all temporary rings.

Polygons with an inner ring touching to an outer ring as shown in figure 2.3 are automatically reconstructed correctly, because the reconstruction process is again based on node information.

An advantage of this method is, that it is possible to reconstruct the rings, without any geometrical calculation. A disadvantage of this method is having to store the additional starting references and signs per face. These signs and references need to be calculated, while storing the topology.

2.3.3 Winged edge topology

Winged edge topology is based on storing at least two signed references per directed edge. Left right information should be stored with the edges. Via the signed edge

references (next left and next right edge), rings can be formed. To know which ring is an outer ring and which ring is an inner ring, for each face also the start edge for the outer ring and start edges for all inner rings are stored together with their direction.

For reconstructing the rings, after retrieval of all edges that belong to one face (based on left right information), one starts at the given start edge, then considering, dependent on the sign of the reference to the next edge, the next left or next right edge will be followed. This process continues until the ring is complete and one arrives at the start edge again. This is done also for all inner rings given in the list with inner rings.

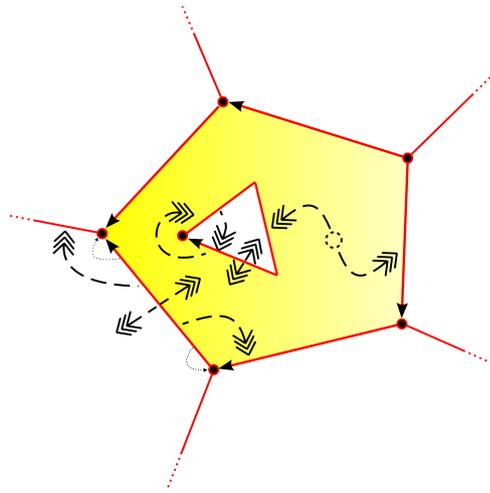


Figure 2.6: Winged Edge topology

Polygons with an inner ring touching to an outer ring as shown in figure 2.3 are not automatically reconstructed correctly. While following the signed edge references, the reconstruction process will mix the outer and the inner ring in this case, because via the references one can end up from the outer ring on the inner ring, or from the inner ring on the outer ring. Only by taking the node information of the edges into account, or by processing the reconstructed geometry of the polygon, this situation can be detected and resolved.

An advantage of this structure is that rings are quite easy to reconstruct, just follow references. A disadvantage of this structure is that it uses a lot of references, which all need to be correct for reconstruction. Second, the detection of touching inner and outer rings is quite difficult. This is only possible via node references or processing the geometry of the polygon formed.

2.4 Reactive data structures for generalization

In this section a description of the tGAP data structure used for generalization is given. This information is a concise summary of the conceptual model given in (Van Oost-

erom, 2005). An in-depth description of how the tGAP structure works, follows in sections 4.1.5 and 4.1.6. Generalization of faces is realized, in this topological reactive data structure, by leaving out unimportant faces for a lower level of detail, assigning their areas to other faces and simplifying the edges bounding the faces.

First, how the faces are stored and how selection can be done for the faces is explained (2.4.1). Second in section 2.4.2, an explanation is given on how to store information to make it possible to also simplify the edges in the structure. Third, how the edges are related to the generalized faces is described (2.4.3).

2.4.1 Generalized Area Partition face tree

The topological Generalized Area Partition (GAP) face tree is a modification of an older concept used for generalization of geometrically stored data: The GAP tree, as described in (Van Oosterom and Schenkelaars, 1995).

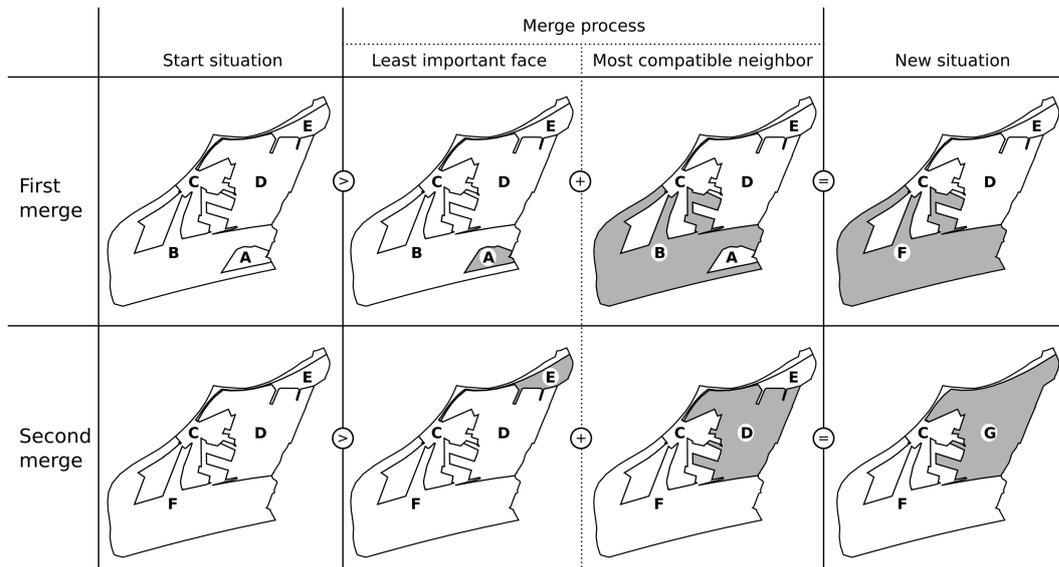


Figure 2.7: Merge steps for construction of the GAP face tree

The GAP face tree is a tree structure that stores information on generalization of topological faces, i.e. making the selection of faces possible, that are important enough for a certain level of detail. Because a planar area partition is used (i.e. no gaps nor overlaps are allowed in the two dimensional domain), it is not possible to just remove faces for a lower level of detail, as this would leave gaps in the area partition. Therefore, on higher importance levels, the areas belonging to the least important faces are assigned to their most important and most compatible neighboring faces. For deciding which face will be removed and assigned to a neighboring face, both an importance and a collapse function are needed. The importance function will return the least important face in the

complete area. With the collapse function, it is decided to which neighboring face the area of the face will be assigned. Three possibilities exist for assigning a thematic value to the new face:

1. Assign the thematic value of the largest face to the new face (a 'deciduous forest'-classified small face and a large 'coniferous forest' classified face will become a new face, classified 'coniferous forest');
2. Assign a mix of thematic values of both faces to the new face (a 'deciduous forest'-classified small face and a 'large coniferous forest' classified face will become a face, classified 'deciduous and coniferous forest');
3. Assign a new thematic value, based on the thematic value of both faces, to the new face (a small 'deciduous forest'-classified face and a large 'coniferous forest' classified face will become a face, classified 'forest').

Although the process of assigning thematic values is an important issue (cf. Van Smaalen, 2003), it is considered out of scope for this research.

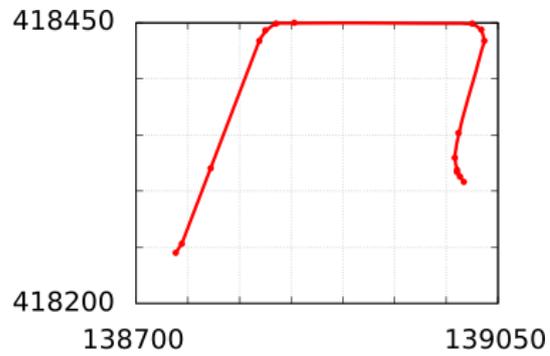
Within the GAP face tree, each face has an importance range assigned. The importance range consists of two stored values that define a range: low importance and high importance. Based on the intersection of an importance level with the importance ranges of the faces, the faces that are valid for that importance level can be selected. Via this selection the amount of faces can be reduced, because less faces reside higher in the tree.

2.4.2 Edge simplification: Binary Line Generalization tree

So far, one part of generalization can be accommodated with the data structures, i.e. a selection can be made to show less faces on lower levels of detail. For simplification of the geometry of the faces, other data structures are needed. Each topological face consists of a collection of edges and nodes that form the face. To simplify the appearance of the face, a simplification of the geometry of the edges is needed. With this simplification, the amount of coordinates associated with the edges is reduced.

Topology offers an advantage for this edge simplification, because in case of an area partition, with a geometrical approach, each edge resides two times in the dataset. With a topological approach, each edge is only stored once, and thus has to be simplified only once. Also neighboring objects do 'fit' in the partition after generalization, so that no gaps or overlaps will exist. An example of an edge that can be simplified is shown in figure 2.8

For the simplification of edges, one can use an algorithm as described in (Douglas and Peucker, 1973). In figure 2.9 the methodology of simplification is illustrated by showing the first two iteration steps of the algorithm for the edge from figure 2.8. This algorithm starts by taking a rough initial estimation of an edge, by connecting the start and end point of the edge. For all remaining vertices of the edge (intermediate points) it is tested which vertex is furthest away from this rough estimation (edge segment). This



(a) Edge

	X	Y
1	139,017.616	418,308.816
2	139,013.207	418,313.918
3	139,010.867	418,317.589
4	139,010.165	418,319.385
5	139,008.140	418,330.509
6	139,012.732	418,352.465
7	139,037.027	418,434.001
8	139,034.531	418,444.289
9	139,025.650	418,449.119
10	138,853.779	418,450.181
11	138,835.343	418,449.081
12	138,825.757	418,443.312
13	138,819.850	418,434.052
14	138,772.031	418,320.513
15	138,744.303	418,253.589
16	138,738.741	418,245.250

(b) Coordinates

Figure 2.8: Edge and its coordinates

distance can be used as a threshold value, if the vertex needs to be taken into account for a certain level of detail of the edge. The furthest vertex in the first iteration step is taken into account with a new guess of the simplified polyline. This guess then consist of two edge segments, for which the distances to the remaining vertices can be calculated. The furthest vertices are taken into account for the next iteration step. Using recursion, this process continues, until all vertices have a distance assigned.

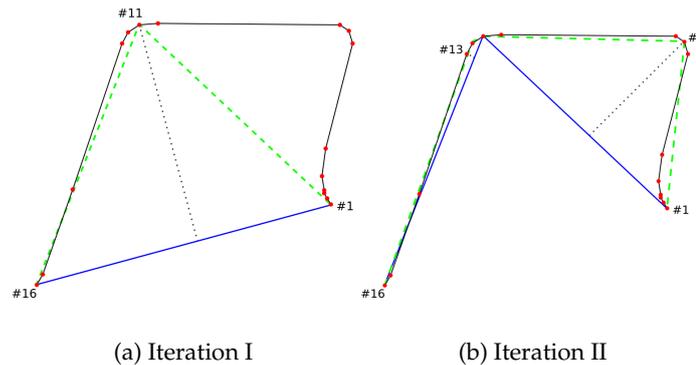


Figure 2.9: First two iterations while using the algorithm of Douglas-Peucker for edge simplification

The algorithm is calculation intensive, because the distances to the vertices have to be calculated in a recursive function and differ in each iteration step. As described in (Van Oosterom, 1990) it is possible to store the results of this line simplification in a tree structure. This is a binary tree structure that is called the Binary Line Generalization (BLG) tree. The result of the Douglas-Peucker algorithm as illustrated in figure 2.9, can be mapped to the (top of the) BLG tree, which is shown in figure 2.10. The values accompanying the coordinates, drawn in orange, are the threshold values.

If this tree structure is used, one can query the tree to get the right amount of edge detail by using the threshold values stored with the vertices, instead of having to calculate all distances over and over again.

Selection of detail is based on the vertex tolerances from the Douglas Peucker algorithm that are stored in the BLG tree. Vertices having larger threshold values are mostly stored closer to the root of the tree (i.e. higher in the tree). In general, a vertex residing lower in the tree has the same or a smaller tolerance than its parent in the tree. To get a particular amount of detail, the tree is descended, until the wanted detail, given by the threshold value, is reached and one can stop with descending the tree in that direction.

2.4.3 Generalized Area Partition edge forest

Edges in the structure can be simplified, with the use of the BLG tree. But another problem has to be solved: With each merging step of faces in the GAP face tree, the

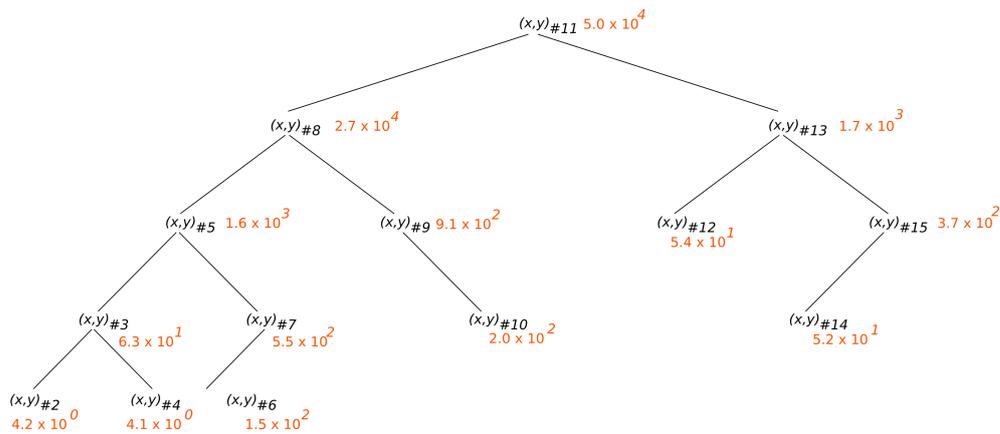


Figure 2.10: BLG tree, that belongs to the edge shown in figure 2.8. The top node of the tree contains coordinate #11 with the calculated threshold, as this coordinate is the one that is furthest away in the first iteration step of the Douglas-Peucker algorithm (as illustrated in figure 2.9). The two nodes below the top of the tree, contain the coordinates that are appointed in the second iteration of the Douglas-Peucker algorithm, as being furthest away (coordinate #8 and #13).

boundary edge(s) between two faces that are merged, will be removed. This removal may leave two edges with one node as a junction, where this junction node then has an incidence relationship with only the two edges that are still left. In a topological structure these two edges should become one edge, because topological nodes normally exist in outer rings when their incidence relation is 3, or more (apart from isles formed by one node and one edge). The two edges should be joined to form a new edge, also enabling simplification for the joined pair. By joining the edges into a new edge, the geometry would be duplicated. Therefore, if the two edges their corresponding BLG trees are joined by making a reference to the two old BLG trees, it is possible to make the edge data structure without geometrical redundancy. However, for the junction node then no information is available on how to simplify the new edge, as opposed to the rest of the vertices, which all have a threshold value assigned (stored in the BLG tree). This threshold value has to be computed when building the GAP face tree, just as is the case in the first initial step of the Douglas-Peucker algorithm. This process of joining edges takes place with building the GAP face tree. Selection of how much detail is needed, can be done in the same way as with the original BLG trees, although the accuracy measure of the junction node is a worst case estimate value (see Van Oosterom, 1990, section 5.4).

Joining of the edges will extend the existing BLG trees into larger trees. However, because boundary edges will be removed, multiple root nodes, starting multiple trees, will exist for the edge data structures after building the GAP face tree. This edge data structure is therefore called, the GAP edge forest.

2.5 Advantages the data structures offer

Although the data structures do not solve all problems associated with generalization, it is possible to tackle some problems of the generalization process. The advantages the data structures offer will be described in this section.

2.5.1 Increasing the use of framework data

As argued by Stoter et al. (2004), large scale data, also termed framework data, is expensive to gather and collect from the field. The use of framework data could be enlarged, if from the large scale datasets, smaller scale datasets could be derived automatically. To reach this objective, large scale data should be quick and good to generalize. However, currently in the field of map generalization research, a lot of islands exists, without an overall solution (Stoter et al., 2004). One of the fundamental problems is the absence of data structures suitable for storing generalization results. This research implements such data structures in a geo-DBMS. This allows the storage of the results of the generalization process, and the results are then usable for real time use. All in all, having these data structures available for storing results of the generalization process can be a stimulus to increase the use of framework data.

2.5.2 Real time need for generalization

Galanda (2003) remarks that “[g]eneralization is a pre-requisite to adaptive mapping with respect to scale and user’s position within LBS”. This is foremost due to the fact, that users want quick responses on their mobile devices. Generalization makes it possible to reduce the amount of data to be send and can guarantee short loading times, while the cartographic quality of the maps remains high.

The data structures solve the problem of the amount of time needed to perform generalization. As mentioned in section 2.2.2, the amount of time needed to make decisions on which conceptual operators to use and how to perform a transformation from one scale to another costs quite a lot of time; Generalization is not possible in real time. Therefore, smart storage of the results of the generalization process is a solution to tackle this problem. By adding a pre-processing step and storing the result in the data structures, this takes away the barrier from generalization that it should be needed to do the transformation calculations in real time.

2.5.3 Continuous range of levels of detail

It is necessary in a GII to accomodate the requests of the users: the scale of required geographic data heavily depends on the users application. Currently the waterfall-like process creates multiple resolution databases, while in a GII variable scale interaction is possible by the interfaces defined in the standards (e.g. WMS and WFS specification by OGC). To accomodate this need for datasets available at a continuous range of LODs, generalization and suitable data structures are needed. The data structures described in

this research can indeed accommodate users with a continuous range of level of details, instead of a few discrete levels, created in advance.

2.5.4 Non-redundant topological storage

The data structures are based on topology. Because topology is used, it is possible to make the storage of geometry of the geographic information in the data structures non-redundant (at least, with the conceptual operators used in this research, see section 4.2.1). This is accomplished by only storing new information on how the generalization process takes place.

2.6 Prerequisites for the data structures

The data structures can be implemented in a DBMS, but there are some prerequisites to be able to do this. This section describes the requirements for implementation of the data structures in a mainstream DBMS.

2.6.1 Requirements with respect to the face information

At first, the DBMS needs a way of storing the spatial attributes of the objects in a topological manner. This means a need for simple feature geometry for the edges and for the nodes, together with references. For the reconstruction separate functionality is needed and is dependent on what type of topology is stored (see section 2.3). Based on the references the geometry of the faces can be inferred.

Because a separate reconstruction algorithm is needed, it is also needed to have a programming language available. This language must be either embedded in the database or it must have bindings to the database, so that SQL commands can be fed to the database and answers can be retrieved.

For quickly finding the edges belonging to a face, it is needed to create an index on the references of the faces pointing to the edges. For filling the data structure it is best not to have a lot of indices, because this means that all indices also need to be updated when a new piece of information is entered within the database.

To guarantee consistency in the dataset, it is also good to have a way to create constraints, like foreign keys, stating that references from one table to another exist. This is of great help for checking if the topological area partition is correct, i.e. all pointers stored with the edges to faces and nodes are pointing to existent face and node objects.

Another example of consistency is the need for unique identifiers for the faces. The faces are stored in their own table. Because the edges are joined to the faces, for reconstruction of the faces it is needed to have a unique identifier. Besides the selection of the faces on id, a more important selection has to be performed: spatial selection. This spatial selection is based on the geometry of the faces, but also on the importance range the faces have. The minimum bounding rectangle (MBR) is needed, to support spatial

selection of the faces. Therefore, also a function for calculation of the MBR needs to be available.

The proposed manner to do the selection is by using a 3 dimensional spatial index based on a function, a so called functional index.

The construction process is based on the size of the topological faces. This area measure might be calculated on the fly, although this means reconstructing the geometry of the faces, but at least it means that a spatial function for calculation of area needs to be available. Together with the area value, it is necessary to have low importance and high importance fields in the table, see page 21.

2.6.2 Requirements with respect to the edge information

The edge table is an intermediary for having edges represented by BLG trees. To store the BLG tree in the database, it is necessary to have a possibility to create custom data types. Because it is a custom data type, the filling needs to happen in a little bit different way, than is possible with the normal data manipulation language. An algorithm is needed to transform edge geometry into generalization information of the edges stored in the BLG trees.

To summarize, the features needed for the data structures are as follows: it must be possible to create tables to store simple features, i.e. spatial data types must be available. Indices for normal and spatial data must be present in the DBMS. It must be possible to create integrity constraints on the data and a programming language to create application logic must be available. If functional indices are available, preferably in 3D, they can be used, but they are not a hard requirement. A hard requirement is that the DBMS is extensible with new data types (to create the BLG tree datatype). All in all, these features were found available in two DBMSs: Oracle, with Spatial Cartridge extension, and PostgreSQL, together with a geographical data plug in, called PostGIS.

2.7 Available DBMSs suitable for implementation

The candidates for selecting a suitable DBMS are Oracle (with the Spatial Cartridge) and PostgreSQL (with PostGIS), because they both can offer (nearly all) functionality described above.

In the research Oracle was chosen, because it looked most promising on having a 3D index for spatial data available, while PostGIS is lacking such functionality. Another advantage, that made the choice for Oracle, was that there was a fully tuned environment available to develop the data structures and a lot of knowledge on how the Oracle DBMS works is available at the research institute where this research has been performed.

2.8 Final remarks

This chapter showed the backgrounds of GIS and generalization, and introduced topological reactive data structures as a possible solution for solving the problems related to generalization. Before the methods used in the implementation and testing phase of the research are justified, the following remarks can be made:

- With all developments for a Geographic Information Infrastructure (GII) in the last decade, generalization remains a key issue in the data collection and map making process. Model generalization gets even more relevant for data reduction, before data is sent over the network.
- Although a lot of research has been performed on the topic of generalization, it still remains a time intensive operation and cannot be performed in real time in a GII during interaction of a user with a data source.
- Reactive data structures, that have been proposed, take another approach, by taking away the need to apply generalization operators in real time, but storing results of these operators in a smart way, i.e. in data structures without redundant geometry (as is the case in multiple representation databases).
- If reactive data structures, like the GAP face tree and GAP edge forest, are used, it is possible to create an on-the-fly generalization. The reactive data structures also enable progressive transfer of vector data. Therefore this approach is most suitable for implementation in a GII.
- To implement these data structures in a mainstream database management system, there is a need for creating tables, integrity constraints and (spatial) indices. Spatial data types must be present in the DBMS. A programming language and a way to extend the database with a new data type should also be available. Having functional indices and 3D indexing available would be an advantage performance wise, but are more a preference than a functional requirement.

Chapter 3

Methodological justification

The new (combination of) data structures, described in (Van Oosterom, 2005) is a model for the storage of geographical information in a computer system in a way that allows generalization with a continuous range of detail levels. The quest for this new model is, amongst other things, driven by societal needs (cf. section 2.2.3). An indication of how an implementation of this new model in an OR-DBMS should be created, is also given in (Van Oosterom, 2005). As is written in the conclusions of the paper, it is necessary to test and experiment with the new model.

This thesis research conducts a prototype implementation and performs benchmarks and tests to check the behavior of the new data structures. This chapter justifies the methods used in this research by reviewing the need for experiments in computer and geographic information science.

3.1 Need for experiments

While Tichy (1998) quotes (Ralston and Reilly, 1993), he observes that:

“... the primary subjects of inquiry in computer science are not merely computers, but information and information processes. Computers play a dominant role because they make information processes easier to model and observe.”

Also geographic information science deals with information and information processes. Mark (2003) quotes (Shuman, 1992), who writes:

“Information science is very difficult to define. ... the field of information science, however, may be defined as one that investigates the properties and behavior of information, how it is transferred from one mind to another, and optimal means for making that transfer, in both natural and artificial systems. Finally, information science is concerned with the effects of information on people and on machines.”

Further, Mark reasons that when the word ‘geographic’ is inserted in front of ‘information’ in the given definition, that:

“this appears to be a reasonable definition for our field, and the contention that Geographic Information Science is, fundamentally, a branch of information science, seems quite tenable.”

He also recognizes in a definition of geographic information science, given in (Mark, 2000), that the field “also overlaps with and draws from more specialized research fields such as computer science. . . .”

The notion by Tichy that for computer science “[e]xperimentation is central to the scientific process” can thus also be applied to geographic information science: “Only experiments test theories. Only experiments can explore critical factors and bring new phenomena to light so theories can be formulated in the first place.”

According to Tichy, conducting experiments in computer science has some “principal benefits:

1. Experiment can help build up a reliable base of knowledge and thus reduce uncertainty about which theories, methods and tools are adequate.
2. Observation and experiment can lead to new, useful and unexpected insights and open up whole new areas of investigation . . .
3. Experimentation can accelerate progress by quickly eliminating fruitless approaches, erroneous assumptions, and fads.” (Tichy, 1998)

3.2 Implementation experiments

The goal of this research is to verify the formulated theory in (Van Oosterom, 2005) and test the performance of the tGAP structure. However, with an experiment it is not possible to completely validate the theory:

“To paraphrase Dijkstra, an experiment can only show the presence of bugs in a theory, not their absence.” (Tichy, 1998, p. 2).

Although that it is not possible to prove that the theory is completely ‘watertight’, by finding flaws the theory might be improved. This is one of the foremost reasons to adopt the methodology of an experiment in this research. Besides finding flaws, there is another reason to carry out an experiment.

A claim is made in the paper that this new model is implementable and that it should work within an object relational database management system (OR-DBMS). If an implementation succeeds, then this claim is valid. Another reason to create an implementation is that:

“...experiments help with induction: deriving theories from observation.” (Tichy, 1998, p. 2)

An implementation can be a vehicle for reasoning for new theories on how to deal with variable scale data sets.

3.3 Benchmarks

During the implementation stage, some choices had to be made in which direction to continue. To supply additional evidence that a certain method was a good one, some benchmarks were carried out.

“A benchmark provides a level playing field for competing ideas, and assuming the benchmark is sufficiently representative, it allows repeatable and objective comparisons. At the very least, a benchmark can quickly eliminate unpromising approaches and exaggerated claims.” (Tichy, 1998)

Benchmarking is not only desirable during the implementation, it is also useful for comparing the chosen model with other approaches. To evaluate the storage and processing time requirements of the data structure the time needed to build the data structure and the size on disk used by the data structures can be measured. These measurements then can be compared with other approaches and models that serve the same goal: storing geographical information with multiple levels of detail. Benchmarking thus can help in a debate on which direction to go with implementing generalization in a DBMS.

3.4 Final remarks

An implementation experiment and benchmarks are appropriate means to reach the goal of this research, because:

1. flaws in the new theoretical models can be found;
2. the claim that an implementation should be possible can be validated;
3. experimenting can be a vehicle for reasoning for new theories and;
4. built experience (‘feeling’) that can be used to further improve the theory;
5. benchmarks deliver information that allow comparison of these models with other approaches for storing geographic information with multiple levels of detail.

Chapter 4

Implementation and testing of the tGAP structure

This chapter gives an overview of the implementation and testing experiments. The GAP-face tree and GAP-edge forest, for short the tGAP structure, are implemented in a mainstream Object-Relational Database Management System. This validates the claim that the tGAP structure is implementable in a mainstream DBMS. Flaws in the theoretical models might be found during implementation and testing. Another goal of testing the tGAP structure is to deliver information, in order to compare this approach of storing geographical data in variable scale manner, with other approaches.

Section 4.1 gives an overview of the implementation in the DBMS, Oracle Spatial. Section 4.2 deals with the testing results. It shows results on the storage and time requirements for the tGAP structure to fill and use. Section 4.3 describes some theoretical changes to the tGAP structure and it shows relevant literature, from which future research directions can be inferred. The results of the implementation and testing phase are summarized in section 4.4.

4.1 Implementation

This section shows the implementation of the tGAP structure. Section 4.1.1 describes topology testing, because the choice which kind of topology will be used has some implications on the implementation of the tGAP structure. After this choice has been made, the tGAP structure can be implemented and filled with data: figure 4.1 shows the steps that are needed, before the structure is complete. First, it is needed to select data, which already is in a topological model and bring it into the physical model needed for the tGAP structure. Therefore, section 4.1.2 gives an overview on which topological data is selected for the implementation and section 4.1.3 describes the logical and physical data models for the source topological data, as well as for the complete tGAP structure. Then, it is also needed to build the BLG trees for the edges in the tGAP structure; this is described in section 4.1.4. Section 4.1.5 and 4.1.6 describe the construction of the GAP

face tree and GAP edge structure. In 4.1.7, a description is given on how the filled tGAP structure can be used in a networked environment, with Google Earth as frontend, to visualize the content of the data structures. Section 4.1.8 summarizes the lessons learnt from the implementation.

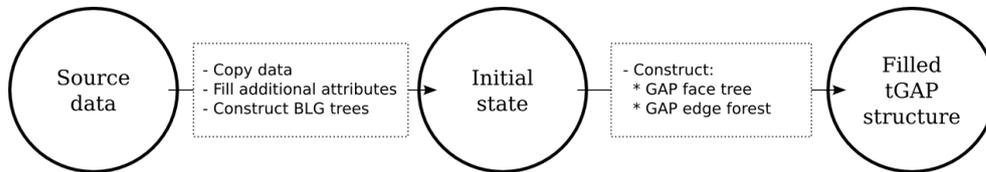


Figure 4.1: Overview of constructing the tGAP structure. First, topological data is loaded into the DBMS and mapped so that it is in an initial state from where the construction of the tGAP structure can start. Second, the GAP face tree and GAP edge forest is constructed.

4.1.1 Topological testing

The data structures used for storing generalized geographical information are topologically structured. To store topology in a DBMS, some proprietary software packages are available, like Radius Topology and Oracle Topology. However, these packages do not suffice for creating the data structures, because there are some special requirements for the edges in the data structures, i.e. joining of Binary Line Generalization trees and associating an importance range with all edges (see figure 4.6, page 41). These extra features cannot be integrated in the standard topology engines. Therefore, three different topology reconstruction algorithms have been implemented, so the requirements for the GAP face tree could be satisfied. After implementation, these algorithms have been benchmarked to support a decision on which kind of topology is best to use with the data structures.

For benchmarking the three algorithms based on topology models from section 2.3, in which left right topology and winged edge topology models are described, are implemented. The algorithms are based on existing source code, either in Java or in PL/SQL, the procedural programming language available in Oracle. To make a good comparison, all algorithms were implemented in PL/SQL. For different areas in the test dataset, as shown in figure 4.3, the time needed to reconstruct the geometry of the faces is measured.

As can be observed in figure 4.2 the winged edge algorithm needs some more time to return the faces than both left right topology algorithms. That this winged edge implementation is slower, may be explained by the fact that this algorithm uses independent queries, while both algorithms that use left and right face information can fire one bulk collect statement to the database, in which all edges belonging to a face are retrieved and stored in memory for further processing. The lesson learnt here is that also the winged edge model needs left-right references to make it possible to retrieve all edges

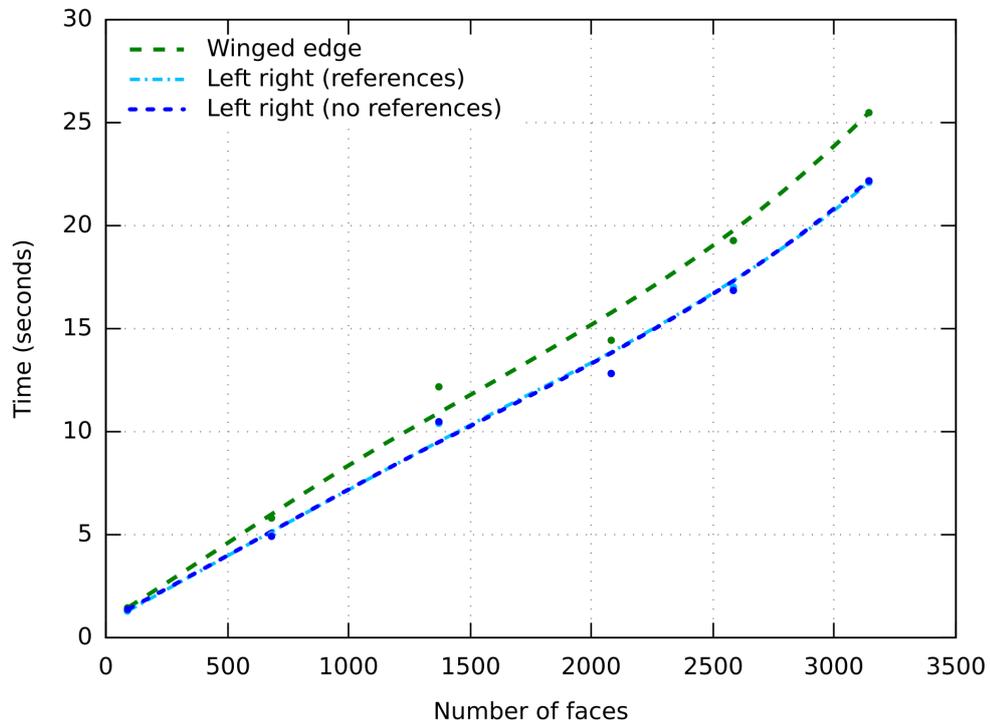


Figure 4.2: Performance of topological face reconstruction

that belong to one face and materialization of the geometry then can be performed in memory.

Further, it is interesting to note that both left right topology algorithms deliver the faces in about the same time. From this can be concluded that storing the extra references for a start edge per ring does not gain performance while reconstructing, but the extra references present in the model, will take more storage space.

The winged edge algorithm needs more references than the left right algorithm. Therefore, also extra information needs to be present and up to date. This is illustrated with an example: In the test data set from the Cadastre a custom object data type, a list stored in a table, is used for storing the start edge references for inner rings and the outer ring. It is difficult to check this object for consistency. While reconstructing with the winged edge algorithm, it appears that some islands are missing, while these islands are present when the faces are reconstructed with the left right algorithm not using start references. Although, the data sets were supposed to be correct, the missing isles are caused by missing start edge information.

Another 'problem' that occurs with using a model that uses references for start edges



Figure 4.3: Test data used for reconstruction of face geometry (areas consist of respectively 91, 679, 1372, 2078, 2586 and 3144 faces)

per ring, is the complexity to handle these references when merging faces (the merge operation is described in section 2.4.1). With a merge operation of two faces it might occur that the common edge that is removed in this operation is the start edge of both outer rings belonging to the original faces. A new start edge then has to be appointed, and a direction has to be given for this new edge. This makes the process of merging more complicated (and not faster as well, because more processing is required). The same problem occurs, when an island list is kept for each face. Due to the merging process, islands can be removed or can come into existence. This also needs to be detected, because for each island ring, an entry in such an island list needs to be stored.

Based on the two observations:

1. that using more references will require more storage space
2. and, more importantly, that using more references adds complexity to the merge operation in the build process of the GAP face tree

the choice is made to implement the left right algorithm without starting references for storing the topological GAP face tree, as opposed to what is done in Van Oosterom (2005), where a winged edge data structure is proposed.

4.1.2 Geographic data selection

To test the tGAP structure, geographic data is needed. In an early stage of the research, an artificial dataset is created (shown in figure 4.4), to make it possible to implement the geometry materialization function of the faces.

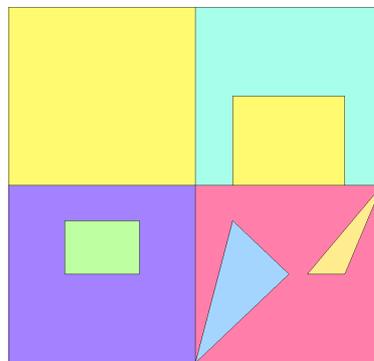


Figure 4.4: Artificial dataset

However, it is difficult to predict all cases that normally exist in a real world data set and therefore it is not easy to create an artificial dataset that accurately reflects the datasets as used in practice. Therefore, real world datasets are selected, as these will have a lot of the cases that are found in practice.

Before the source data sets are brought into the physical model needed for constructing the tGAP structure, it is needed that the source data sets are validated and errors

found should be corrected, otherwise unexpected failures may occur during construction of the tGAP structure.

Available data

Three sets are used for implementing and testing the tGAP structure: two cadastral data sets and one data set with topographic information from the municipality of Amsterdam. Table 4.1 shows the amount of information available in each data set.

	Cadastral (small)	Cadastral (large)	Amsterdam
Faces	161	50238	170368
Edges	499	178815	418530
Nodes	341	129442	281216

Table 4.1: Information in the source topological data sets: amount of nodes, edges and faces that is stored

The cadastral test datasets consist of two area partitions of parcels, as kept by the Dutch Cadastre in their large scale Geographic Information System (in Dutch: Landmeetkundig Kartografisch Informatiesysteem, LKI). The data are stored in meters in the coordinates of the Dutch reference system (Rijksdriehoek-stelsel, or RD). However, in the database, the reference system is not associated with the coordinates, i.e. for the system the coordinates appear to be just numbers.

Another dataset that is used for testing has been supplied by the municipality of Amsterdam. The dataset consists of more data than a topological area partition only, so to use the data as source data for constructing the tGAP structure, the faces, edges and nodes need to be selected from the complete object model that is supplied. This way an area partition is derived¹. The data are stored in the RD system and the coordinate system is associated with the coordinates.

Validation of source data

Both datasets are validated in two ways:

1. Using the validation function provided by the database. The geometry of the individual edges and nodes and the reconstructed face geometry can be validated. This is made possible by the function that the Oracle DBMS offers for validating geometry. This function is called `validate_geometry_with_context()` and takes two parameters: a geometry and an accuracy measure (Oracle dimension information). The most common mistakes that are found when validating have to do with either the orientation of the rings of faces (outer ring oriented counter clock wise, inner ring oriented clock wise), self-intersecting geometry or multiple coordinates

¹This mapping is done with creating views on the original tables and selecting data from these views into tables.

in one geometry that appear to be on the same location (e.g. due to stroking of a circular arc).

2. Another way of validation of topological data is performed with enforcing constraints on the data. With constraints, e.g. foreign keys, the referential integrity is checked; This makes sure that all references in the DBMS do exist and are pointing to existing objects.

That validation is needed, shows the Amsterdam data set. It is coming from the systems on which geographic data is edited and the validation of geometry shows that the data set is not completely clean (mainly due to coordinate accuracies, probably due to the import and export procedures). With validating geometry of the materialized faces the coordinate errors are found and are changed by rounding the coordinates and removing duplicate coordinates in the geometry, automated by a script.

4.1.3 Data models

The source data sets offer more topological references than are needed for left right topology used for the tGAP structure. Therefore, the left right topology model can be derived from all the data sets to be used. To have a way to use different data sets as input for the building process of the GAP face tree, the topological source data is brought into the same logical and physical model.

Logical data model for source topological data

The logical model can be described in UML, the Unified Modelling Language, as shown in figure 4.5. In this UML diagram, all boxes show classes of objects. Each top of a box shows the name of a class to be stored. For faces, edges and nodes information is stored in the DBMS.

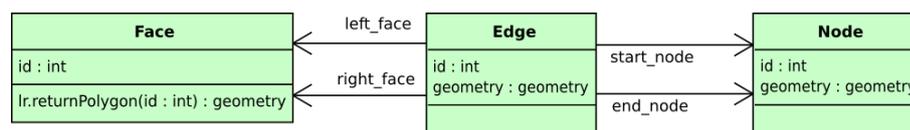


Figure 4.5: UML class diagram showing information to be stored for left right topology

In the center of the boxes all attributes for a class are shown. Which information is to be stored for each class is explicitly mentioned. The lowest part of the box eventually contains behavior of a class. This behavior (logic) has to be implemented in a programming language, like PL/SQL, the procedural language the Oracle database offers. For the left right topology the following information is stored:

Face Per face a unique identifier is stored. To access the geometry of the faces, a function is needed that can materialize the geometry based on the edge and node information.

Edge For each edge a unique identifier and its geometry is stored, together with 4 references: a left face, a right face, a start node and an end node reference.

Node Node information consists of a unique identifier and the node geometry.

Physical data model for source topological data

The UML class diagram can be mapped to a physical model, in which create statements are used to enter the models in the DBMS. All classes from the UML diagram, are mapped to their own table. The result of this mapping is shown in listing 4.1.

Listing 4.1: Source topology: physical data model

```

1  -- description of the face table
2  sql> desc face;
3  name                null?    type
4  -----
5  face_id             not null number
6
7
8  -- description of the edge table
9  sql> desc edge;
10 name                null?    type
11 -----
12 edge_id             not null number
13 left_face_id        number
14 right_face_id       number
15 start_node_id       number
16 end_node_id         number
17 geometry            mdsys.sdo_geometry
18
19
20 -- description of the node table
21 sql> desc node;
22 name                null?    type
23 -----
24 node_id             not null number
25 geometry            mdsys.sdo_geometry

```

Logical data model for tGAP structure

To store the conceptual model described in section 2.4 for the GAP face tree and GAP edge forest in a computer system, the conceptual model needs to be mapped to a logical and, after that, to a physical model.

Figure 4.6 shows an UML class diagram, showing all information that is needed for storage of the data structures in the DBMS:

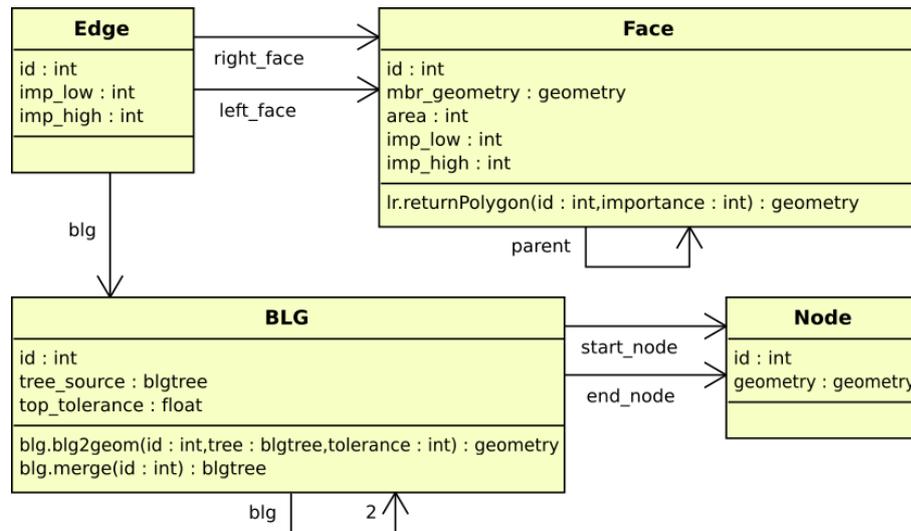


Figure 4.6: UML class diagram showing information to be stored for the GAP face tree and GAP edge forest

Face For the faces a unique identifier, together with the bounding box and the area value and the importance range is stored. Also a function is needed to allow retrieval of geometry of the faces.

Edge The edges get an identifier and an importance range is stored, together with 3 references: to a left face, a right face and a BLG tree.

BLG For the BLG trees a unique identification is stored. The BLG tree class consists either of original BLG tree objects, together with their tree source or of a join of two BLG tree objects, where it is needed to store two references and a top tolerance. Also two functions are required: one to merge BLG trees and another to retrieve geometry from the (merged) BLG trees, with a certain level of detail.

Node For the nodes a unique identifier and the node geometry needs to be stored.

Physical data model for tGAP structure

In this implementation of the GAP face tree 4 tables will exist. Listing 4.2 describes the layout of the tables.

Listing 4.2: tGAP structure: physical data model

```

1 -- description of the tGAP face table
2 sql> desc tgap_face
3 name          null?    type
4 -----
5 face_id       not null number
  
```

```

6  mbr_geometry          mdsys.sdo_geometry
7  area                 number
8  imp_low              number
9  imp_high             number
10 parent_id            number
11
12
13 -- description of the tGAP edge table
14 sql> desc tgap_edge;
15 name                  null?    type
16 -----
17 edge_id               not null number
18 imp_low               not null number
19 imp_high              number
20 left_face_id         number
21 right_face_id        number
22 blg_id                number
23
24
25 -- description of the tGAP BLG table
26 sql> desc tgap_blg ;
27 name                  null?    type
28 -----
29 blg_id                not null number
30 start_node_id         number
31 end_node_id           number
32 child1_id             number
33 child2_id             number
34 tree_source           blgtree
35 top_tolerance         float(126)
36
37
38 -- description of the tGAP node table
39 sql> desc tgap_node
40 name                  null?    type
41 -----
42 node_id               not null number
43 geometry              mdsys.sdo_geometry

```

4.1.4 Binary Line Generalization (BLG) tree data type

In the tGAP structure, for storing the edge geometry and its simplification, the Binary Line Generalization tree is used. To store this tree in the database a new data type is created. First the design of the data type is described, then is described how to fill the data type.

Design of the data type

The design of the data structure is largely taken from earlier research work (see Vermeij, 2003, chapter 3), but in this case also the geometry is integrated into the BLG data type, so the original geometry of the edges is not needed any more. Listing 4.3 shows the creation of the data type within the Oracle DBMS. The edge shown in figure 2.8 (p. 22) and its BLG tree in figure 2.10 (p. 24) are used to illustrate this data type. Table 4.2 shows an example of a filled data type belonging to this BLG tree.

Listing 4.3: Creating the BLG tree object within Oracle

```

1  -- create the type for list of tolerances
2  create or replace type
3      float_list as
4      varray(524288) of float;
5  /
6
7  -- create the type for internal references
8  create or replace type
9      int_list as
10     varray(524288) of integer;
11 /
12
13 -- create the type for BLG node geometry
14 create or replace type
15     blgpoint as
16     object (
17         x number,
18         y number
19     );
20 /
21
22 -- create the type for list of points
23 create or replace type
24     point_list as
25     varray(524288) of blgpoint;
26 /
27
28 -- create the type for a BLG tree
29 create or replace type
30     blgtree as
31     object(
32         error float_list,
33         leftnode int_list,
34         rightnode int_list,
35         points point_list
36     );
37 /

```

Four lists form the base for the object that can hold the BLG tree structure inside the DBMS: A tolerance value list, a left and a right node list and a list with the intermediate vertices of the geometry of the edge. All positions in the lists are implicitly indexed, i.e. this positions are not stored explicitly and information that is on the same position in each list, belongs to the same node in the BLG tree. With the information in the left and right node lists the tree structure can be reconstructed. The entries in this list are internal pointers to which node is the right or left child of that particular node. The information on the first position of each list, belongs to the root node of the tree. If a zero is present in the left or right node list, this means that this node does not have any left or right node child. Leaf nodes have a zero in both the left and the right node list.

	1	2	3	4	5	6	
Left	2	3	4	5	0	0	
Right	11	9	7	6	0	0	
Tolerance	5.0×10^4	2.7×10^4	1.6×10^3	6.3×10^1	4.2×10^0	4.1×10^0	
Coordinate	$(x, y)_{\#11}$	$(x, y)_{\#8}$	$(x, y)_{\#5}$	$(x, y)_{\#3}$	$(x, y)_{\#2}$	$(x, y)_{\#4}$	
7	8	9	10	11	12	13	14
8	0	0	0	12	0	14	0
0	0	10	0	13	0	0	0
5.5×10^2	1.5×10^2	9.1×10^2	2.0×10^2	1.7×10^3	5.4×10^1	3.7×10^2	5.2×10^1
$(x, y)_{\#7}$	$(x, y)_{\#6}$	$(x, y)_{\#9}$	$(x, y)_{\#10}$	$(x, y)_{\#13}$	$(x, y)_{\#12}$	$(x, y)_{\#15}$	$(x, y)_{\#14}$

Table 4.2: Binary Line Generalization data type, filled with data. Information on position 1 in this list belongs to the top node of the tree. From here, the two nodes below can be derived, by following the information in the left and right lists. Information on the left node below the top, is stored on position 2 of the lists and information on the right node below the top is stored on position 11 of the lists.

Filling of the data type

For filling the BLG data structures, the Douglas Peucker algorithm is used to assign threshold values to all nodes in the tree. To save the expense of doing square root calculations, the tolerances used in the BLG tree are stored as the squared tolerance values. This means that when interacting with the BLG tree, the squared value of the tolerance has to be used. Because filling the BLG data type is only a one-time operation this probably is not the wisest choice: it would be better to use the real distance in the tree, as this prevents squaring the value of the threshold at visualization time.

A characteristic of the BLG trees is that only the intermediate vertices on a polyline are stored. The geometry of beginning and end point of an edge are not taken into account in the BLG tree data structure. To prevent storing this information in a redundant way with respect to geometry, a separate table for storing the geometry of the beginning and end nodes is necessary. Then this table contains node geometry and a unique iden-

tifier for the nodes and references pointing to the start and end nodes are stored in the BLG table.

4.1.5 Generalized Area Partition (GAP) face tree

After the tGAP structure is brought into its initial state, by copying data from the source topology tables and creation of BLG trees, the tGAP structure is ready to be filled. This filling consists of two (integrated) steps: constructing the GAP face tree (filling the tGAP face table) and GAP edge forest (filling the tGAP edge and BLG table).

To use the GAP face tree for generalization, it normally is possible to steer which faces are merged with which faces. This is accomplished by setting up a compatibility matrix and assigning weights to all thematic values (cf. Van Putten, 1997). Based on a weighting, a decision is made which two objects are merged. Because this weighting does not affect the working of the data structures (only the ordering of the faces in the tree), it was decided to leave the weighting out of the process, to focus purely on the data structures. The selection of which faces have to be merged in a step, is in this implementation based on the area of the topological faces only (join the smallest face in the data set to its largest neighbor).

Because the area of the faces is needed to make a decision to merge which two faces, it is decided to also store the area explicitly within the table: With spatial functions, it is possible to use an area calculation function to calculate the area of the faces in real time. However, for calculating the area of a topologically stored face, it is also needed to materialize the geometry of each topological face, before the area can be calculated. To prevent that for each iteration the reconstruction process for all faces has to be executed (to know which face is smallest), it is convenient to explicitly store the area in the face table. The same holds for the calculation of the bounding box.

The construction of the GAP face tree consists of a number of steps. All steps are implemented as functions within a PL/SQL package. A package groups all functions within one namespace in the Oracle database. By iterating over all the steps the GAP face tree structure is filled. Conceptually the steps to construct the GAP face tree are as follows:

1. Start with a queue of all faces in the planar partition
2. Find the least important face in the queue
3. Find the most important neighbor for this face
4. Merge these two faces (i.e. handle the edges of those two faces correctly) and remove two old faces from the queue
5. Add the merged face to the queue
6. Repeat steps 2–6, until only one face exists in the queue

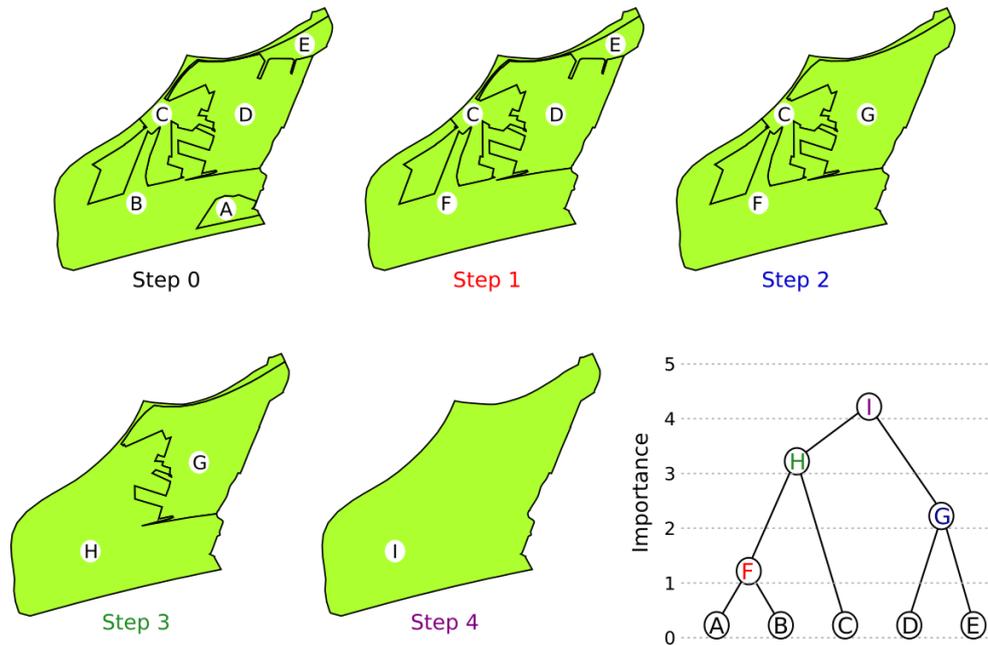


Figure 4.7: Steps for construction of the GAP face tree. Step 0 is the initial state of the partition. In step 1 face A and B are merged and succeeded by face F. Step 2 merges face D and E into face G. This process continues, until one face is left: face I.

Figure 4.7 shows an illustration of the conceptual steps and the GAP face tree that is the result of the process. Table 4.3 gives the information that is stored in the implementation in the face table. The prototype implementation of the conceptual steps of building the GAP face tree consists of the following steps to fill the tGAP face table correctly:

- All faces in the tGAP face table start with their low importance value set to zero, and the high importance value set to null. Faces with high importance set to null are still on the queue. Initialize a 'current importance' value in memory and set it to 1. Now repeat the following steps, until only one face has its high importance set to null:
 - Find the least important face, i.e. has the smallest area of the faces that are in the queue. In step 0 of the example, this is face A;
 - Find the neighbor of this face that has the biggest area. It can be found via the edges of the least important face. Face B will be selected in step 0 of the example;
 - Merge the two faces. This means create correct edge entries for the new face in the tGAP edge and tGAP BLG table (see next two sections). The two old faces can be removed from the face queue by setting their high importance value to the current importance value in the tGAP face table. Also the parent

id of the two faces is set with the face id of the new face id. In the example face A and B both get their high importance set to 1 and parent id to F (see table 4.3);

- For the new face: calculate the area, by summing the values of the two old faces, and the bounding box, by aggregating the bounding box geometry of the two old faces into a new bounding box;
 - Add the merged face to the face table, with its low importance set to the current importance value and its high importance set to null (to get it on the queue). In the example, face F is added to the tGAP table.
 - Raise the 'current importance' value with 1;
- For the last face, set the high importance to the current importance value.

face id	imp low	imp high	area	mbr	parent id
A	0	1	...	$(x_{ll}, y_{ll}, x_{ur}, y_{ur})$	F
B	0	1	...	$(x_{ll}, y_{ll}, x_{ur}, y_{ur})$	F
C	0	3	...	$(x_{ll}, y_{ll}, x_{ur}, y_{ur})$	H
D	0	2	...	$(x_{ll}, y_{ll}, x_{ur}, y_{ur})$	G
E	0	2	...	$(x_{ll}, y_{ll}, x_{ur}, y_{ur})$	G
F	1	3	$area_{A+B}$	$mbr_{A \cup B}$	H
G	2	4	$area_{D+E}$	$mbr_{D \cup E}$	I
H	3	4	$area_{F+C}$	$mbr_{F \cup C}$	I
I	4	5	$area_{H+G}$	$mbr_{H \cup G}$	-

Table 4.3: tGAP face table after constructing the GAP face tree. Colors of the text correspond to the step in which the information is added (steps are shown in figure 4.7)

This way the GAP face tree is constructed. Now, from table 4.3 the tree shown in figure 4.7 can be visualized by connecting the face id with the parent id references stored in the table, starting at the face that does not have a parent id filled in, i.e. face I.

4.1.6 GAP edge forest

Because left right topology is used in the implementation, the faces consist of a collection of edges having references to the faces. This information is stored in the tGAP edge table and is created, while merging the faces.

Filling the tGAP edge table

With the merging of the faces to construct the GAP face tree, three conceptual things may happen to the edges:

- If an edge is the common boundary between the two faces that are merged, it is removed. For example, this happens to edge 1 in figure 4.8 as it is the common boundary edge between face A and B;
- An edge still can be used after a face merge step, but the face references need to be adapted to the new situation, e.g. edge 3 in figure 4.8 has face F on its left side in step 1, instead of face B in step 0;
- Two or more edges need to be joined and a new edge is needed. For example in figure 4.8, this is the case for edge 2, 4 and 5 from step 0 that are merged into edge 14 in step 1.

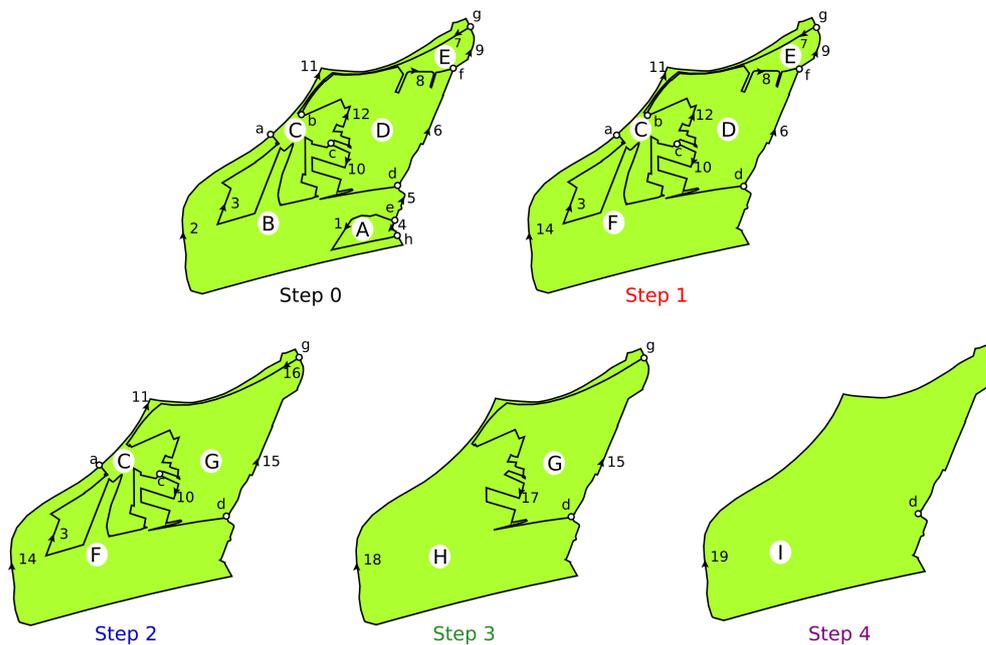


Figure 4.8: Edges in the tGAP structure: the GAP edge forest

To accommodate these possibilities, the conceptual steps for creating the GAP edge forest are as follows:

1. When the face merging process starts, a global edge queue with all edges is created;
2. With each merge of two faces, the edges that belong to the two faces are removed from the global edge queue and added to a local edge queue;
3. For all edges in the local edge queue is checked, whether:

- (a) It is a common boundary edge, then it is removed from the local queue without further action;
- (b) It is an edge that still can be used. Then, the edge its face references are adapted to the new situation and it is placed back in the global queue;
- (c) It is an edge that needs to be joined. The edges that it needs to be joined with are deleted from the local queue and inserted into a list, with which the joins of the BLG trees can be stored, therefore see the next section. After joining is handled, the joined version of the edge is added to the global edge queue.

edge id	start node	end node	left face	right face	imp low	imp high	blg id
1	e	h	A	B	0	1	1
2	h	a	-	B	0	1	2
3	c	a	B	C	0	1	3
4	h	e	A	-	0	1	4
5	e	d	B	-	0	1	5
6	d	f	D	-	0	2	6
7	g	b	E	C	0	2	7
8	b	f	E	D	0	2	8
9	f	g	E	-	0	2	9
10	c	d	D	B	0	1	10
11	a	g	-	C	0	3	11
12	c	b	C	D	0	2	12
14	d	a	-	F	1	3	14 = ((2U4)U5)
3	c	a	F	C	1	3	3
10	c	d	D	F	1	2	10
15	d	g	G	-	2	4	15 = (6U9)
16	g	c	G	C	2	3	16 = (7U12)
10	c	d	G	F	2	3	10
18	d	g	-	H	3	4	18 = (14U11)
17	g	d	G	H	3	4	17 = (16U10)
19	d	d	-	I	4	5	19 = (18U15)

Table 4.4: tGAP edge table after constructing the GAP edge forest. Colors of the text correspond to the step in which the information is added (steps are shown in figure 4.8)

Table 4.4 shows the tGAP edge table after constructing the GAP edge forest, for the area partition shown in figure 4.7. In the prototype implementation the construction of the GAP edge forest starts with setting the low importance value to zero and the high importance to null for all edges in the tGAP edge table. Edges with high importance set to null are on the global edge queue. With each face merge step the following things are done:

- For all edges belonging to the two faces that are merged, all required information is retrieved from the tGAP edge table, with high importance set to null (the global edge queue) into memory (the local edge queue). The high importance of these edges in the tGAP edge table is set to the 'current importance' value (i.e. these edges are removed from the global edge queue).
- For all edges retrieved into memory, it is checked, whether:
 1. It is a common boundary edge, then it is just removed from memory;
 2. It is an edge that still can be used. Then, the edge its face references are adapted to the new situation and a new entry is created in the tGAP edge table. This edge gets the same 'edge id' as it had, but the low importance value is now set to the 'current importance' value. The high importance value is set to null, this way the edge is on the global queue again;
 3. The edge is an edge that needs to be joined. The edges that it needs to be joined with are deleted from memory and add to a list with which joining is handled (see next section). When the joined version of the edge is created in the tGAP edge table, a new 'edge id' is generated. The low importance value of this edge is set to the 'current importance' value and the high importance value is set to null.

After handling all edges that are retrieved to memory, the local edge queue will be empty for the next face merge step and the face merging process can continue. Because the edges can be valid on different importance levels, the edges are uniquely identified by a combination of the 'edge id' and 'imp low' field. During construction of the GAP edge forest, edges can be selected from the tGAP edge table based on a combination of edge id and imp high set to null, because the edge id will be unique within this queue.

Storing information on joins of BLGs in the TGAP BLG table

To make simplification of the edge geometry possible, the Binary Line Generalization tree is used, instead of the geometry of the edges. These trees are created and stored before constructing the tGAP structure. However, during construction of the tGAP structure, it is possible that edges need to be joined (see previous section). The BLG tree has the advantage, that it is not needed to duplicate the geometry in the case edges are merged, but that it is possible to join the BLG trees of the edges, by storing information on how to join the BLG trees. Then also information for simplification of the joined edges is available.

The merging of BLG trees will happen pair wise, because in this way the node in which the join takes place can hold a threshold value, just as is the case for nodes stored in the BLG trees themselves (see for an explanation of how this works Van Oosterom, 1990, chapter 5). Conceptually, this is what happens when storing the (joined) BLG trees in the tGAP structure:

- A global queue of BLGs is created;

- A list of edges, which are represented by BLG trees and need to be joined, is given;
- From this list BLG trees that are next to each other are joined pair wise;
- For each join pair, an accuracy measure is computed and the join is stored in the global queue of BLGs, as this join may be used for pairwise joining of BLG trees again.

blg id	tree source	start node	end node	child 1	child 2	top tolerance
1	blgtree	e	h			-1
2	blgtree	h	a			-1
3	blgtree	c	a			-1
4	blgtree	h	e			-1
5	blgtree	e	d			-1
6	blgtree	d	f			-1
7	blgtree	g	b			-1
8	blgtree	b	f			-1
9	blgtree	f	g			-1
10	blgtree	c	d			-1
11	blgtree	a	g			-1
12	blgtree	c	b			-1
13		e	a	-4	2	...
14		d	a	-5	13	...
15		d	g	6	9	...
16		g	c	7	-12	...
17		g	d	16	10	...
18		d	g	14	11	...
19		d	d	18	-15	...

Table 4.5: tGAP BLG table after constructing the GAP edge forest. Colors of the text correspond to the step in which the information is added (steps are shown in figure 4.8)

In the implementation this is done as follows:

- The tGAP BLG table holds the BLG trees, all identified by their blg id field. This table will also hold the information for joins of BLGs;
- While filling the tGAP edge table, lists of edges *read*: (joined) BLGs, are generated from the local edge queue hold in memory, which need to be merged into one edge, *read*: one BLG. In the example, in case of step 1 of the face merge process, there will be one list of BLGs to be joined consisting of BLG 2, 4 and 5;
- Each list of BLGs to be merged is then treated as follows, until no joins are needed any more because one BLG is remaining:

1. Take two BLGs from the list that have one node in common. Which BLGs have one node in common can be figured out from the tGAP BLG table, because each BLG has a start node and an end node associated, e.g. BLG 2 and 4 have node h in common;
2. Join the two BLGs into a new BLG in this node, by calculating a tolerance value for this join and generating a new blg id for this joined BLG. In the example, BLG 2 and 4 will become BLG 13. Store this BLG in the tGAP BLG table. If the BLG has an incorrect direction compared to the new start and end node, a minus sign is stored with the child reference. E.g. -4 in case of BLG 13, because it has to be flipped before it has the correct direction, that is from node e going in the direction of node a;
3. Remove the two old BLGs from the list in memory and place the newly created BLG on the list;
4. Continue at step 1, until there is only one BLG in the list. In the example this would mean one more iteration, in which BLG 13 and BLG 5 are merged to a new BLG, number 14.

4.1.7 Architecture to visualize the content of the tGAP structure

After implementing the tGAP structure and loading data into it, it is ready for being used at visualization time. In 2005, Google introduced its Google Earth product. It shows a 3D view of the globe on which raster and vector data can be projected. To retrieve the information broadband streaming technology is used.

Google Earth has the capacity to load external geographic data onto its 3D globe, via a so-called network link. This network link connects to a web server and retrieves geographic information (raster or vector) described in an open format, developed by Google. This format is called KML (short for Keyhole Markup Language), specification is available from (Google, 2005). It is an XML based format and has some characteristics taken from the Geographic Markup Language (GML) as created by OGC, available from (OGC, 2003). XML stands for eXtensible Markup Language and is a standard format developed by the World Wide Web Consortium (W3C, 2004). Originally this standard was meant for electronic publishing, but it plays more and more an important role in the electronic data interchange field.

To connect Google Earth with the data stored in the tGAP structure in the DBMS it is needed to connect a web server to the database. This can be done through embedding a programming language in a web server, and, based on the functionality the programming language offers, connect to the database, retrieve the geographic data from the database and transform it to the format desired by Google Earth, i.e. KML. This architecture is depicted in figure 4.9.

To make this possible, the choice was made to use the Apache web server, the Python programming language and the Geospatial Data Abstraction Library (GDAL), that offers bindings for Python. Apache, the web server takes care of processing and answering the requests from Google Earth (made over the HyperText Transfer Protocol, HTTP).

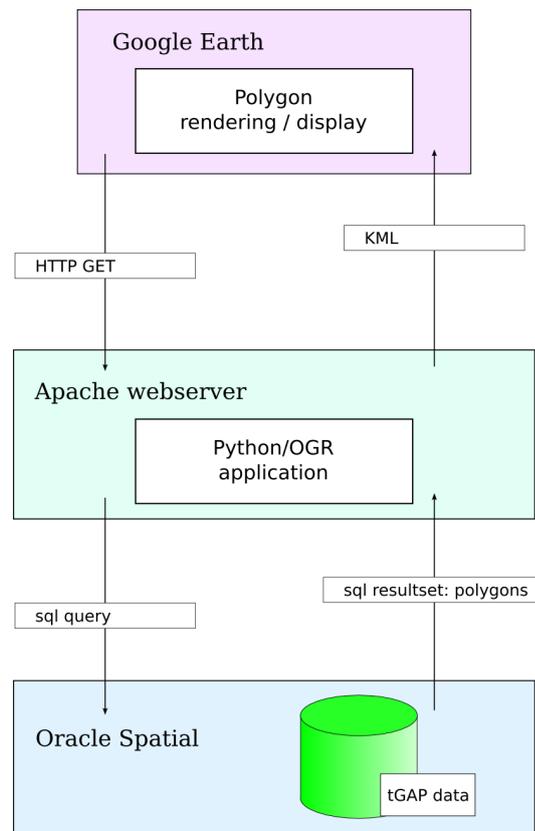


Figure 4.9: Visualizing the contents of the tGAP structure in Google Earth. Source: Marian de Vries

The Python programming language is embedded into Apache, via ModPython (Apache Software Foundation, 2006). This makes it possible to create application logic in the web server. GDAL is a translator library for diverse geospatial raster data formats. It also contains a Simple Feature library called OGR (see Warmerdam, 2006). The library is programmed in the C++ programming language and offers also bindings for Python. With the OGR application programming interface (API) it is possible to retrieve geographic information from a database management system (Oracle) and export it to the Geography Markup Language. Because GML and KML are both XML based formats, it is possible to translate one format into the other with standard XML tools. The eXtensible Stylesheet Language Transformations, XSLT, (see W3C, 2001), can be used to transform the intermediate format (GML) into KML. Appendix A gives an example of the request-response sequence that is made in the created architecture to retrieve geographic data from the tGAP structure.

Coordinate transformations

Because Google Earth can show the complete earth in 3D, it uses a coordinate system, that can describe coordinates all around the globe. The World Geodetic System 1984 (short, WGS84) is used for storing coordinates. Because the geographic data in the database are stored in the Dutch reference system, it is needed to transform these coordinates to the WGS84 system. This functionality of transforming is also offered by the OGR library, but can also be done at the database side.

To perform coordinate transformations at the database side, it is needed that the database knows about the source and destination coordinate system. The data of the Dutch Cadastre did not have any geographic reference system attached to the coordinates, as described in section 4.1.2. To apply a coordinate transformation it is needed to associate a geographic reference system. This association can be performed by an update statement, which loops over all geometries in the table and sets the spatial reference system number: SRID, spatial reference identifier in Oracle terms (an example is shown in listing 4.4). For the Dutch reference system the SRID is 90112.

Listing 4.4: Updating spatial reference identifier for geometry in a table

```
1 -- replace <table> and <geom_column> with the correct values
2 update <table> a set a.<geom_column>.sdo_srid = 90112;
```

Also, with GDAL it is possible to perform the required transformation. A problem that appears when visualizing transformed geographic data from the Dutch reference system to the WGS84 system onto the 3D globe in Google Earth, is that the data appears to be intersecting with the surface of the globe, giving strange rendering effects. By raising the z-coordinates of all geographic features, this intersection can be prevented.

Another problem, that is encountered with both coordinate transformations (using either the database its transformations or GDAL its transformations), is that the geographic data is not correctly positioned on top of the globe, but shifted to the South (around 200 meters). A custom transformation can solve this problem.

With the required changes to the transformations (i.e. raising z-values and shifting the data to the North), a choice was made to embed the transformations in the middle ware, because the transformation parameters are easy to change with the API offered by GDAL. No further investigations have been performed what is causing these 'coordinate errors'.

4.1.8 Lessons learnt from implementing

This section describes some lessons learnt from the prototype implementation:

Materialize geometry needs edge detail The function to materialize the geometry of the faces, the 'return polygon function' as described in (Van Oosterom, 2005), takes only one parameter as argument, i.e. the face identification. This parameter does not suffice when the BLG trees are used for the edge geometry: Faces to be shown on a certain importance level form a complete area partition. The edges of the faces

are stored as BLG trees. The BLG trees can offer edges with more or less detail, based on a tolerance value. To specify the amount of detail, an extra parameter for the function. If the return polygon function only has one parameter (i.e. face identification), there is no way of knowing how much detail is needed for the edges *and* to be sure that this amount of detail is the same for all edges belonging to different faces. The amount of detail of the edges needs to be equal for all faces, otherwise the constraints of an area partition (partition completely filled, no gaps, no overlaps), might be violated, i.e. gaps and overlaps can come into existence.

Smartly descend the trees To materialize the geometry of the edges, it is needed to (a) descend the trees, to get the wanted level of detail, and (b) transform the part of the tree, that is needed for the wanted level of detail, to geometry.

The implementation does this, by first joining all BLG trees from the tGAP blg table into one (large) BLG tree, and then descending to the wanted level. This is not very smart: When materializing geometry of a join of BLGs, the junction node of the joins already contains information on the wanted level of detail. It is thus more advanced to take the wanted level of detail already in the joining step into account, preventing unnecessary joins.

Thus, for descension, joining and materializing geometry out of the stored information on the BLGs, one integrated algorithm is needed, that descends the tree in a smart way based on the required tolerance, as well as materializes geometry.

Direction of BLGs is important Although the edges are represented by BLGs, the edge direction is still important to be able to form valid geometry for a topologically stored face (outer rings oriented opposed to inner rings). Because edges point to BLGs, it is needed to have the BLGs oriented correctly, to create correct edge geometry from this stored BLG information. The used approach in the implementation is that the child 1 and child 2 pointers are prefixed with a minus sign, if the BLG has to be turned, for being in the right direction. However, this means that one node reference of each join is redundant, as this points to the same 'junction' node.

All in all, for storing joins of BLGs, it is needed to pay attention to the direction of the BLG trees, although this was not mentioned explicitly in (Van Oosterom, 2005).

4.2 Testing

In this section testing of the data structures is described. First, in section 4.2.1 geometric redundancy is treated. Section 4.2.2 shows the requirements for the data structures, logical (number of rows in the tables), as well as physical (size on disk), in comparison with the source topology. This last comparison is not easy to make, because from the logical sizes (and table contents) can be derived, that it is better to change the logical model of the tGAP structure a bit first.

The second requirements that are measured, is the time needed for filling the tGAP structure with data (section 4.2.3–4.2.4) and the time needed to visualize data stored in the tGAP structure (section 4.2.5). Some problems are encountered with building a 3D functional index (4.2.6) and the mapping of the scale to importance level (4.2.7). Section 4.2.8 summarizes lessons learnt from testing the tGAP structure.

4.2.1 Removing redundant geometrical storage

The implementation of the data structures shows that it is feasible to remove all geometric redundancy. This is made possible by the usage of the BLG tree data structure, that only stores the intermediate points of the edges, while the nodes store the geometry of the junctions of the edges only once. Therefore, all geometry is stored only once, while for the rest references are stored. However, these references will, and do, take up storage space, but have the advantage that the source geometry is only stored in one place.

That it is possible to remove all geometric redundancy from the data structures is also due to the choice which generalization operator to use in this research: face merging and reclassification and (line) simplification. For certain generalization operators (the ones that work on the geometry of the objects, like moving objects, see table 2.1) it is difficult to prevent redundancy of geometry, because two versions of geometry will have to be stored with such an operator. A solution to this problem might be that the geometrical transformation (thus *not* the resulting geometry) should be stored in the GAP face tree. However, this might be too calculation intensive for processing, and storing redundant geometry then might be the only solution for this problem.

4.2.2 Size of tGAP structure, in comparison with the source topology

The size of the tGAP structure can be measured. But to set the measurements in context, it is needed to compare these measures, with other approaches of storing geographic data. The tGAP structure is special in this way that the structure makes variable access to geographic data possible, which all other approaches available do not do (or only with a fixed number of detail levels, cf. MRDB approach). Therefore, comparison is a not really straightforward.

To put the comparison in context, from all source topology face tables, the geometrical variant is derived. This is done by selecting the face id and its geometry into a new table, in which the face geometry is stored as a geometrical polygon. The size of these tables is given in table 4.6.

dataset	size (MB)
Cadastrre (small)	.17
Cadastrre (large)	24.73
Amsterdam	82.03

Table 4.6: Size of geometrical version of the source topology tables

Then, the sizes of the source topology and tGAP tables is measured. To make the comparison, all tables of the three different data sets do reside in the same logical model as shown in listings 4.1 and 4.2, containing the same number of references. The result of this comparison is shown in table 4.7.

dataset	table	source topology		tGAP structure		
		size (MB)	# of rows	size (MB)	# of rows	# of distinct 'id' rows
Cadastre (small)	face	.01	161	.05	321	321
	edge	.12	499	.17	3938	770
	blg			.16	791	791
	node	.02	341	.02	341	341
Cadastre (large)	face	.66	50238	16.84	100475	100475
	edge	38.55	178815	148.87	3258262	263223
	blg			51.48	272046	272046
	node	4.87	129441	4.87	129441	129441
Amsterdam	face	2.02	170368	56.27	340735	340735
	edge	94.16	418530	291.38	7113680	614707
	blg			132.84	658219	658219
	node	10.70	281216	10.70	281216	281216

Table 4.7: Number of rows in, and physical size of, source topology and tGAP tables

What immediately strikes, is that the physical size of source topology tables in two of the three cases already exceeds the geometrical approach (small cadastral set: geometry .17 MB versus topology .15 MB, large cadastral set: geometry 24.73 MB versus topology 44.08 MB and Amsterdam data set: geometry 82.03 MB versus topology 106.88 MB). Because the source topology model used in this thesis research is quite a minimalistic approach for storing topology, but in most cases requires more storage space than the geometrical approach, it is needed to keep the tGAP structure as lean as possible to keep size requirements low. More topological references and attributes are added to this structure in comparison with the source topology model. So, it is clear that the tGAP structure will also require more space than the source topological data (and thus also more space than a geometrical approach).

With the comparison of the size of the source topology data to the size of the tGAP data, first, the focus is on the number of rows within the tables. Compared to the original node table, the amount of information on nodes in the tGAP structure stays the same. This is indeed shown by the amount of rows, that is the same in both cases. The tGAP face table shows the following relation to the source topology face table: two times the number of source faces minus one. This is due to the binary structuring of the GAP face tree and is also normal. What is striking, is that the amount of edge entries has grown explosively in the tGAP edge table. This is due to the fact that new versions of edges are created and inserted to the tGAP edge table with each face merging step. To find out, how many edges are duplicated to new importance levels, the last column in

table 4.7 shows the distinct amount of edge ids in the tGAP edge table. Based on this number, one can conclude that in the tGAP edge table around 30 percent new, joined edges are stored, while the rest of the edges has different versions (with different left face and right face pointers). This can also be observed in the example edge shown in table 4.8, which shows (if not taking the low importance and high importance values into account as this are 'version numbers' of the edge) that for this edge only the left face information is changing over the rows. Therefore, during personal communication with Van Oosterom, a suggestion was made to store the edge information that changes into lists (varray's in Oracle terminology), instead of storing duplicate, but nearly identical, versions of information as new rows each time. This way storing new rows is avoided, which saves storage space.

edge id	left face	right face	imp low	imp high	blg id
141532990	140969346	140969343	0	7	141532990
141532990	140969346	140969508	7	9	141532990
141532990	140969346	140969510	9	12	141532990
141532990	140969346	140969513	12	17	141532990
141532990	140969346	140969518	17	18	141532990
141532990	140969346	140969519	18	27	141532990
141532990	140969346	140969528	27	29	141532990
141532990	140969346	140969530	29	36	141532990
141532990	140969346	140969537	36	37	141532990
141532990	140969346	140969538	37	40	141532990
141532990	140969346	140969541	40	42	141532990
141532990	140969346	140969543	42	43	141532990
141532990	140969346	140969544	43	48	141532990
141532990	140969346	140969549	48	51	141532990
141532990	140969346	140969552	51	56	141532990
141532990	140969346	140969557	56	57	141532990
141532990	140969346	140969558	57	62	141532990
141532990	140969346	140969563	62	63	141532990
141532990	140969346	140969564	63	66	141532990
141532990	140969346	140969567	66	67	141532990

Table 4.8: Example edge, showing that some fields stay the same in different versions of the same edge

Now, the physical size of the tGAP structure will be compared. At this moment, this is not really a realistic comparison, as the logical model can be changed (as described above), so that the tGAP structure will get leaner and less rows have to be stored. In the current implementation, the physical size is around 8 times larger, than the minimalistic source topology approach. This comparison is not completely fair, as the source topology model, does not have any area or geometry physically stored, while this was stored

in the tGAP structure.

When taking indices into account, it is probable that the tGAP structure will need even more space, because the structure uses more references than the source topological data models. The sizes of the present indices in the implementation are shown as indication in appendix B, but will not be taken into account here, because indexing is used for fast retrieval of information and which information needs fast retrieval is dependant on the application. E.g. fast retrieval is different, if one wants to return complete polygons (i.e. do materialization of geometry of the faces) from the data structure and do this reconstruction at the database side, or if one wants to send parts of the data structure to the client side, which then can perform materialization of geometry, as for example will be the case with progressive transfer.

4.2.3 Loading speed

The time needed to perform the filling step for the tGAP structure could be accurately measured, as the building process is one function (calling all other functions from the PL/SQL package). Time to execute this function can be returned by the database. The timing results of the building process for the three different data sets are shown in table 4.10. The graph shows that the process is linearly expanding for the datasets used. However, the figure shows the results that are obtained after tuning the build process, because an early estimate showed that the process would be taking too much time.

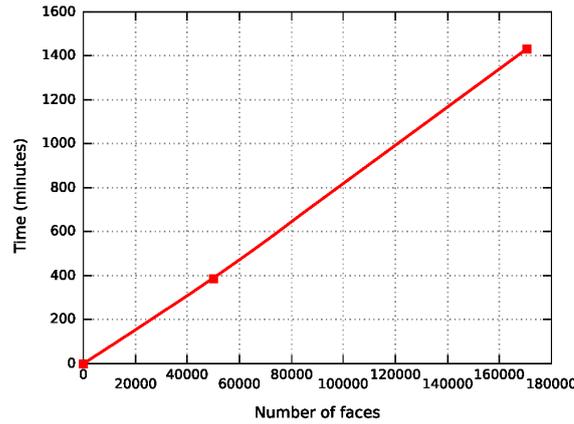
4.2.4 Tuning loading speed

Before running the construction process of the tGAP data structure for the fairly large data set of Amsterdam (around 170,000 face records), an estimate was made how long the process would take. The process was started and the amount of iterations of building (merging) was limited to 50 times. The time needed for these 50 iterations was linearly extrapolated to the total amount of iterations needed. This extrapolation showed time needed of more than two weeks. Therefore, it was needed that an optimization took place. To see which parts of the process took long, time recording statements were embedded within the PL/SQL code, to measure which function calls took which amount of time. Based on the measurements an average was calculated, how much time was spent in each function. The result of this code profiling benchmark is shown in figure 4.11.

The functions adding the largest amount of time to the total process are then further analyzed. It appears that the most time in the first place was spent in the functions that finalize the faces (update their high importance value) and in the process of joining the BLGs. Analyzing the code in these two functions, showed that the face table and the node table are heavily used. Seeing which indexes were present or absent revealed why the time needed to perform these two functions was quite high: The information in the tGAP face table was updated, on which a unique index for all fields (except the bounding box) was present. For visualization, this 'over-indexing' was used for selecting the faces to be shown on the screen with a certain importance level. However, because the index is unique, for each new piece of information inserted into the table, or updated,

Amount of faces	Time (minutes)
161	0.05
50238	389
170368	1431

(a)



(b)

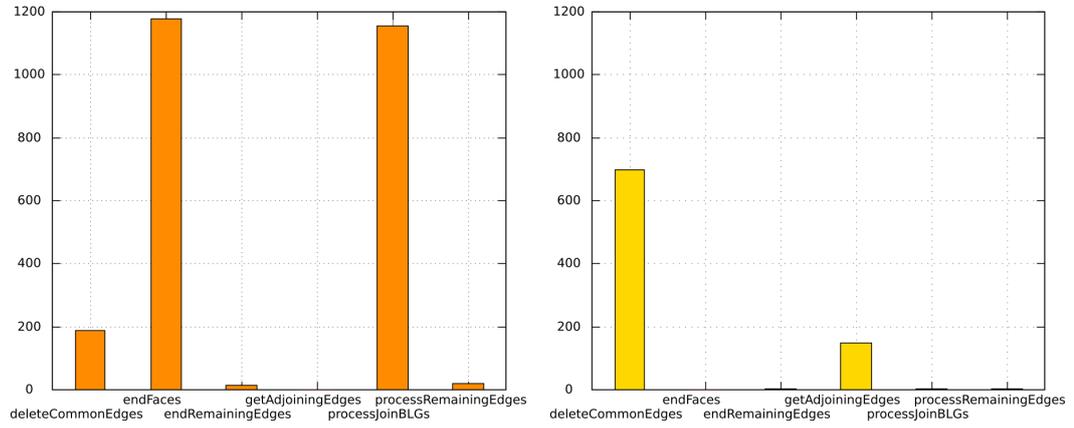
Figure 4.10: Time needed to build the GAP face tree

this constraint needs to be enforced and thus entails checking if the new or updated information is indeed unique. Because the unique index on the face table is not needed, the index was removed.

For the join process of the BLG table, the node incidences of the edges are checked. Here the node table is used and after a check, it appeared that accidentally the index on the primary key field (node identifier) was left out. This index is added to the node table. After these changes are made to the build process, another iteration for checking is performed.

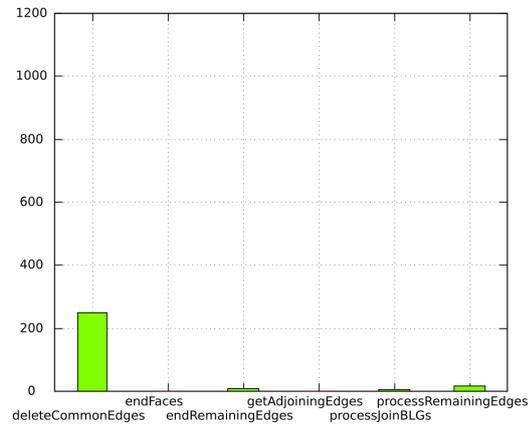
The second graph of figure 4.11 shows that the ending of the faces and joining of the BLGs are not the bottlenecks any more after deleting or adding the relevant indexes. The same problem, as seen with the face table, is also present with the edge table (over-indexed). This is shown with the delete common edges function. This function uses the edge table. On the edge table a unique index was present to make selection of the edges faster. After this unique index is removed, the performance of the process improves.

In the last iteration, this delete common edges function still appears to be one of the bottlenecks, but the total time needed to perform the operation, was brought within



(a) First try

(b) Second try, after removing indices



(c) Final result

Figure 4.11: Tuning building speed of the GAP face tree

time available: The total time needed then predicted, based on the 50 iterations, was lowered to one day. As this was acceptable, no further tuning was performed.

However, the delete common edges function, still requires a substantial amount of time. The delete common edges function compares the amount of edges going in and out for each node that is related to one of the two faces in a face merging step. This way, the incidence rate the edges with the nodes have is figured out. If, after a face merge step, this incidence rate of an edge to a node is changed to two, this means that the edges in that node should be joined. This is found out via counting the nodes and edges in the table. It would be faster if this operation could be performed in memory: then, the process of building can even possibly be tuned more.

It might also be possible to make the whole process faster, by implementing it in a different programming language that allows storing the complete initial face, edge and BLG queues in memory and then select the information during construction of the tGAP structure from memory, instead of that selection takes place from the tables. Now the process requires a lot of concurrent, i.e. both read and write, disk activity from the DBMS, which can slow down the process.

4.2.5 Visualization speed

For visualization Google Earth is used. With this program it is possible to interactively browse through the geographic data sets, with zooming in and out and panning over the complete area. Google Earth times out, when after 15 seconds the request is made, no answer is received. Therefore, selection of which faces have to be shown, reconstruction of the faces and coordinate transformations have to be possible within a time range of 15 seconds. Using a spatial index on the bounding boxes and an index on the importance low and high values of the faces or using a 3D index (see next section), it was possible to interactively browse through the complete data set without problems.

For visualizing the whole area of 50 thousand faces from the original data source, around 6 minutes is needed as a timing experiment showed and also can be extrapolated from figure 4.2.

No further analysis of visualization speed was carried out, as from these two results can be concluded that the data structures make it possible to use the generalized data in real time (i.e. within a time span of 15 seconds from the start of a request), while this would be problematic if only the source data is available.

However, it would be interesting to compare the implementation with a setup were multiple layers of geographic data are available, to see how this would behave with visualization. This is also interesting for usability research (e.g. is it better to show gradual changes in geographic information when interacting with a geographic data source, or do predefined levels of detail comfort the user more).

4.2.6 Building a 3D functional index

To speed up visualization another test was performed. In (Van Oosterom, 2005) a possibility was mentioned, to create a 3D, functional index on the face table, to make fast

selection of the faces possible. The proposed function described in the paper has more than one argument (i.e. geometry, importance low and importance high). When trying to create a 3D spatial index on a function with more than one parameter, the index creation failed with an error message.

After reducing the function parameters to only one parameter (face identification), the creation of the 3D functional index succeeded. The listing that works, is shown in listing 4.5.

Listing 4.5: Creating the 3D functional index

```

1 create or replace function compute_3d_block(id in number) return mdsys.
  sdo_geometry deterministic is
2   imp_low number;
3   imp_high number;
4   mbr mdsys.sdo_geometry;
5   geometry mdsys.sdo_geometry := mdsys.sdo_geometry( 3002,
6               90112,
7               null,
8               sdo_elem_info_array(1, 2, 1),
9               sdo_ordinate_array() );
10  ordinates mdsys.sdo_ordinate_array := mdsys.sdo_ordinate_array();
11 begin
12  select mbr_geometry, imp_low, (imp_high - 1) into mbr, imp_low, imp_high from
13     tgap_face where face_id = id;
14
15  ordinates.extend(6);
16  ordinates(1) := mbr.sdo_ordinates(1);
17  ordinates(2) := mbr.sdo_ordinates(2);
18  ordinates(3) := imp_low;
19  ordinates(4) := mbr.sdo_ordinates(3);
20  ordinates(5) := mbr.sdo_ordinates(4);
21  ordinates(6) := imp_high;
22  geometry.sdo_ordinates := ordinates;
23
24  return geometry;
25 end;
26 /
27
28 insert into
29   user_sdo_geom_metadata
30   values
31   ('tgap_face',
32    '<schema>.compute_3d_block(face_id)',
33    mdsys.sdo_dim_array(
34     mdsys.sdo_dim_element('x',125000,160000,1.0e-1),
35     mdsys.sdo_dim_element('y',415000,440000,1.0e-1),
36     mdsys.sdo_dim_element('z', 0, 51000,1.0e-1)),
37    90112)
38 /

```

```

38
39 create index
40   x_tgap_face_3dbox
41 on
42   tgap_face(compute_3d_block(face_id))
43 indextype is
44   mdsys.spatial_index
45 parameters
46   ('sdo_indx_dims=3 sdo_rtr_pctfree=0 tablespace=indx')
47 /

```

Then, to see what the differences are while using the indices, two queries are crafted. One query is assumed to use the 2D spatial index on the bounding boxes of the geometry and a separate index for selecting the faces available for an importance level. This query is shown in listing 4.6. The two criteria in the where-clause of the query will select faces from the tGAP face table that are valid for a certain spatial extent *and* an importance level.

Listing 4.6: Query which will use a 2D and a 1D index

```

1 select
2   f.face_id
3 from
4   tgap_face f
5 where
6   sdo_filter(
7     mbr_geometry,
8     mdsys.sdo_geometry(
9       2003,
10      90112,
11      null,
12      mdsys.sdo_elem_info_array(1, 1003, 3),
13      mdsys.sdo_ordinate_array(132209.2,413430.8,
14                               147705.7,426725.5)
15    ), 'querytype=window') = 'true'
16 and
17   imp_low <= 50227.0
18 and
19   imp_high > 50227.0;

```

Listing 4.7 shows the other query that is assumed to use the 3D functional index. It will select the same faces as the query in listing 4.6 (i.e. same extent and importance level is used in both queries).

Listing 4.7: Query which will use the 3D functional index

```

1 select
2   f.face_id
3 from
4   tgap_face f

```

```

5 where
6   sdo_filter(
7     compute_3d_block(face_id),
8     mdsys.sdo_geometry(
9       3003,
10      90112,
11      null,
12      mdsys.sdo_elem_info_array(1, 1003, 3),
13      mdsys.sdo_ordinate_array(132209.2,413430.8,50227,
14                               147705.7,426725.5,50227)
15    ), 'querytype=window') = 'true';

```

To see, if the DBMS is really using the indices as planned, it is instructed to show the way it handles the queries by showing its query plans. Listing 4.8 shows the plan of the query that will use the two separate indices.

Listing 4.8: Query plan of query using the 2D and a 1D index

id	operation	name
0	select statement	
* 1	table access by index rowid	tgap_face
2	bitmap conversion to rowids	
3	bitmap and	
4	bitmap conversion from rowids	
5	sort order by	
* 6	domain index	x_tgap_face_mbr
7	bitmap conversion from rowids	
8	sort order by	
* 9	index range scan	x_tgap_face_ih

predicate information (identified by operation id):

```

16 1 - filter("imp_low"<=50227.0)
17 6 - access("mdsys"."sdo_filter"("mbr_geometry","mdsys"."
18   sdo_geometry"(2003,90112,null,"mdsys"
19   ."sdo_elem_info_array"(1,1003,3),"mdsys"."
20   sdo_ordinate_array"
21   (132209.2,413430.8,147705.7,426725.
22   5)), 'querytype=window')='true' and "imp_high">50227.0)
23 9 - access("imp_high">50227.0)
24 filter("imp_high">50227.0)

```

Listing 4.9 shows the plan of the query that uses the 3D functional index.

Listing 4.9: Query plan of query using the 3D functional index

id	operation	name
0	select statement	
1	table access by index rowid	tgap_face

```

5 * 2 | domain index | x_tgap_face_3dbox
6
7 predicate information (identified by operation id):
8
9 2 - access("mdsys"."sdo_filter"("<schema>". "compute_3d_block"("
    face_id"), "mdsys"."sdo_geo
10     metry"(3003,90112,null,"mdsys"."sdo_elem_info_array"
        (1,1003,3), "mdsys"."sdo_ordinate_array"
11     (132209.2,413430.8,50227,147705.7,426725.5,50227)), '
        querytype=window')='true')

```

A comparison of the two query plans shows that the 3D index is useful with respect to fast selection (listings 4.8 and 4.9): The query plans show that the indices indeed are used as intended. In case of the 3D index, only one index has to be used instead of two separate ones, which means less overhead as the shorter query plan in listing 4.9 shows. Furthermore, during testing of the queries, the following things are observed: A cold query (fired for the first time to the DBMS after it was just started), shows a difference of time required of 6 seconds (2D + 1D index) versus 3 seconds (3D index). Hot queries (repeating the queries with data already loaded into the cache of the DBMS), show more extreme results in favor of the 3D index: average of 1.5 second query time for the 2D + 1D index approach versus 0.05 seconds for the 3D index.

4.2.7 Problems encountered with mapping scale to importance

When using the data structures for visualization, the choice made for importance levels, seems to be biased in the implementation made in this research. The importance levels of the faces are related to the choices the generalization algorithm has made, i.e. the collapse function and importance function are intermingled due to the choice of the low importance and high importance values for the faces: All faces start at the same importance level, that is zero. Each time two faces are merged, the importance level is raised with one.

When querying the structure, it is needed to calculate which importance level is best for the size of the area to be shown. A mapping has to be found between scale and importance level. Because with each step in the generalization process the importance value is raised, this creates a spatial dependence of the importance level, because the decisions which faces are merged are based on the size of the area of the faces. In general, it is likely that smaller faces are found more often in urban areas, and larger faces are found in rural areas. It thus takes longer, from the start of the build process, before merging of faces takes place in the rural area, then in the urban area.

In this case, the mapping was found through data analysis. An analysis that is performed shows the average size of the faces per importance level (for the complete area). Based on this relationship, the mapping function to map the size of the area to the screen towards importance is derived. It appears that this relationship is non-linear.

The problem that occurs with this mapping, can be described as follows: the mapping that is performed is globally described. However, locally, especially in urban area,

a lot of changes can occur in the importance ranges of the faces (due to a lot of merges, with small faces). The global mapping can not satisfy this quick changes in importance ranges locally, and with zooming in and out, not all importance levels can be adequately queried. A better selection criterion to use for the importance levels possibly could be to take the value of the area of the smallest face in a merge operation into account for the importance values of the faces. This might be a good choice, because of the following: When visualizing, there should be a way, to tell that certain objects are not to be shown any more on that scale, because they have become too small (cf. amount of edge detail with the BLG, because points are coming too close and will form e.g. 1 pixel on the screen). Also objects having a too large area for that scale should not be shown on that particular scale. In the case of the topological GAP face tree, too large objects would mean too much processing of the topological faces, to be shown. The mapping of scale to importance thus must select objects in relation to their size, that will fit the required level of detail.

It would be good, if importance levels have a relation to the area of the faces. The selection criteria of size of faces is also good to use in the case of merging faces, but when other generalization operators are considered, this probably is not the best choice any more. Therefore, more research is needed on: 1. how to steer the process of building the gap face tree (cf. research done by Van Putten, 1997), but also how this relates to 2. how to use the importance value for real time visualization.

Another approach was also tried, when trying to map scale to importance. This approach tries to map the importance level to a fixed amount of objects (as mentioned in (Van Oosterom, 2005)) per level of detail. Basically, this approach tries to count the amount of objects per importance level for a given spatial extent (i.e. the area shown on the screen of the user). Because it is too resource intensive to count for all importance levels how many objects there are in the given area, a better solution is needed; First, an approximation of which importance level can deliver the wanted amount of objects is generated from calculation (based on a mapping between importance and scale, as described above). After this calculation, a recursive function can be used, that tries to find from the database the importance level that exactly contains the amount of objects that is wanted (with a limited amount of queries).

However, this way the importance levels are made dependent on the spatial extent shown on the screen of the user: If a user pans from an urban area (containing a lot of small faces), to a rural area (containing lesser, and larger faces), it is possible that the system determines a different importance level, containing a better approximate for the wanted number of objects. This way, the amount of information on the screen is kept constant (according to the fixed amount of faces), but it may give unwanted side-effects for the interpretability and usability for the user that is operating the system, because the geographic information is constantly changing (not only with zooming in and out, but also with panning).

4.2.8 Lessons learnt from testing

This section shows some lessons that are learnt from testing the tGAP structure:

Size of the tGAP structure The implemented tGAP structure uses, with respect to physical size, around 8 times more space than the minimalistic source topology model. The two main tables that influence the logical size (w.r.t. number of new rows) of the tGAP structure the most are the tGAP edge and BLG table. The increase of the number of rows in the tGAP edge table is caused by storing the edges that are valid for a higher importance level with new rows (with partly changed information) in the tGAP edge table. Reduction in size of the tables can probably be reached, by using a changed physical model for the GAP edge forest (see next section for a proposal);

Loading speed The loading speed of the tGAP structure can be tuned, by looking at all the parts of the construction process and improving the parts that take longest;

Visualization speed The tGAP structure makes it possible to interactively, i.e. within 15 seconds of the start of the request, browse through the complete data sets

3D functional index Building a 3D functional index on a function with more than one parameter failed in this implementation experiment. However, the experiment shows that it is possible to build a 3D index on a function with one parameter. This 3D index has less overhead compared to a 2D index, combined with a 1D index;

Scale to importance It is needed to pay attention to the mapping between LOD and importance of the faces, not only for construction of the tGAP structure, but also for using the tGAP structure for visualization.

4.3 Possible theoretical improvements to the data structure

The implementation brings a way to reason about the data structures, as cases can be found in the implementation and testing, that will bring difficulties when applying the theory. In this respect, section 4.3.1 shows some proposed changes to the data model of the tGAP structure. Also, during literature study, some theoretical improvements are found, trying to solve some problematic cases as found in the data structures (sections 4.3.1–4.3.5). These features are not implemented nor tested and provide only ideas for future research directions.

4.3.1 Changing the data model with respect to the GAP edge forest

The size measurements show that the size increase of the tGAP structure (in comparison with the source topology tables) is caused by large growth of the tGAP edge and tGAP BLG table, which together form the GAP edge forest. To make the structure leaner with

respect to size requirements, it may be possible to save storage space with the GAP edge forest. In this section, some changes are proposed to save space with the GAP edge forest.

Besides the proposed change of Van Oosterom for storing the changing information of edges (i.e. low and high importance and left face and right face pointers) in varray's, some other things can be changed to the GAP edge forest. The edge information is currently stored in two tables, the tGAP edge and tGAP BLG table, both are implemented as described in listing 4.2, page 41.

The tGAP BLG table holds both original and joined BLGs. This implementation is convenient for the joins of the BLGs, which means looping over the BLG table. If the algorithm for joining of the BLG trees starts at a joined BLG tree, then the table is descended, until the given threshold is reached. If the threshold is set to zero, this means that also the original BLG trees will be used. With descension, the algorithm now does not need to switch to another table if not a joined, but an original BLG tree is encountered, because either a join, or an original BLG tree can be retrieved from the same table.

However, the current data model shows a lot of empty fields in the BLG table (this can also be observed in table 4.5 of the example). For example, in the datasets used, around 35 percent of the BLGs consists of joined BLGs (i.e. in this case the 'top tolerance' field in a row is set to something else than '-1'). Further, this also means that 65 percent of the rows (i.e. the original BLGs) does not have the child 1 and child 2 fields filled in, which again means a lot of empty fields.

It also means, that if the BLG table is split in two tables (one source BLG tree table and one table with joins of BLG trees), the top tolerance field can be dropped from the source BLG tree table, saving the storage space for 65 percent of one field.

Another thing is that there exists redundancy in the start and end node references of the BLGs with the given implementation. This also causes extra storage requirements. These start and end nodes are needed in the BLG table for the original BLG trees, because the BLGs themselves do not have a start or end node associated. In case of a joined BLG tree, the reference to the top node is stored twice (via the start and end node), which means redundancy.

Again, if the BLG table is split into two tables, then this redundancy of two node references pointing to the same node can be saved, if a 'top node' field is introduced in the table that holds only the joins of the BLGs. This thus means replacing the two start and end node fields for one top node field. Totally this will be a 35 percent storage reduction of one field. Together with the already saved space, this means that in total one column can be saved from the tGAP BLG table.

Also the BLG identifier can be dropped from the edge table, if the edge id and the blg id are kept the same. This would mean gaps in the 'edge id' sequence, but would save the need for the additional 'blg id' field (see the example tGAP edge table 4.4, in which the 'edge id' and 'blg id' already are kept the same).

To summarize, to reduce the amount of needed space, the model possibly can be improved in the following way:

- Leave the node table as it is;
- Change the face table, as it is shown in listing 4.10.

Listing 4.10: Proposed tGAP face table

```

1 face_id      number
2 imp_low     number
3 imp_high    number
4 area        number (probably: store in functional index)
5 mbr         number (probably: store in functional index)
6 parent_id   number

```

- Store the edge table as described in listing 4.11.

Listing 4.11: Proposed tGAP edge table

```

1 edge_id      number
2 imp_low     number
3 imp_high    varray (number)
4 start_node_id number
5 end_node_id  number
6 left_face_id varray
7 right_face_id varray

```

- Split the original BLGs and joined BLGs into two tables as follows in listing 4.12 and 4.13

Listing 4.12: Proposed tGAP original BLG table

```

1 blg_id       number
2 tree_source  blgtree

```

Listing 4.13: Proposed tGAP joined BLG table

```

1 blg_id       number
2 child1_id    number
3 child2_id    number
4 top_node     number
5 top_tolerance number

```

- Create a view on top of these two tables to allow the algorithm to descend in the same way as it is done now (listing 4.14). Eventually, if the start and end nodes are needed together with the BLGs, they can be joined to this view from the tGAP edge table.

Listing 4.14: Proposed tGAP BLG view, so that it appears as one table

```

1 create view
2   tgap_blg

```

```

3  as
4    (select
5      blg_id,
6      tree_source,
7      null as child1_id,
8      null as child2_id,
9      null as top_node,
10     null as top_tolerance
11   from
12     tgap_blg_original b)
13 union all
14   (select
15     blg_id,
16     null as tree_source,
17     child1_id,
18     child2_id,
19     top_node
20     top_tolerance
21   from
22     tgap_blg_joined);

```

4.3.2 Self-intersections of edges due to the Douglas-Peucker algorithm

In the original paper by Van Oosterom (2005) a problem that can occur when using the Douglas-Peucker algorithm is mentioned: the edges in the data structure could possibly be self-intersecting when the edges are represented by BLG trees. Implementation shows that this problem indeed occurs. Another problem of the BLG trees, is intersecting with each other (e.g. figure 4.12). Narrow features such as roads show this. A possible solution to this problem is presented in (De Berg et al., 1995).

However, improving the data structure, so that there does not exist any intersections within the edges, or within multiple edges, will be not straightforward to implement, because to solve this problem, one has to introduce the surroundings of the edges as a constraint for which generalization steps in the simplification of the edges are allowed. Because the data structure of the edges is fully dynamic and the amount of detail is dependent on a tolerance value that is chosen, also the surroundings of the edge when simplifying is changing with the change to use another tolerance.

Not only calculation will be problematic, but which intermediate vertices may or may not be removed, should then be stored inside the tree, so this information is available when visualizing. Because this information is needed within the tree, it is also needed for the pairwise joins of BLG trees. An extension for the BLG tree possibly should be proposed for this problem.

4.3.3 Smooth zooming, gradual changes in display

In (Sester and Brenner, 2004a) and (Sester and Brenner, 2004b), a method for continuous improvement of the displaying of geographic data on a screen is shown. This method



Figure 4.12: Intersection of BLGs

probably can be used for improving the data structures, by implementing the encoding of Sester and Brenner inside the data structures, because relationships are already stored between the faces and the edges in the tGAP structure. This should make it possible to continuously improve the view at the side of the client, so called continuous zoom. To supply the continuous zooming process with geographic data, possibly progressive transfer can be used, to deliver data just when it is needed.

In the work also the BLG tree is used for storing information on lines to be generalized. However, Sester and Brenner argue that the Douglas-Peucker algorithm is not the best approach for generalizing structured objects, like houses, because the appearance of the houses will be distorted. It would be interesting to see if it is possible to use the simplification methodology for houses used in this work, to be integrated within the tGAP structure.

4.3.4 Collapse of area objects to lines and points

Bobzien and Morgenstern (2002) describe a methodology for collapsing of topologically stored faces to edges and points and edges to points when using a topological data structure for generalization. At this moment this methodology is difficult to combine with the data structures in this research, as only a polygonal area partition is stored, while in the work of Bobzien and Morgenstern a complete topological feature model is in place.

The planar area partitions alone might be too narrow for certain applications that need collapsing of features. However, it may be possible to model features on top of the topological primitives of the area partition (thus store also thematic information to the nodes and edges). This will introduce extra complexity, specially if networked features

are modelled on top of the topological primitives, where constraints are placed on these features, e.g. that after generalization such a network is still connected. Furthermore, the data structures at this moment are implemented as a binary tree. For collapsing, it might be needed to leave the format of the binary tree, so that the data structure of the faces becomes a Directed Acyclic Graph (DAG), as is done in (Ai and Van Oosterom, 2002). This then makes it possible that a face will be split in two objects.

4.3.5 Simplification of Geographic Shape by Discrete Curve Evolution

During personal communication with Van Oosterom a suggestion was made that the approach described in (Barkowsky et al., 2000) could be used as a replacement for the Douglas Peucker algorithm. It was shown with an example that the methodology described in this paper for storing edge simplification can also be stored in a tree structure. It would be interesting to see if the tGAP structure indeed works with another algorithm for edge simplification.

The described algorithm is less calculation intensive than the Douglas-Peucker algorithm as the replacements for all edge segments only have to be calculated once. Therefore, less time will be needed to fill such a structure, than for the BLG tree data structure (although that this time requirement is currently not one of the bottlenecks of the use of the data structures). Also the visual characteristics of this algorithm might be different from the characteristics offered by the Douglas-Peucker algorithm and can possibly be applied to different data sets.

4.4 Final remarks

This chapter has shown that implementation of the data structures in a mainstream OR-DBMS is possible. With this, some remarks about the data structures and the original paper have to be made:

- The winged edge data structure, as proposed to use in (Van Oosterom, 2005) needs more references (more information to be stored) and is also more complicated to deal with in the build process of the GAP face tree. Therefore, left right topology is used in this research;
- Validation of source data (i.e. geometry validation and enforcing constraints) is needed to be sure to start with a valid topological area partition;
- The return polygon function should include a way to influence the amount of edge detail (i.e. take another parameter into account);
- It is important to take the direction of the BLGs into account in the tGAP structure;
- The 3 dimensional indexing mechanism offered by Oracle can be used to build a functional index (on one parameter). Two separate indexes (one on bounding

boxes and one on importance levels) also can make the data structures work, but the overhead associated with this approach is larger;

- The data structures take up around 8 times more space than a minimalistic topological model, when only comparing the size tables take;
- The complete size of the tGAP structure will benefit, if the GAP edge forest is slimmed down with respect to storage requirements;
- The tuning of the GAP face tree shows, that it is necessary to have timing data available for all parts of the system, to make it possible to tune the whole system;
- It is made possible by the data structures to navigate through a fairly large data set without performance problems, because a selection on the amount of detail can be made by the data structures;
- As is shown with Google Earth, the data structures can be used for visualization of geographic data in a networked environment and can implement on the fly generalization in such a networked setup;
- The implementation provides a means to reason about the data structures by which future research directions can be derived.

Chapter 5

Conclusions, recommendations and future research

5.1 Conclusions

Generalization remains one of the key issues in the data collection and map making process. With all developments around using networks for disseminating geographic information (so called Geographic Information Infrastructures) it gets even more important to generalize, to reduce the amount of data, before data is sent over the network.

However, generalization remains a time intensive operation and can not be performed in real time during user interaction, because of the complexity of the decision to be made by the generalization operators. Reactive data structures, such as researched in this thesis, take another approach by taking away the need to apply generalization operators in real time and storing results of these operators in a smart way, i.e. in reactive data structures.

The good things about having these data structures inside the DBMS is that if a reactive data structure, like the tGAP structure is used, it is possible to create an on the fly generalization, an approach that is suitable for implementing generalization in a GII. In this way, the data structures can be a stimulus to increase the use of large scale data. The data structures also offer a continuous range of levels of detail (LODs), instead of a limited, pre-defined set of levels (as found nowadays in MRDBs).

This research has shown that it is feasible and possible to implement these data structures in a mainstream database management system, with the use of tables, indices (normal, spatial and functional), spatial and custom data types, integrity constraints and a programming language. Theory for the structures as described in Van Oosterom (2005) served as a guideline and implementation of the conceptual model sketched, is possible. However, the implementation experiment and tests performed in this research delivered information to change the theory on some points and make suggestions for improvements (see next section). The performance of the data structures can be described, by measuring size (i.e. storage) and time (i.e. needed time for building) requirements. This way, it is possible to compare the tGAP structure with the source data

models, to see how size requirements increase when using the data structures, and with other approaches, like Multi-Resolution Databases (future research).

It is made possible by the data structures to navigate through a fairly large geographic data set without performance problems, because a selection on the amount of detail can be made by the data structures. The data structures can be setup in such a way that they can be used in a networked environment, to perform data reduction, before data is sent over the network.

The main conclusion of this research is that it is feasible to implement the data structures for generalization in a DBMS, when real time use of generalized data is needed.

5.2 Recommendations

Based on this research, the following recommendations can be given:

- The winged edge data structure does need more references and is more complicated to deal with in the construction process of the tGAP structure. Therefore, a better choice, to save storage space and avoid complexity with building the tGAP structure, is to use left right topology.
- When using the data structures, validation of source data (i.e. geometry validation and enforcing constraints) is needed to be sure to start with a valid topological area partition.
- The function to materialize geometry of the faces should have a way to influence the amount of edge detail (i.e. take another parameter into account).
- It is needed to take the direction of the BLGs into account.
- The 3 dimensional indexing mechanism, such as offered by Oracle, is an appropriate manner to be used as functional index, saving overhead when selecting the correct faces. Therefore, it should be used if it is available.

5.3 Future research

Future research can take place in the following directions:

- To investigate if the data structures can be used for gradual changes in presentation, so-called continuous zooming.
- Progressive transfer of geographic data, supported by the tGAP structure, is another option to investigate in further detail.
- To slim down the GAP edge forest in size, by applying a different physical model to the tGAP structure. Size matters, if the tGAP structure is used for progressive transfer of data, when the complete content of the data structure is replicated at the client side.

- The choice for importance values, and the mapping from level of detail to importance needs attention: how to choose relevant importance values, so that the building process can be steered, but also that it is easy to use the importance values for visualization.
- To make sure that the constraints for a topological area partition are enforced, more research is needed on how to prevent self-intersections of the BLG trees, as well as intersections of each other.
- See if it is possible if the data structures can be used for type change, i.e. changing an area object, into a line or a point object or a line object into a point object.
- More research is needed on how users experience continuous change when interacting with a scaleless data set, i.e. do users want to have multiple levels of detail, or want the behavior where always the same amount of faces is shown on the screen (and e.g. detail is changed when panning from rural to urban area).

Appendix A

Example of request–response sequence of Google Earth, Apache and Oracle

When a user interacts with Google Earth to retrieve geographic data from the tGAP structure stored in the DBMS, a chain of request and response actions follows. This appendix shows an example of one request. The request–response interaction is graphically depicted in figure 4.9 on page 53.

In listing A.1, a piece of the logfile of the Apache web server, is depicted and shows a request that has been made.

Listing A.1: Request from Google Earth to Apache over HTTP

```
1 130.161.233.101 - [14/Jun/2006:15:11:34 +0200] "GET /kml.py?BBOX
    =5.080946047654876,51.72451483349305,5.185669429687565,51.77546557848719
    HTTP/1.1" 200 1134 "GoogleEarthMac/LT3.1.0621.0"
```

In this logfile, the following information can be observed:

- Source IP address from the client;
- Date and time of the request;
- The contents of the request. The 'kml.py' script is requested from Apache, with 'BBOX' in WGS84 as a parameter. The request is made following the HTTP standard (version 1.1);
- State of returned answer: request is answered correctly (code 200), i.e. no failures occurred during processing the request;
- Size of returned KML document in bytes. In this case, the returned information is 1134 bytes;
- Request is made with the Google Earth client, running on the Macintosh platform.

After mapping the bounding box (BBOX), which was given as parameter, from WGS84 to RD and calculating which importance level is needed (in this case 160), the statement shown in listing A.2 is fired to the database.

Listing A.2: Database query fired from Apache to Oracle DBMS

```

1 select
2   face_id,
3   return_polygon(face_id, 160) as geometry
4 from
5   tgap_face
6 where
7   sdo_filter(
8     compute_3d_block(face_id),
9     mdsys.sdo_geometry(3003,
10      90112,
11      null,
12      mdsys.sdo_elem_info_array(1, 1003, 3),
13      mdsys.sdo_ordinate_array(133812.4,415137.3,160,141062.8,420830.2,160)
14    ), 'querytype=window') = 'true';

```

The database returns materialized polygon geometries in RD to Apache. The coordinates of all polygons are transformed from RD to WGS84 by Apache. After the coordinate transformation has been performed, the polygons can be retrieved into the memory of the Webserver as GML document, as shown in listing A.3¹.

Listing A.3: GML document, as available in the memory of the webserver after transformation

```

1 <?xml version="1.0" encoding="utf-8" ?>
2 <gml:featureCollection
3   xmlns:xlink="http://www.w3.org/1999/xlink"
4   xmlns:gml="http://www.opengis.net/gml">
5
6   <gml:featureMember>
7     <gml:Polygon>
8       <gml:outerBoundaryIs>
9         <gml:LinearRing>
10          <gml:coordinates>
11            5.151450562320917,51.742366757013613,20
12            5.151457206331063,51.742370051607502,20
13            ...
14            5.151938721841817,51.742139497836448,20
15            5.151450562320917,51.742366757013613,20
16          </gml:coordinates>
17        </gml:LinearRing>
18      </gml:outerBoundaryIs>
19    </gml:Polygon>
20  </gml:featureMember>

```

¹This GML might be invalid according to the OGC GML schema specifications, but as it is used as intermediate format, no validation has been performed

21
22 </gml:featureCollection>

The GML document residing in memory of the webserver can be transformed into a KML document with the eXtensible Stylesheet Language Transformations (XSLT). Listing A.4 shows the XSLT used for this job.

Listing A.4: GML to KML with XSLT

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <xsl:stylesheet version="1.0"
3   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4   xmlns:wfs="http://www.opengis.net/wfs"
5   xmlns:gml="http://www.opengis.net/gml"
6   xmlns="http://earth.google.com/kml/2.0"
7   exclude-result-prefixes="gml wfs">
8
9   <xsl:output method="xml" indent="yes" encoding="utf-8" />
10  <!-- ===== -->
11  <xsl:template match="/">
12    <kml>
13      <Placemark>
14
15        <name>topological Generalized Area Partitioning</name>
16
17        <Style id="s1">
18          <PolyStyle>
19            <color>4400ff00</color>
20          </PolyStyle>
21          <LineStyle>
22            <width>1.8</width>
23          </LineStyle>
24        </Style>
25
26        <MultiGeometry>
27          <xsl:for-each select="gml:featureCollection/gml:featureMember/gml:Polygon
28            ">
29            <Polygon>
30              <styleUrl>#s1</styleUrl>
31              <extrude>0</extrude>
32              <altitudeMode>relativeToGround</altitudeMode>
33              <xsl:for-each select="gml:outerBoundaryIs">
34                <outerBoundaryIs>
35                  <LinearRing>
36                    <coordinates>
37                      <xsl:value-of select="gml:LinearRing/gml:coordinates" />
38                    </coordinates>
39                  </LinearRing>
40                </outerBoundaryIs>
41              </xsl:for-each>

```

```

41     <xsl:for-each select="gml:innerBoundaryIs">
42         <innerBoundaryIs>
43             <LinearRing>
44                 <coordinates>
45                     <xsl:value-of select="gml:LinearRing/gml:coordinates" />
46                 </coordinates>
47             </LinearRing>
48         </innerBoundaryIs>
49     </xsl:for-each>
50 </Polygon>
51 </xsl:for-each>
52 </MultiGeometry>
53
54 </Placemark>
55 </kml>
56 </xsl:template>
57 <!-- ===== -->
58 </xsl:stylesheet>

```

After applying the XSLT to the GML document, a KML document is returned to Google Earth (listing A.5).

Listing A.5: Apache response to Google Earth in KML

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <kml
3   xmlns="http://earth.google.com/kml/2.0">
4   <Placemark>
5
6     <name>topological Generalized Area Partitioning</name>
7
8     <Style id="s1">
9       <PolyStyle>
10        <color>4400ff00</color>
11      </PolyStyle>
12      <LineStyle>
13        <width>1.8</width>
14      </LineStyle>
15    </Style>
16
17    <MultiGeometry>
18      <Polygon>
19        <styleUrl>#s1</styleUrl>
20        <extrude>0</extrude>
21        <altitudeMode>relativeToGround</altitudeMode>
22        <outerBoundaryIs>
23          <LinearRing>
24            <coordinates>
25              5.151450562320917,51.742366757013613,20
26              5.151457206331063,51.742370051607502,20

```

```

27         ...
28         5.151938721841817,51.742139497836448,20
29         5.151450562320917,51.742366757013613,20
30         </coordinates>
31     </LinearRing>
32 </outerBoundaryIs>
33 </Polygon>
34 </MultiGeometry>
35
36 </Placemark>
37 </kml>

```

Finally, Google Earth can draw the polygons inside the KML document onto the screen.

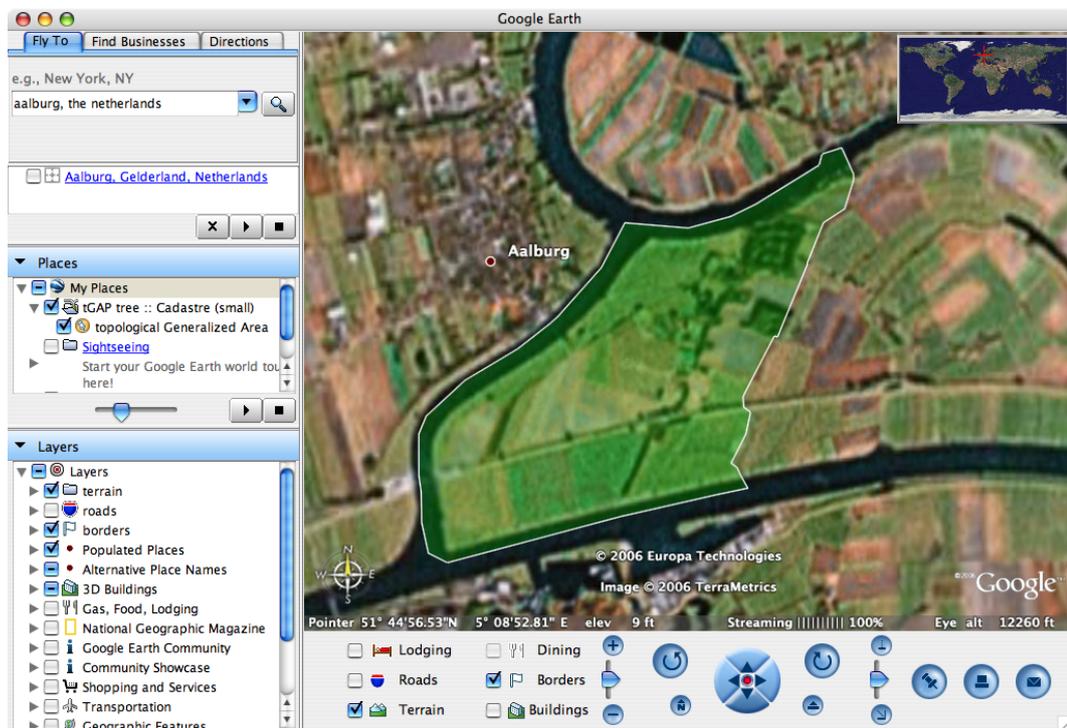


Figure A.1: Window of Google Earth after request–response sequence

Appendix B

Size of indices on the source topology and tGAP tables

dataset	table	columns	size (MB)
Cadastre (small)	face	id	.01
	edge	start node	.02
	edge	end node	.02
	edge	left face	.02
	edge	right face	.02
	edge	id	.02
	node	id	.01
	Cadastre (large)	face	id
edge		start node	3.12
edge		end node	3.11
edge		left face	3.51
edge		right face	3.51
edge		id	3.31
node		id	2.12
Amsterdam		face	id
	edge	start node	7.48
	edge	end node	7.48
	edge	left face	7.59
	edge	right face	7.46
	edge	id	7.10
	node	id	4.72

Table B.1: Index size of indices on non-spatial attributes of source topology structure.

dataset	table	index	size (MB)
Cadastre (small)	face	mbr	0.03
	face	3D function	0.04
Cadastre (large)	face	mbr	8.78
	face	3D function	11.49
Amsterdam	face	mbr	33.03
	face	3D function	35.57

Table B.2: Physical size of spatial (function based) indices. These indices are defined on the columns of the tGAP tables.

dataset	table	columns	size (MB)
Cadastre (small)	face	id (unique)	.01
	face	id, imp high, area	.02
	edge	id, imp low (unique)	.13
	edge	left face	.11
	edge	right face	.11
	edge	imp low	.05
	edge	imp high	.10
	blg	id (unique)	.02
	blg	start node	.02
	blg	end node	.02
	blg	id, child 1, child 2	.02
	node	id (unique)	.01
	Cadastre (large)	face	id (unique)
face		id, imp high, area	4.48
edge		id, imp low (unique)	101.82
edge		left face	93.65
edge		right face	94.63
edge		imp low	49.87
edge		imp high	88.93
blg		id (unique)	6.09
blg		start node	5.87
blg		end node	5.81
blg		id, child 1, child 2	7.95
node		id (unique)	2.12
Amsterdam		face	id (unique)
	face	id, imp high, area	14.06
	edge	id, imp low (unique)	223.16
	edge	left face	199.40
	edge	right face	196.53
	edge	imp low	110.55
	edge	imp high	187.72
	blg	id (unique)	13.62
	blg	start node	15.09
	blg	end node	15.30
	blg	id, child 1, child 2	17.81
	node	id (unique)	4.72

Table B.3: Index size of indices on non-spatial attributes of the tGAP structure

Bibliography

- Ai, T. and Van Oosterom, P. (2002). Gap-tree extensions based on skeletons. In *Advances in Spatial Data Handling*, pages 501–513. 10th International Symposium on Spatial Data Handling.
- Apache Software Foundation (2006). Mod_python - Apache/Python Integration. <http://www.modpython.org/>.
- Barkowsky, T., Latecki, L. J., and Richter, K.-F. (2000). Schematizing maps: Simplification of geographic shape by discrete curve evolution. *Lecture Notes in Computer Science*, 1849:41–53.
- Bobzien, M. and Morgenstern, D. (2002). Geometry-type change in model generalization - a geometrical or a topological problem? In *Multi-Scale Representations of Spatial Data*, Ottawa. Joint ISPRS/ICA Workshop.
- Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387.
- De Berg, M., Van Kreveld, M., and Schirra, S. (1995). A new approach to subdivision simplification. In *Twelfth International Symposium on Computer-Assisted Cartography*, volume 4, pages 79–88, Charlotte, North Carolina.
- Douglas, D. and Peucker, T. (1973). Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *The Canadian Cartographer*, 10(2):112–122.
- E.F. Glynn II (2005). efg's Graphics – Polygon Area and Centroid. <http://www.efg2.com/Lab/Graphics/PolygonArea.htm>.
- Galanda, M. (2003). *Automated Polygon Generalization in a Multi Agent System*. PhD thesis, University of Zürich.
- Google (2005). Google Earth KML 2.0. http://www.keyhole.com/kml/docs/Google_Earth_KML.pdf.
- Greenfield, A. (2006). *Everyware : The Dawning Age of Ubiquitous Computing*. New Riders Press.

- ICA (1973). *Multilingual Dictionary of Technical Terms in Cartography*. Franz Steiner Verlag, Wiesbaden.
- Lagrange, J., Ruas, A., and Bender, L. (1993). Survey on generalization. Technical report, IGN.
- Longley, P. A., Goodchild, M. F., Maguire, D. J., and Rhind, D. W. (2005). *Geographical Information Systems and Science*. John Wiley & Sons, Ltd.
- Mark, D. (2000). Geographic information science: Critical issues in an emerging cross-disciplinary research domain. *Journal of the Urban and Regional Information Systems Association*, 12(1):45–54.
- Mark, D. M. (2003). *Geographic Information Science: Defining the Field*, chapter 1, pages 3–18. Foundations of Geographic Information Science. Taylor & Francis, London.
- McMaster, R. B. and Shea, K. S. (1992). *Generalization in Digital Cartography*. Association of American Geographers.
- Müller, J., Lagrange, J., and Weibel, R., editors (1995). *GIS and Generalization: Methodology and Practice*. Number 1 in GISDATA. Taylor & Francis.
- OGC (2003). OpenGIS® Geography Markup Language (GML) Implementation Specification, version 3.00. <http://www.opengeospatial.org/docs/02-023r4.pdf>.
- Open Geospatial Consortium (2005). OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 2: SQL Option.
- Porcino, D. and Hirt, W. (2003). Ultra-wideband radio technology: potential and challenges ahead. *IEEE Communications Magazine*, pages 66–74.
- Quak, W., Stoter, J., and Tijssen, T. (2003). Topology in spatial DBMSs. Brno. The 3rd international symposium on digital earth: information resources for global sustainability.
- Ralston, A. and Reilly, E. D., editors (1993). *Encyclopedia of computer science (3rd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA.
- Sester, M. and Brenner, C. (2004a). Continuous generalization for fast and smooth visualization on small displays. In *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, volume 35. ISPRS.
- Sester, M. and Brenner, C. (2004b). Continuous generalization for visualization on small mobile devices. In Fisher, P., editor, *Developments in Spatial Data Handling*, pages 469–480. Springer Verlag.
- Shuman, B. (1992). *Foundations and Issues in Library and Information Science*. Libraries Unlimited Inc., U.S., Colorado.

- Stoter, J., Kraak, M., and Knippers, R. (2004). Generalisation of framework data: a research agenda. In *ICA Workshop on "Generalisation and Multiple representation"*, Leicester.
- Thewessen, T. and Brentjens, T. (2005). Are you being served? *Geo-Info*, 2(6):280–287. In Dutch.
- Tichy, W. F. (1998). Should computer scientists experiment more? *IEEE Computer*, 31(5):32–40.
- Töpfer, F. and Pillewizer, W. (1966). The principles of selection, a means of cartographic generalization. *Cartographic Journal*, 3(1):10–16.
- Uitermark, H. (2001). *Ontology-based geographic data set integration*. PhD thesis, University of Twente.
- Van Oosterom, P. (1990). *Reactive Data Structures for Geographic Information Systems*. PhD thesis, Leiden University.
- Van Oosterom, P. (1991). The reactive-tree: A storage structure for a seamless, scaleless geographic database. In *Proceedings Auto-Carto 10*, Baltimore, Maryland. Cartography and Geographic Information Society (CaGIS).
- Van Oosterom, P. (1995). *The GAP-tree, an approach to "On-the-Fly" Map Generalization of an Area Partitioning*, volume GIS and Generalization: Methodology and Practice of GISDATA, chapter 9, pages 120–132. Taylor & Francis, London.
- Van Oosterom, P. (2001). De geo-database als spin in het web. Delft. In Dutch.
- Van Oosterom, P. (2005). Variable-scale topological data structures suitable for progressive data transfer: The gap-face tree and gap-edge forest. *Cartography and Geographic Information Science*, 32(4):331–346.
- Van Oosterom, P., Quak, C., and Tijssen, T. (2003). Polygons: the unstable foundation of spatial modeling. In *ISPRS Joint Workshop on 'Spatial, Temporal and Multi-Dimensional Data Modelling and Analysis'*, Québec.
- Van Oosterom, P., Quak, W., Tijssen, T., and Verbree, E. (2000). The architecture of the geo-information infrastructure. In Fendel, E., editor, *Proceedings of UDMS 2000, 22nd Urban Data Management Symposium*, pages II.9–II.18, Delft.
- Van Oosterom, P. and Schenkelaars, V. (1995). The development of an interactive multi-scale gis. *International Journal of Geographical Information Systems*, 9(5):489–507.
- Van Oosterom, P. and Vijlbrief, T. (1996). The Spatial Location Code. In *The Seventh International Symposium on Spatial Data Handling*.
- Van Putten, J. (1997). Experiences with the gap-tree. Master's thesis, University of Utrecht.

- Van Putten, J. and Van Oosterom, P. (2000a). Generaliseren van vlakkenpartities (1): Gap-trees, theorie en implementatie. *Geodesia*, 42(10):443–448. In Dutch.
- Van Putten, J. and Van Oosterom, P. (2000b). Generaliseren van vlakkenpartities (2): Gap-trees, testresultaten en verbeteringen. *Geodesia*, 42(11):499–505. In Dutch.
- Van Smaalen, J. (2003). *Automated Aggregation of Geographic Objects: A New Approach to the Conceptual Generalisation of Geographic Databases*. PhD thesis, Wageningen University and Research Centre.
- Vermeij, M. (2003). Development of a topological data structure for on-the-fly map generalization. Master's thesis, Delft University of Technology.
- Verschuren, P. and Doorewaard, H. (1999). *Designing a Research Project*. Lemma, Utrecht.
- W3C (2001). Extensible Stylesheet Language (XSL). <http://www.w3.org/TR/xsl/>.
- W3C (2004). Extensible Markup Language (XML). <http://www.w3.org/TR/REC-xml/>.
- Warmerdam, F. (2006). OGR Simple Features Library. <http://www.gdal.org/ogr/>.
- Weibel, R. (1997). *Algorithmic Foundations of Geographic Information Systems*, volume 1340 of *Lecture Notes in Computer Science*, chapter 5, pages pp. 99 – 152. Springer Verlag.
- Worboys, M. F. (1995). *GIS: A Computing Perspective*. Taylor & Francis, London.