Storage and Manipulation of Topology in Postgres

Sylvia de Hoop,
Centre for Geographical Information Processing,
Wageningen Agricultural University,
P.O. Box 339, 6700 AH Wageningen, The Netherlands.
Email: hoop@rcl.wau.nl.

and

Peter van Oosterom
TNO Physics and Electronics Laboratory,
P.O. Box 96864, 2509 JG The Hague, The Netherlands.
Email: oosterom@fel.tno.nl.

In this paper two different approaches towards topology in a Geographic Information System are described. These two are the formal data structure *for single-valued vector maps of Molenaar and a* TIGER*-based approach for storing multiple (polygon network) map layers. A significant aspect of the fds is that in addition to topology at primitive level, it also models topology at feature level which is the level users normally work. The emphasis is on the actual implementation of topology in the extensible database management system Postgres. The two implementations are compared to each other with the use of some sample data sets. Finally, a suggestion is given how the two approaches can be integrated whilst keeping the strengths of both. This should result in a solid basis for building an advanced Geographic Information System.*

1 Introduction

Two approaches towards topology in a Geographic Information System (GIS) are described, implemented, and compared. The first approach is based on the *formal data structure* (fds) for single-valued vector maps [9] of Molenaar at Wageningen Agricultural University (WAU). The second approach towards topology is taken at TNO and is based on the more traditional *TIGER* structure [1], limited in this paper to area features. However, the model is extended in such a manner that it can deal with multiple layers.

WAU and TNO cooperate in a project to compare the two approaches. The first phase of the project deals with the actual implementation of both topological structures in the same (relational) environment. For GIS-purposes the relational data model must be extended with geographic data types, spatial operators, and multi-dimensional index structures. Postgres [18] is an open database and offers extension possibilities, and is therefore a suitable environment. The actual implementation of the topology models results in two database schema's. In the next phase of the project the two approaches are compared. The last phase of the project deals with the integration of both approaches.

Postgres can be considered as a Relational Database Management System (RDBMS) extended with object-oriented features. Some important characteristics of Postgres are given in section 2. Section 3 gives a description of the fds for single-valued vector maps and its implementation in Postgres, the topology of terrain features is analyzed in an example. In the next section a variant of the TIGER approach for multi-layer topological data structures is described together with a map overlay example. Section 5 compares the fds and the TIGER-based approach. Finally, the conclusions and future work are described in Section 6.

2 Short description of Postgres

Traditional RDBMSs cannot meet all the requirements of GISs. For GIS-purposes the relational data model should be extended with geographic data types, spatial operators and spatial (multi-dimensional) index structures [22]. In contrast with most DBMSs, Postgres is an open system. This means new data types, functions, operators, and index structures can be added to the system. Postgres users can interact with their databases by using the query language Postquel. The three main concepts in Postquel are [15, 16]:

- data types, namely base types, array types (fixed and variable length) and composite types;
- functions: normal functions in C or Postquel, operators, aggregate functions (e.g. sum, average, count, max) and inheritable functions (methods);
- rules to trigger DBMS actions; they have the form: on condition then do action. Two rule systems are available: Instance-level and Query Rewrite.

The fundamental idea in Postgres is that of a class (relation or table). A class defines an entity type and comprises a set of instances (records or tuples) of that entity type. Postgres gives each instance a unique identifier (`oid`), which cannot be changed by a user. It is possible to use the identifier of one record as a data item in a second record. In Postgres a class can inherit the class structure from one or more other classes. User defined Postquel or C-functions can have a class name as argument. When these functions have been registered to Postgres, they can be considered as a method for each instance of the class or as a new attribute for the class.

Postgres contains four primary geometric data types: `point`, `lseg`, `path` and `box`. The type `lseg` implements a single line segment. Polylines and polygons are both represented by `path`. The `box` represents a two-dimensional axes-parallel rectangle. The R-tree [5] is available for spatially indexing classes with a `box` attribute. Postgres supports historical queries and therefore has to store all the old data. A record that is deleted, is normally no longer visible for a user. However, this record can still be retrieved with a historical query:

```
retrieve (a.all) from a in area_feat[January 10, 1991]
```

The storage space is occupied with historical data until the relation is purged. Postgres may require a huge storage capacity for historic data which could best be archived on a writable optical (WORM or MO) disk. The current data remains on the hard disk.

3 A formal datastructure for single valued vector maps

A vector map is a vector structured terrain description, in the sense of a geographic data set not in the sense of a physical map. The term single-valuedness will be explained later, but it may roughly be interpreted as one map layer. The formal datastructure of single-valued vector maps, described in this section, is a (terrain) feature-oriented data model. The terrain features are defined by geometric and thematic attributes. The latter are organized in a hierarchical classification scheme. Geometric data covers four aspects: location (coordinates), size, shape and topology information. Location, size, and shape information together form the *metric* data. There are three basic types of topological relationships: connectivity, adjacency, and inclusion [21]. A unique aspect of the fds is that it can deal with topology at different semantic levels, i.e. it captures topology at a geometric level, feature level and in-between. Topology between terrain features is most important to end-users.

A complete description of the formal datastructure can be found in [9]. Subsection 3.1 gives a short summary. The implementation aspects are described in Subsection 3.2 which gives a definition of the required relations in Postgres. A sample data set is given in Subsection 3.3 together with two characteristic topological queries.

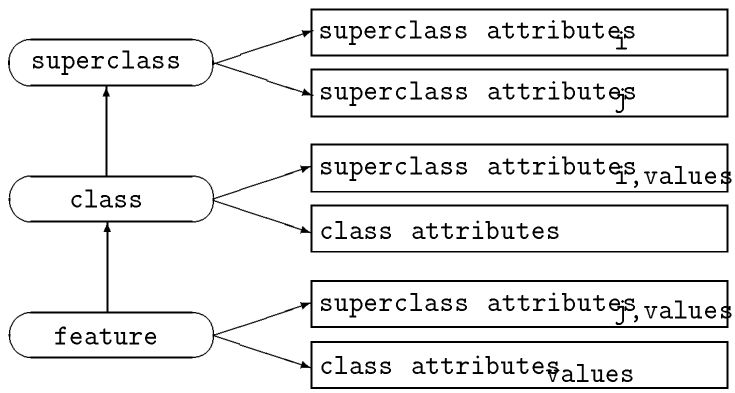

Figure 1: The classification scheme of terrain features

3.1 Short description of the fds

The fds is based on three main concepts [9]:

- terrain features: point features, line features, area features;
- thematic attributes of the terrain features;
- geometric attributes of the terrain features.

The thematic attributes are organized in a hierarchical classification scheme (Figure 1). The scheme refers to generalization and specialization operations on one type of the terrain features. The term (super)class in the classification scheme should not be confused with the definition of a class in Postgres. A class here represents a thematic description of a specific abstraction level of terrain features, independent of its implementation in a database model. The upward links in the scheme, which denote generalization, represent `is_a` links. E.g., the instance Rhine belongs to the class of rivers, while rivers belong to the (super)class of waterways. Therefore, the river Rhine `is_a` waterway. Note that the `is_a` relationship should not be confused with the `part_of` relationship, which is also often used in data modelling. For example, municipalities are `part_of` a province.

The class of waterways can also contain for instance a class of channels. The class of rivers and the class of channels inherit the thematic attribute structure of the class of waterways, but both can have some distinct thematic attributes in addition [10]. Postgres supports this inheritance notion [15]:

```
create waterway (depth=int4, name=char16, ....)
create river    (velocity=float8, ....) inherits waterway
create channel  (construct=int4, ....)  inherits waterway
```

In Postquel queries can be formulated that refer to instances of a class and optionally also to all of the descendents of the class. The latter can be attained by putting a star (*

after the class name concerned, for instance the following query selects all the rivers and channels that are deeper than 4 meters:

```
retrieve (w.all) from w in waterway* where w.depth > 4
```

In the remainder of this paper, the classification scheme will be restricted to one point class, one line class and one area class. The geometric structure of a vector map provides information about the shape, size and location of terrain features and their topological relationships. The geometric attributes of a vector map are analyzed through the graph-structure of the map. The primitives of the fds are arcs and nodes. A vector map will be called a single-valued vector map if it is structured like the fds and if it complies with the following seven conventions [9]:

1. The feature classes must be mutually exclusive, this means that each terrain feature has only one class label.
2. A feature class contains features of only one type.
3. When the map is analyzed as a graph, all points used to describe the geometry of a vector map will be treated as nodes.
4. The arcs of this graph are geometrically represented by segments of straight lines.
5. For each pair of nodes there is at most one arc connecting them. In addition to that the nodes may be connected by one or more chains.
6. In the geometric representation two arcs may not intersect.
7. For each geometric element there is only one occurrence of each of its link-types to a feature.

¿From the conventions follows that the fds represents one thematic map layer, which contains several different mutually exclusive thematic feature classes [11]. Figure 2 represents the fds for a single-valued vector map. The sets of data types are represented by ellipses. The link-types are represented by arrows. Each arrow indicates a one-to-many relationship, for instance many arcs can have at their left side the same area feature. Three levels of topological relationships can be distinguished in single-valued vector maps [9]:

- Low level topology: the relationships between the primitives as given by the graph-structure of the vector map.
- The linkage of the primitives of the vector map (the arcs and nodes) with the terrain features.
- High level topology: the relationships between the terrain features.

The first two mentioned levels of topological relationships (low levels) give geometric information about terrain features. The last mentioned level of topological relationships (high level) defines topology at feature level. The high level topology allows communication with the GIS at user-level rather then at system-level [11], e.g. an end-user can state the query: "Give all the line features that branch off the given line feature `Road_1`." In order to

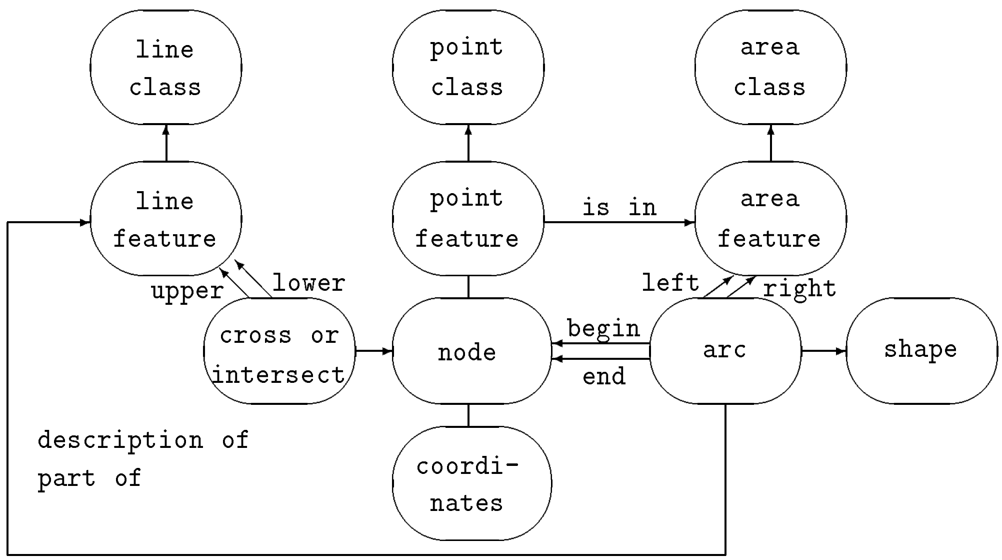

Figure 2: The fds for a Single Valued Vector Map

solve this query, the GIS has to use the topology directly related to the geometric primitives (a low level). As can be drawn from this query example, the high level topological feature relationships can be decomposed in low level topological relationships. Some examples are implemented in Postgres and described in Subsection 3.3. Low level topology may still be used directly as in the query: "What are the end nodes of a given arc."

3.2 Definition of the fds-relations in Postgres

In accordance with Bouloucos et al. [2] the fds has been interpreted into a set of fully normalized relations. The shape of arcs and the possibility of crossing or intersecting line features (Figure 2) were not considered. These data types have been omitted from the relations. In the first attempt, the object-oriented features of Postgres were not used. The following five relations were derived:

```
area_feat  (aid = int4, aclass = char[])
line_feat  (lid = int4, lclass = char[])
point_feat (pid = int4, pclass = char[],
            paid = int4, node_nr = int4)
arc        (from_node_nr = int4, to_node_nr = int4,
            left_aid = int4, right_aid = int4,
            lid = int4)
node       (node_nr = int4, x = float4, y = float4)
```

Note that `[]` indicates a variable length array. The area identifier `aid`, line identifier `lid` and point identifier `pid` will be called user identifiers. This in contrast to the system identifiers (`oid`) of Postgres. This system `oid` can be used in the relations instead of the

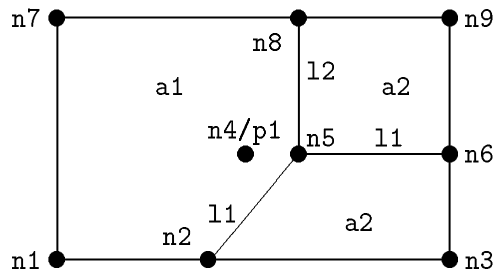

Figure 3: A sample map

user identifiers. The `x` and `y` coordinates in relation `node` can be replaced by the data type `point`. This leads to the following five, somewhat simpler, relations:

```
area_feat  (aclass = char[])
line_feat  (lclass = char[])
point_feat (pclass = char[], paid = oid, node_nr = oid)
arc        (from_node_nr = oid, to_node_nr = oid,
            left_aid = oid, right_aid = oid,
            lid = oid)
node       (location = point)
```

The advantage of this approach is that the user does not have to generate and administrate the identifiers of the nodes and of the features point, line, and area. The use of system generated identifiers, avoids errors such as giving the same id to different features. The use of the type `point` in the node relation instead of the separate `x` and `y` coordinates, enables the user to treat this attribute as a *true* point. For example, the distance operator (symbol is `<-->`) can be used in the query "Find all nodes within 5 length units from the origin:"

```
retrieve (node.all)
where (node.location <--> "(0,0)"::point) < 5
```

3.3 An example

The relations defined in the previous section will be filled up with the data represented in the vector map of Figure 3. In Postgres, rules can be defined for data entry and consistency checks (Subsection 4.3). The data can be entered in the relations in different ways, for instance a user can give the following information per feature:

- For area features: coordinates of an orientation point in the area and the class name.
- For line features: coordinates of the `from_node` and the `to_node` of the `arcs`, the class name of the line feature and the class names of the areas at the left side and the right side of the arc.

- For point features: the point coordinates and its class name.

This information is sufficient to fill up the relations correctly. The tables are filled with the data given in Appendix A. The topology in single-valued vector maps (Subsection 3.1) can be analyzed for the terrain features described in these filled-up relations. Two examples of topological queries will be given, formulated in Postgres using the pure relational database schema. In the first example a line/area relationship is analyzed. The question is "Give the areas in which the `Road_1` can be found." This question can be answered by joining the `line_feat`, the `area_feat`, and the `arc` relations:

```
retrieve unique (area_feat.aid)
where line_feat.lclass = "Road_1" and line_feat.lid = arc.lid and
   (area_feat.aid = arc.left_aid or area_feat.aid = arc.right_aid)
```

For this query a Postquel function may be defined: `which_area (lclass)`. This is the first step in the direction of a *topological query language*. The Postquel function can be used more easily when an unary operator is associated with it: `A?` which operates on one right hand side operand of type `lclass`. The expression `A? "Road_1"` returns the `aids` of the area features that contain this `Road_1`. So the names of the area features that contain the `Road_1` are obtained by the query:

```
retrieve (area_feat.class)
where area_feat.aid in_set (A? "Road_1")
```

Note that the implementation of the Postquel function `which_area` is dependent on the database schema. Knowledge about the attributes (e.g. `arc.left_aid`) is used in the Postquel function `which_area`. This might in fact be true for most components of the topological query language in contrast to the metric query language, which works independent of the database schema. E.g., the point distance `<-->` operator only depends on the data type `point`. Problems with misspelled or forgotten attributes used for the topological structure, can be avoided by regarding these attributes as standard attributes which are always present. The topology attributes might not even be visible to the end-user by using *views*. In this case, all topological manipulation should be performed through the use of the topological query language.

The second example deals with a line/line relationship: Give all the line features that branch off `Road_1`. This question is answered and represented in three steps:

```
/* Select the nodes of "Road_1".
 */
retrieve into hr1(ar.from_node, ar.to_node)
from l in line_feat, ar in arc
where l.lclass = "Road_1" and l.lid = ar.lid

/* Now search the arcs that are connected to one of the nodes of "Road_1".
 */
```

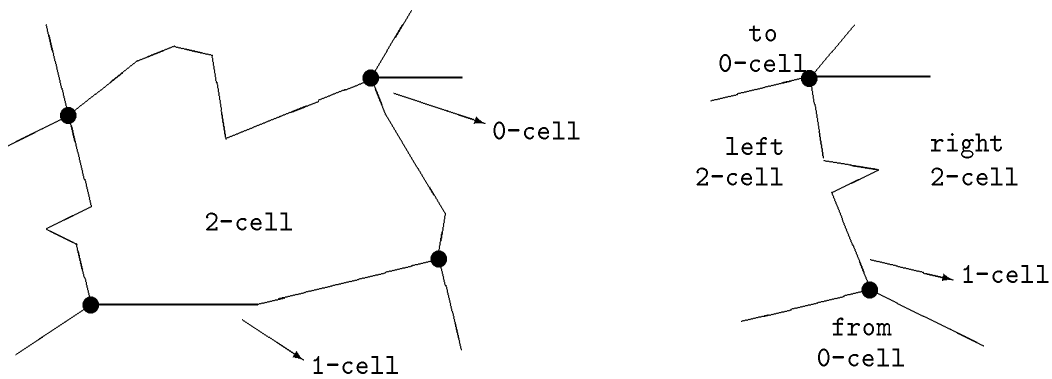

Figure 4: The "low level" topology in TIGER

```
retrieve into hr2 unique (ar.lid, ar.from_node, ar.to_node)
from l in line_feat, ar in arc
where (hr1.from_node = ar.from_node or hr1.to_node = ar.to_node or
   hr1.to_node = ar.from_node or hr1.from_node = ar.to_node) and
   ((l.lclass != "Road_1" or l.lclass != "Border") and (l.lid = ar.lid))

/* Present the line features that branch off line feature "Road_1".
 */
retrieve (l.lclass)
from l in line_feat
where hr2.lid = l.lid
```

4 Topology and Map Layers

The second approach towards topology described in this paper is taken at TNO and is based on the more traditional *TIGER* structure [1]. In contrast to the fds, now only area features are considered and the emphasis is on low level topology; see Figure 4. However, the model is extended in such a manner that it can deal with multiple layers each representing a different theme. For example, one polygon layer describes the land-use while another layer describes the soil type. Attention is paid to map-overlay operations and queries, such as: "Give all regions with soil type *sand* and land-use *industry*." Though metric data may be shared by multiple map layers, the topological information belongs to one map layer.

The Topologically Integrated Geographic Encoding and Referencing (*TIGER*) System, is developed by the United States Bureau of the Census [1, 7]. The description of the TIGER structure in Subsection 4.1 is based on [1]. Subsection 4.2 describes the implementation of this structure in the case of multiple map layers. Subsection 4.3 gives an impression of how topological structured data may be entered. Finally, a map overlay example of two polygonal map layers is given in the last subsection.

4.1 The TIGER Structure

The TIGER structure is the refined successor of older *chain-node* structures, such as DIME also developed by the US Bureau of the Census [20], and POLYVRT [14] developed at the Harvard Laboratory for Computer Graphics and Spatial Analysis.

When storing polygonal areas, the structure captures the relationships between the polygon, polyline and point primitives, which are called 2-cells, 1-cells and 0-cells respectively. A 1-cell is a polyline, the first and the last points of which are by definition 0-cells. These 0-cells are called *from* and *to* 0-cells and they are the same in case of a *loop* 1-cell. The points of a 1-cell between the 0-cells are called *curvature points* and are connected by vectors. A 1-cell has two sides, a *left* and *right* 2-cell, see Figure 4. The complex of 1-cells, that is, the set of all 1-cells, covers a finite region of the two-dimensional space. The unbounded region that surrounds the 1-cell complex is represented by a 2-cell labeled with a special code: *outside*. In addition to these definitions, the topological structure must obey two rules. The rule of *Topological Completeness* requires that the topological relationships between cells are complete, for example each 2-cell, except *outside*, is completely surrounded by a set of "connected" 1-cells. The rule of *Topological-Geometric Consistency* requires a consistent relationship between the geometric placement of cells and the pure topological relationships of cells, for example no two 2-cell interiors share a common coordinate.

4.2 The implementation of multiple map layers

An important function in GISs is the map overlay operation: two maps layers are combined into one new map layer. Therefore, it must be possible to represent multiple layers in the data model. For purpose of discussion, the map layers are limited to topologically structured polygon networks. The relations that interpret the TIGER structured networks, are somewhat related to the ideas of Van Roessel [24]. The representation is based on two levels: the metric information and the topological relationships. The metric level is described by three relations corresponding to the elementary primitives point, polyline and polygon:

```
Points    (pnt = point)
Polylines (pln = path)
Polygons  (pgn = oid[], mbr = box)
```

To avoid needless storage of copies of the polyline, the definition of the polygon uses references to the polylines. The first attribute is a variable length array that orders the polylines by referring to their system identifier (`oid`). Adding an attribute for the minimal bounding rectangle, makes the use of an R-tree index possible, which speeds up spatial queries. This construction may also be used for the other classes.

The `MapLayer` relation specifies the map layers in the database. The topological level relates spatial elements in one map layer to each other. Following the TIGER naming convention, the relations are[1]:

```
MapLayer (name = char[])
NullCell (pntoid = oid, layer = oid)
OneCell  (plnoid = oid, layer = oid, fromnode = oid,
          tonode = oid, leftpgn = oid, rigthpgn = oid)
TwoCell  (pgnoid = oid, layer = oid, thematic = char[])
```

These topological relations contain references to the metric elements. The topological relationships in two different map layers, described in the topological relations, can refer to the same metric element. Each topological relation contains the object identifier of the layer (`oid`).

4.3 Creating a map layer

Creating a topologically structured map layer is done with the aid of a separate program, that creates a polygon network with a set of polylines and the terrain class name as input. The program takes the following steps[2]:

- for each polyline, it appends an entry to the `Polylines` table;
- when all polylines are inserted, it executes an algorithm which "structures" the map layer if the input is correct.

The addition of instances to the `Polylines` relation causes the relations `Points`, `OneCell`, and `TwoCell` to be populated with instances. The rule mechanism of Postgres (Section 2) generates actions to add instances to the other relations; for example, the rule `AddPoints1`, for appending the start point of a polyline to the Points relation:

```
define rule AddPoint1 is
    on append to Polylines
    do append Points(pnt = PointsSelector(new.pln, 1))
```

The integrity is also kept by rules. For example, rules that check the presence of elements in a relation before adding them; e.g.:

[1]An alternative to including the `layer` attribute in topological relations is having separate relations (with the same structure) for each map layer: `NullCell_l1`, `OneCell_l1`, `TwoCell_l1`, `NullCell_l2`, `OneCell_l2`, and `TwoCell_l2`.

[2]An alternative approach is an incremental procedure. This procedure guarantees that after each topological edit operation the structure remains correct

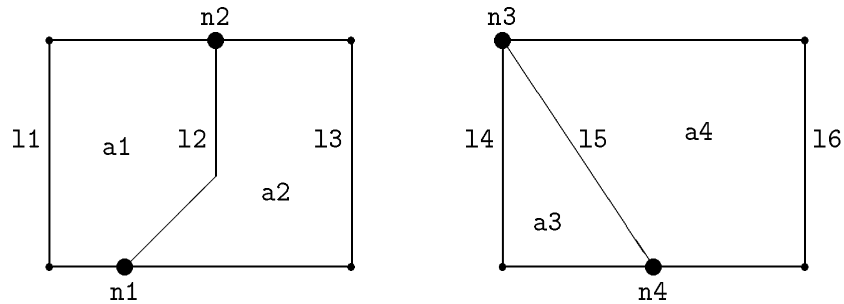

Figure 5: Two map layers in the same area

```
define rule CheckEqualPlines is
    on append to Polylines
       where Equal2PlnPln(new.pln, Polylines.pln)
    do instead nothing
```

The rules to insert entries to the topological relations `OneCell` and `NullCell` are a little more complicated, they are described in [8]. The `Points`, `Polylines`, `NullCell` and `OneCell` relations are used in the next stage to populate the `Polygons` and the `TwoCell` relations by the program that constructs them (if no errors occur by input). This program also fills the `leftpgn` and `rightpgn` attributes of `OneCell`.

A typical operation during the construction of a polygonal network is the ordering of all `OneCell`s (polylines) that are connected to a `NullCell` (point). This is often done on basis of the angle that the polyline makes with the positive x-axis. However, the angles are not stored in the `Polylines` table. Defining two new functions for the type `path`, that is, extending the abstract data type `path` with the functions `out_angle` and `in_angle`, solves the problem. An example shows that it is now possible to use the angles in a query:

```
retrieve (angle = in_angle (Polyline.pln))
```

4.4 An example

The two map layers depicted in Figure 5 are loaded into the database according to the procedure described in the previous subsection. Note that the tables in Appendix B also show the system `oid`s and that the metric data is shared among the two map layers. The `oid`s are not shown in the figure, because of its clarity.

Now the map overlay operation can be performed. This results in some new `NullCell`s caused by the intersection of the different layers. The map overlay produces a new MapLayer (called `landuse+soiltype`). This layer can be queried as any other map layer. For example, retrieve all regions that have a `sand` soiltype and have `agricultural` land-use:

```
retrieve (Polygons.oid)
```

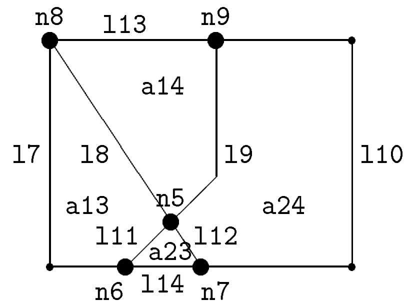

Figure 6: The result of map overlay

```
where TwoCell.pgnoid = Polygons.oid and TwoCell.thematic = "agric+sand" and
  TwoCell.layer = Maplayer.oid and Maplayer.name = "landuse+soiltype"
```

Figure 6 shows the result of the map overlay operation. Note that the topology is not shared by the different map layers, including the new one. This in contrast to the metric which is shared. The second part of Appendix B shows the new tuples as a result of the overlay.

5 Comparison

In this section the formal datastructure is compared with the TIGER-based structure, references are made to their interpretation in Postgres. Although the fds resembles the TIGER-based structure, there are some clear differences. In the fds two main semantic levels can be distinguished: primitive level and feature level. The feature level deals with three types of terrain features and their classification. The thematic feature classification is provided by means of the relations `area_feat`, `line_feat`, and `point_feat`. The three different terrain features are integrated in one map layer. Our implementation of the TIGER structure was limited at feature level to area features. The thematic aspect in the TIGER-based solution is reflected in the `thematic` string attribute of a `TwoCell`. In contrast with the fds, the TIGER-based structure is applied to multiple layers.

The principal differences between the fds and the TIGER-based structure can be found at the metric level. At this level the TIGER-based structure makes a clear distinction between metric and topology, whereas the fds mingles them. Pure metric is represented by the `node` relation of the fds and the `Points`, `Polylines`, and `Polygons` relations of the TIGER-based structure. Low level topology is described by the `arc` relation of the fds and the following relations of the TIGER-based structure: `NullCell`, `OneCell`, and `TwoCell`.

Area features and polygons (`TwoCells`) in the TIGER-based structure have a one-to-one relationship, because the structure regards only area features, so indirectly area features and polylines (`OneCells`) have a one-to-many relationship. The `Polygons` relation is redundant, because the set of polylines forming one polygon can be derived from the tables:

OneCell, TwoCell, and Polygons. However, this redundant storage makes the manipulation of polygons easier and rules can be defined to guard the consistency of the database. Polylines and polygons lack in the fds. Line features in the fds may consist of several arcs, for instance line feature l1 in Figure 2. Similarly, area features may also consist of several polygons, this is shown by area feature a2 in Figure 2.

The fds makes no distinction between nodes and vertices (curvature points of an arc). All nodes, including vertices, are gathered in one node relation. In the TIGER-based relations all the vertices and the end points (nodes) are stored in the Polylines relation with the pln-attribute of type path, and the nodes are also stored in the Points relation. The use of the data type polyline leads to a decrease in the number of nodes compared to the fds.

The topological relationship between a point feature and an area feature is explicitly stored in the fds. This is not the case in the TIGER-based approach, but high level topology can also be derived from the metric properties of the terrain features.

Careful analysis of the two implementations shows that three groups of relations can be identified:

- *feature classification* relations, emphasized in the fds approach;
- *metric* relations, emphasized in the TIGER based approach; and
- *topological base* relations, present in both structures.

The last phase of the project deals with the integration of both approaches. This must be done in such a manner that the best of both is kept without resulting in a too complex data model. One of the considered alternatives is:

```
/* FEATURE CLASSIFICATION RELATIONS (here shown without hierarchy): */
point_feat (pclass = char[])
line_feat  (lclass = char[])
area_feat  (aclass = char[])
MapLayer   (name = char[])
/* METRIC RELATIONS: */
Points     (pnt = point)
Polylines  (pln = path)
Polygons   (pgn = oid[], mbr = box)
/* TOPOLOGICAL RELATIONS: */
NullCell   (pntoid = oid, layer = oid, pclass = oid)
OneCell    (plnoid = oid[], layer = oid, lclass = oid
            fromnode = oid, tonode = oid, leftpgn = oid, rightpgn = oid)
TwoCell    (pgnoid = oid[], layer = oid, aclass = oid)
```

The topological relations contain references to the metric relations (through pntoid, plnoid, pgnoid), the map layer and the feature classifications (through pclass, lclass, and aclass).

6 Conclusion

The fds stores all coordinates, including those of curvature points, in the `node` relation. For very large databases this approach leads to reduction of performance. This problem can be solved by using a polyline. This is in fact a redefinition of an arc. The fourth convention of single-valued vector maps (Section 3.1) will have to be adjusted, i.e. arcs can have a shape. In the TIGER relations references to the polylines that form a polygon are explicitly stored in the `Polygons` relation (Section 5). This is redundant data, because the polygons can be constructed after the proper polylines have been selected in the database using the topological information. Of course, constructing instead of storing polygons will reduce the interaction speed.

A system identifier (`oid`) of an instance in one relation can be used as data item in another relation. When the relation with the original `oid`s is being removed from the database, then these `oid`s are no longer valid in the other relation. Integrity rules must be defined to guarantee a consistent database.

The resulting data model in this paper (Section 5) combines the strength of the fds which allows interaction with the GIS at user-level, with a representation of multiple layers which is based on a clear distinction of metric and topology.

Postgres features that are usually not available in commercial databases, but that are used in these implementations are: *multi-dimensional data types and operators* used to model metric primitives, *spatial index structures* enabling their efficient manipulation, *rules* for guarding the integrity, *object identity* instead of relational user defined keys, *class inheritance* for modeling the classification hierarchy, and *postquel functions* for referencing a sequence of queries by one name. The latter is a powerful tool for implementing a query language which offers complex high level topological queries as primitives.

The result is a solid basis for building an advanced GIS. In a subsequent project this topological data model and query language will be incorporated in the research-oriented GIS, called GEO [23].

Acknowledgments

The basis of this report was the discussion at FEL-TNO among Pamela Mercera (FEL-TNO), John Stuiver (WAU), and the authors during the week of 19–23 August 1991. Many valuable comments and suggestions on an preliminary version of this paper were made by the following persons: Martien Molenaar, Rene van der Schans, and Tom Vijlbrief. We would also like to thank the Postgres Research Group (University of California at Berkeley) for making their system available.

References

[1] Gerard Boudriault. Topology in the TIGER file. In *Auto-Carto 8*, pages 258–269, 1987.

[2] T. Bouloucos, O. Kufoniyi, and M. Molenaar. A relational data structure for single valued vector maps. In *Comm. III ISPRS, Wuhan*, 1990.

[3] DGIWG. DIGEST – digital geographic information – exchange standards – edition 1.0. Technical report, Defence Mapping Agency, U. S. A., Digital Geographic Information Working Group, June 1991.

[4] ESRI Environmental Systems Research Institute, Redlands, California. *ARC/INFO Volume 1*, 2nd edition, January 1988. Users Guide.

[5] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. *ACM SIGMOD*, 13:47–57, 1984.

[6] Cam-Chuong Huynh, Wolfgang Reinhardt, and Hongguang Yang. Neuer SICAD-Baustein zur Vektorverschneidung. Realisierungskonzept, not public, September 1990.

[7] Christine Kinnear. The TIGER structure. In *Auto-Carto 8*, pages 249–257, 1987.

[8] Pamela A. Mercera. A geometric extension to Postgres. Technical Report FEL-91-S304, FEL-TNO Divisie 2, September 1991.

[9] Martien Molenaar. Single valued vector maps – A concept in Geographic Information Systems. *Geo-Informationssysteme*, 2(1):18–26, 1989.

[10] Martien Molenaar. Object-hierarchies. Why is data standardisation so difficult. In Kockelhoren et al., editor, *Kadaster in Perspectief*, pages 73–86. Dienst van het Kadaster en de Openbare Registers, Apeldoorn, 1991.

[11] Martien Molenaar. Terrain objects, data structures and query spaces. In Matthaüs Schilcher, editor, *Geo-Informatik, München*, pages 53–70, 1991.

[12] Scott Morehouse. ARC/INFO: A Geo-Relational Model for Spatial Information. In *Auto-Carto 7, Washington*, pages 388–397, 1985.

[13] Scott Morehouse. The architecture of ARC/INFO. In *Auto-Carto 9, Baltimore*, pages 266–277, April 1989.

[14] Thomas K. Peucker and Nicholas Chrisman. Cartographic data structures. *The American Cartographer*, 2(1):55–69, 1975.

[15] Postgres Research Group. The Postgres reference manual, version 3.1. Technical Report Memorandum, Electronics Research Laboratory, College of Engineering, December 1991.

[16] Jon Rhein and Greg Kemnitz (editors). The Postgres user manual. Technical Report Memorandum, Electronics Research Laboratory, College of Engineering, December 1991.

[17] Siemens Nixdorf, Munich, Germany. *SICAD-KRT1 (BS2000) SICAD-Baustein für die Kartenbearbeitung Stufe 1*, March 1990. Benutzerhandbuch.

[18] Michael Stonebraker and Lawrence A. Rowe. The design of Postgres. *ACM SIGMOD*, 15(2):340–355, 1986.

[19] Unisys Corporation. Argis 4ge – database administration – student guide, August 1990.

[20] U.S. Bureau of the Census. The DIME geocoding system. Technical Report 4, Census Use Study, U.S. Deparment of Commerce, Bureau of the Census, 1970.

[21] Peter van Oosterom. *Reactive Data Structures for Geographic Information Systems*. PhD thesis, Department of Computer Science, Leiden University, December 1990.

[22] Peter van Oosterom, Marcel van Hekken, and Marco Woestenburg. A geographic extension to the relational data model. In *Geo '89 Symposium, The Hague*, pages 319–333, October 1989.

[23] Peter van Oosterom and Tom Vijlbrief. Building a GIS on top of the open DBMS "Postgres". In *Proceedings EGIS'91, Second European Conference on Geographical Information Systems*, pages 775–787, April 1991.

[24] J.W. van Roessel. Design of a spatial data structure using the relational normal forms. *International Journal of Geographical Information Systems*, 1(1):33–50, 1987.

[25] Hongguang Yang. Vektorverschneidung von SICAD-daten für die Umwelt. Studie, not public, June 1990.

Appendix A: Data in fds Relations

There is no significant difference here between the pure relational solution and the Postgres approach with the `oids`. The data and queries will be stated in the pure relation form. For purpose of readability the id's of the features (point, line, and area) and nodes are prefixed with a single letter indicating the type:

```
area_feat:          line_feat:        point_feat:
aid  aclass         lid  lclass       pid  pclass  paid  node_nr
-----------------   ---------------   --------------------------
a1   Arable land    l1   Road_1       p1   Mill    a1    n4
a2   Meadow         l2   Road_2
a3   Outside        l3   Border

arc:
from_node to_node   left_aid  right_aid lid
--------------------------------------------
n1          n2      a1           a3     l3
n2          n3      a2           a3     l3
n1          n7      a3           a1     l3
n2          n5      a1           a2     l1
n3          n6      a2           a3     l3
n5          n6      a2           a2     l1
n5          n8      a1           a2     l2
n6          n9      a2           a3     l3
n7          n8      a3           a1     l3
n8          n9      a3           a1     l3

node:
node_nr       x y
-----------------
n1            1 1
n2            3 1
n3           10 1
n4            5 4
n5            6 4
n6           10 4
n7            1 8
n8            6 8
n9           10 8
```

Appendix B: Data in Map Layers Relations

The `oid`'s of the tuples are system generated values. However, in this appendix the `oid`'s of `Points`, `Polylines`, and `Polygons` tuples are simulated. The `oid`'s in the other tables are represented by "readable" symbols similar to the fds-example; in reality these are also system generated values.

```
Points:                 Polylines:
oid        point        oid      pln
----------------        ------------------------------------
101        (3,1)        201      {(3,1),(1,1),(1,8),(6,8)}
102        (6,8)        202      {(3,1),(6,4),(6,8)}
103        (1,8)        203      {(3,1),(10,1),(10,8),(6,8)}
104        (5,1)        204      {(5,1),(1,1),(1,8)}
                        205      {(5,1),(1,8)}
                        206      {(5,1),(10,1),(10,8),(1,8)}


Polygons:
oid    pgn          mbr
-------------------------------
301    {201,202}   (1,1),(6,8)
302    {202,203}   (3,1),(10,8)
303    {204,205}   (1,1),(5,8)
304    {205,206}   (1,1),(10,8)

MapLayer:               NullCell:
oid    name             oid     pntoid    layer
---------------         -----------------------
t1      landuse         n1      101       t1
t2     soiltype         n2      102       t1
                        n3      103       t2
                        n4      104       t2

OneCell:
oid    plnoid  layer   fromnode  tonode  leftpgn  rightpgn
-----------------------------------------------------------------
l1     201     t1      n1        n2      OUT      a1
l2     202     t1      n1        n2      a1       a2
l3     203     t1      n1        n2      a2       OUT
l4     204     t2      n3        n4      OUT      a3
l5     205     t2      n3        n4      a3       a4
l6     206     t2      n3        n4      a4       OUT
```

TwoCell:

oid	pgnoid	layer	thematic
a1	301	t1	industry
a2	302	t1	agriculture
a3	303	t2	sand
a4	304	t2	clay

ADDED TO CONTENTS AFTER MAP OVERLAY:

Points:

oid	point
105	(4.3,2.3)

Polylines:

oid	pln
207	{(3,1),(1,1),(1,8)}
208	{(4.3,2.3),(1,8)}
209	{(4.3,2.3),(6,4),(6,8)}
210	{(5,1),(10,1),(10,8),(6,8)}
211	{(3,1),(4.3,2.3)}
212	{(5,1),(4.3,2.3)}
213	{(1,8),(6,8)}
214	{(3,1),(5,1)}

Polygons:

oid	pgn	mbr
301	{201,202}	(1,1),(6,8)
302	{202,203}	(3,1),(10,8)
303	{204,205}	(1,1),(5,8)
304	{205,206}	(1,1),(10,8)

MapLayer:

oid	name
t3	landuse+soiltype

NullCell:

oid	pntoid	layer
n5	105	t3
n6	101	t3
n7	104	t3
n8	103	t3
n9	102	t3

OneCell:

oid	plnoid	layer	fromnode	tonode	leftpgn	rightpgn
l7	207	t3	n6	n8	OUT	a13

l8	208	t3	n5	n8	a13	a14
l9	209	t3	n5	n9	a14	a24
l10	210	t3	n7	n9	a24	OUT
l11	211	t3	n6	n5	a13	a23
l12	212	t3	n7	n5	a23	a24
l13	213	t3	n8	n2	OUT	a14
l14	214	t3	n6	n7	a23	OUT

TwoCell:

oid	pgnoid	layer	thematic
a13	305	t3	industry+sand
a14	306	t3	industry+clay
a23	307	t3	agric+sand
a24	308	t3	agric+clay

Appendix C: Other Topology Implementations

In this appendix four other topology implementations will be described: ARGIS 4GE from Unisys, DIGEST military exchange standard, ARC/INFO from ESRI, and SICAD from Siemens Nixdorf. All four implementations are described in a relational manner.

ARGIS 4GE

Note that this subsection is based on [19] and this information is proprietary to Unisys and may not be disclosed without written permission of the Law department, Unisys Corporation.

ARGIS 4GE has a *dual architecture* [23]. The metric data is stored in a separate storage system and the other data (topological and thematic) is stored in a relational DBMS. The links between the two storage systems are formed by unique `FEAT_NUM`s. Features are organized in map layers, which may contain topology in a so called "network." There are two types of networks: linear and polygon. The coordinate (metric) component stores, besides coordinates, also some additional information: feature type, layer, network, min/max extents, coordinate count, etc. A quadtree index is provided for efficient access. The most important relations ("system tables") in the relational DBMS are:

```
master  (FEAT_NUM, TYPE, FEAT_CODE, LAYER, NETWORK, LENGTH)
layer   (LAYER, NAME, DESCRIPTION)
network (NETWORK, LAYER, NAME, TYPE, DESCRIPTION)
node    (FEAT_NUM, ANGLE, LINE, COORD_DIR, FLOW_DIR)
polygon (FEAT_NUM, AREA, TOTAL_AREA)
edge    (FEAT_NUM, LEFT_POLY, LEFT_FLAG, RIGHT_POLY, RIGHT_FLAG)
```

`TYPE` in the `master` relation can be 1, 2, 3, or 4 (codes for point, line, polygon, and nodes respectively). The `TYPE` in the network can be `p` or `l` (polygon or linear). Other tables are: `feat_code`, `network_error`, `network_overlay`, `snip_points`, and `attention_point`.

DIGEST

The data model described in this section is from DIGEST: DIgital Geographic information Exchange STandards [3]. DIGEST supports four types of geographic data models: Topological vector, Spaghetti vector, Matrix and Raster. This subsection concentrates on the topological vector model. This model is quite similar to the fds, as it recognizes above the level of primitives (face, edge, node). One simple area, line, or point feature may consist of several primitives of the same type: face, edge, or node respectively. Complex features may consist of several different simple features. Figure 7 shows the DIGEST topological vector model.

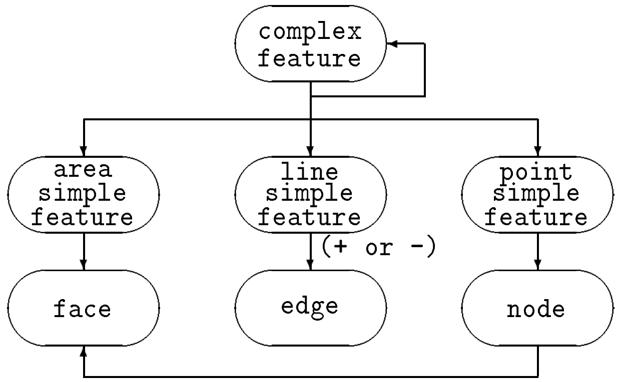

Figure 7: The DIGEST topological vector model

Though the original DIGEST documentation [3] uses normalized relations to describe the standard, here they are given as in an (more compact) un-normalized form. Repetition is indicated with three dots (...).

```
FEATURE   (ID, MB, GRP, relations..., attr...)
FACE      (ID, MB, GRP, attr...)
EDGE      (ID, MB, GRP, coord..., START_NODE, END_NODE,
           LEFT_FACE, RIGHT_FACE, attr...)
NODE      (ID, coord, IN_FACE, attr...)
```

`ID` stands for an identification, `MB` for minimum bounds (two points), `GRP` is the Geographic Reference Point, `attr...` is used for one or more thematic attributes (of different types: int, float, text,...), and `coord...` represents one or more coordinates.

ARC/INFO

Like ARGIS 4GE, ARC/INFO is a geographic information system with a dual architecture. The geometric data are represented by a topological data model (the ARC part). The thematic data are represented by a tabular or relational data model (the INFO part) [12]. ARC/INFO contains four types of primary geometric elements to represent line features, point features and area features on maps: arcs, nodes, label points and polygons. Arcs can be part of line features, the borders of polygons, or both. An arc is stored as an ordered series of x,y-coordinates (a chain) which define its metric properties: location and shape. Nodes are stored as the endpoints of an arc. Label points represent point features or can be applied as identification points for polygons (not both). A label point is described by one x,y-coordinate, in case of a polygon a label point may be an arbitrary point within the polygon. Polygons represent area features. A polygon is described by the series of arcs which form its border, including islands. These geometric elements are assigned a unique (but not fixed) sequence number by the system, a so called *internal number* which will here be denoted by `#`. Internal numbers are used for two purposes: as topology references and

to relate the geometric data to the thematic data.
The thematic data are stored in feature attribute tables: point (PAT), arc (AAT) and polygon attribute tables (PAT) respectively. Beside the above-mentioned internal number, a user can assign an integer value to each geometric element (User-ID). User-IDs are mainly used to relate records in different tables.
ARC/INFO has the ability to create and update topology. The connectivity of arcs is represented by the nodes. Adjacency between polygons is represented by storing the left-polygon and right-polygon internal sequence numbers for each arc [4].
The primary relations in ARC/INFO are [4, 13]:

```
ARC  (arc#, x,y..., x,y, User-ID)
AAT  (f_node#, t_node#, l_poly#, r_poly#, length, arc#, User-ID, attr...)
LAB  (label#, poly#, x,y, User-ID)
PAL  (poly#, arc#..., arc#, User-ID)
PAT  (area, perimeter, label#/poly#, User-ID, attr...)
```

`LAB` and `PAL` are short for (the geometric part of) label points and polygons. Users have no access to the relations `ARC`, `LAB` and `PAL`. In the relations `AAT` and `PAT` users can manipulate at most the User-IDs and the attributes. Note that the internal numbers and coordinates of nodes are only stored in two separate relations `ARC` and `AAT`.

SICAD

The information in this subsection is retrieved from [25, 6]. These documents are not public.

The geographic information system SICAD (SIemens Computer Aided Design) has also a dual architecture: the graphical (geometric) elements are stored in a network database and the thematic attributes are stored in a relational database. The two subsystems are connected through common identifiers (ID's). The thematic attributes may partly be stored within the geometric elements.
Recently arc-polygon topology has been incorporated in the SICAD data structure as means for efficiently performing map overlay (of mainly polygonal maps) and to extend the set of topological queries. To facilitate map overlay map layers are converted to so called topological layers. Topological layers consist of simple, connected geometric structures, i.e. no isolated points, lines and areas. The basic geometric elements are points (PG), polylines (LY), and areas (FL). At feature level area features (AR) exist. Area features can consist of one or more simple areas.
The key relations in the SICAD topological data structure are:

```
LY   (..., code, ID, l_fl, r_fl, b_ly, e_ly, ..., attr.)
PG   (x,y,z, ..., code, ID, attr.)
FL   (x,y, perimeter, ..., code, ID, attr.)
AR   (x,y, area, ..., code, ID, attr.)
```

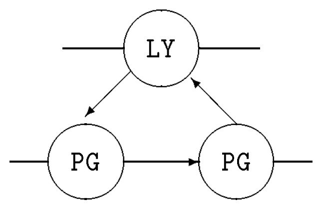

Figure 8: Line LY is master of both details PG

The item *code* in the four relations contains marks for metric and topological properties. Besides connecting the separately stored geometric and thematic data, the ID's are also used as topology references in the relation LY. In the network part of the system a line and its begin- and endpoint are stored according to *master-detail* relationships (Figure 8). The order of the points in the ringstructure gives direction to the line. In the topological layer all (directed) lines are sorted by increasing angle (counter-clock wise). In the relation LY each line contains therefore references to its "begin" and "end" line, i.e. those two lines, connected to either begin- or endpoint of the line considered, which have the smallest angle (counter-clock wise) with that line. The relation LY also contains references to the left and right simple areas, but not to the area features.
Note that relation PG contains an item z for a third dimension (height).

In addition the SICAD data structure element OB (object) can be used to represent different terrain features. An OB is a logic aggregate of arbitrary geometric elements. OB's may be combined to form super-OB's [17].