Topology in spatial DBMSs

Wilko Quak, Jantien Stoter and Theo Tijssen
Department of Geodesy, Faculty of Civil Engineering and Geosciences
Delft University of Technology, The Netherlands
Telephone: +31 (0)15-2783756
{c.w.quak|j.e.stoter|t.p.m.tijssen }@citg.tudelft.nl

Abstract

Nowadays, GISs are changing to an integrated architecture in which spatial and attribute data are maintained in one DataBase Management System. Mainstream DBMSs have implemented spatial data types, spatial functions and spatial indexes. The spatial data types are implemented using a geometrical model, consequently the relationships between neighbouring spatial objects cannot be stored and can only be queried with a geometrical query.
In our research we implemented topology in the DBMS in two ways. We also studied the possibilities of Laser-Scan's Radius Topology: recently released software in which a topological model has been implemented on top of Oracle Spatial. In this paper we describe our solution for topology in DBMSs and our experiments with Radius Topology. We end with a discussion on the advantages and disadvantages of the three approaches.

Section 1 Introduction

Spatial data, as used in GISs, is mostly part of a complete work- and information-process. In many organisations there is a growing need for a central DBMS: a system in which spatial data and attribute data are maintained in one integrated environment. Many DBMSs are capable of maintaining spatial data in 2D, since mainstream DBMSs (Oracle (Oracle, 2001), IBM DB2 (IBM, 2000), Informix (Informix, 2000), Ingres (Ingres, 1994)) have implemented spatial data types in 2D more or less similar to the OpenGIS Consortium (OGC, 1998) Simple Features Specification for SQL (OGC, 1999). The implementations consist of an SQL extension that supports storage, retrieval, query and update of simple spatial features (points, lines and polygons).
According to the OpenGIS specifications, the spatial object is represented by two structures, i.e. *geometrical (i.e. simple feature specifications)* and *topological (i.e. complex feature specifications)*. While the geometric structure provides direct access to the coordinates of individual objects, the topological structure encapsulates information about spatial relationships. Most current implementations are based on the geometrical model. With the implementations of the geometrical model it is possible to store and query spatial features in a DBMS, but the relationships between neighbouring spatial objects cannot be stored and can only be determined with a geometrical query. The geometrical model causes redundancy: shared edges and shared nodes are stored twice. Topology avoids redundancy and the performance of some spatial queries improves. Recently, Laser-Scan (Laserscan, 2003) has implemented a topological model on top of the geometrical model in Oracle Spatial 9i.

In our Department we carried out research on the possibilities of storing and using topology in a DBMS. For experiments we use Oracle Spatial 9i. Although Oracle Spatial 9i is not OGC compliant on all levels, our experiments show generic aspects and solutions for supporting topology in a DBMS. The research focuses on the topological structure of a planar partition.

Section 1.1 Dataset

For our experiments we use a data set of cadastral parcels, provided by the Netherlands' Kadaster. This data set is modelled topologically, i.e. the geometry of the parcels is not stored, but can be inferred from the cadastral boundaries that are stored geometrically. The most important tables are 'boundary' (cadastral boundaries) and 'parcel' (parcel identifiers). There is no need for the geometric data type 'polygon', because the area features (parcels) are stored topologically in the 'parcel' and 'boundary' table. The edges in the boundary table contain references to other edges according to the winged edge structure (Baumgart, 1975), which are used to form the complete boundary chains (parcels). The edges also contain a reference to the left and right parcel (Oosterom and Lemmen, 2001).

A parcel has exactly one reference to one of the surrounding boundaries and one reference to a boundary of each enclave. The structure of the topological references and the relationship between parcels and boundaries is visualised in Figure 1.

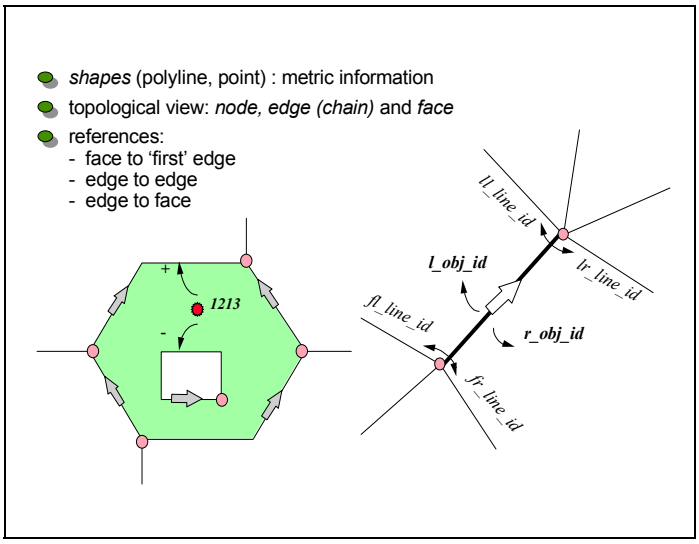

Figure 1: Topological model in the spatial DBMS of the Netherlands' Kadaster

Section 1.2 Topology in the DBMS

A disadvantage of storing the parcels in a topological model in the DBMS is that the DBMS is not aware of the geometry of the parcels. Because there is no geometry attribute in the parcel table, it is for example not possible to calculate the area of a parcel. By extending the DBMS with a function that materializes the geometry from the boundaries it is possible to store the data topologically whilst it is still possible to perform operations on the geometry.
We implemented this by creating a function 'return_polygon' which realizes the geometry of a polygon. This function can be used in an sql-statement, e.g. in a query to compute the area of a parcel:

```
select sdo_geom.sdo_area(return_polygon(object_id), 1) from parcel;
```

A function-based spatial index is created in order to optimise the performance. Since version 9i, Oracle offers function-based indexes, i.e. an index which is created on the return value of a function in addition to a normal index created directly in the value of an attribute. A function based spatial index facilitates queries that use local information (of type sdo_geometry) returned by a function. The spatial index is created based on the pre-computed values returned by the function. This is implemented in Oracle 9i in two steps. First, the USER_SDO_GEOM_METADATA view was updated to specify the function name.

```
insert into user_sdo_geom_metadata values(
  'PARCEL', 'return_polygon(object_id)', mdsys.sdo_dim_array (
  mdsys.sdo_dim_element('X', 82291, 84261, 0.0005),
  mdsys.sdo_dim_element('Y', 453039, 455632, 0.0005)), NULL);
```

Then the spatial index is created by specifying the function name and parameters. For example, creating an R-tree index, is done with the following SQL-statement:

```
create index parcel_idx on parcel(return_polygon(object_id))
  indextype is mdsys.spatial_index;
```

Without a function-based spatial index it would not have been possible to properly index the faces. During an overlap query (or any other search query using the spatial index), objects are filtered by means of this index. That is, using the pre-computed boxes which are stored in the R-tree. Then the return_polygon function is executed to obtain the complete geometry of filtered objects used in the exact overlap test.

Section 1.3 Overview of this paper

First a short introduction in the formal specifications of topology and geometry is supplied (section 2). The function used in our research is implemented in two ways. The first solution uses the information on the relationship between edges (winged edge structure). The second solution is based on the left-right information of edges, which also is stored. Section 3 describes our own implementation including a comparison with the formal specifications. In section 4 we describe how Laser-Scan implemented topology in the DBMS and compare this to our own experiments. We will end the paper with a discussion in which we will draw a conclusion on the state-of-the-art of topology in DBMSs based on our experiments. We also give recommendations on what still has to be developed to make full and generic support of topology in DBMSs possible.

Section 2 ISO, OGC and topology of planar partitions

Since this research focuses on the topological structure of a planar partition, the interesting feature to consider is 'face'. Face is the topological equivalent of polygon. This section describes how ISO and OpenGIS consortium define the 'face' and 'polygon' primitive.

ISO
The ISO standard 19107 'Geographic information — Spatial schema' defines geometrical primitives of which the code starts with 'GM' and related topological primitives, of which the code starts with 'TP'.
A TP_FaceBoundary consists of one or more TP_Rings. One of these rings is distinguished as being the exterior boundary. Each ring is oriented so that the face is on its left. A TP_Ring is used to represent a single component of a TP_FaceBoundary. It consists of a number of TP_DirectedEdges in a cycle (an object whose boundary is empty). The endNode of a TP_DirectedEdge is in the sequence the startNode of the next TP_DirectedEdge. TP_rings are oriented in TP_FaceBoundary objects in such a way that the face is on its left in any geometric realization.
According to these ISO standards a face is defined by edges and those edges are anti-clockwise oriented. Every edge has a reference to the next and a reference to the previous edge. An edge is described by the start- and end-nodes. The ISO standard defines several constraints on the geometrical equivalent of face, which is the polygon. From the specifications it can be derived that the outer boundary of a polygon has to be simple (i.e. non-selfintersecting). Whether or not the outer boundaries of a polygon are allowed to intersect with the interior boundaries is not specified.

OGC specifications for SQL
The ISO definition of the topology for planar partition is at the abstract level. The OpenGIS Consortium adopted these abstract specifications and translated these to the implementation level in the OpenGIS Simple Feature Specification (SFS) for SQL. Until now, the specifications for SQL are based on the geometrical model. Data types are geometrically described. Topological relationships can be derived by spatial operations. These topological operations do not give the dimensionality of the relationship as a result. For example the query 'Find all adjacent parcels to a given parcel' using the touch relationship will return all parcels that touch the query parcel, regardless whether the touch situation is a point or line. To restrict the result dataset to only parcels that touch in a line, the query should be extended with the condition that lines of two parcels should (partially) overlap. If the query to find adjacent parcels to a query parcel is performed on topologically structured data according to the winged edge model, only the parcels that touch in an edge will be given as a result.
Since the Specifications for SQL do not define topology, we will have a look at the geometrical model of a polygon. A polygon is defined as a simple surface that is planar. In the specifications, the polygon is defined precisely. The main characteristic relevant for this paper is that no point on the same ring can be equal to another point defining that ring and that it is allowed for an inner-ring to touch the outer boundary.

Section 3 Self implemented software

Section 3.1 Realising geometry based on winged-edge structure

The function ‘return_polygon’ based on the winged-edge structure, starts with the table with ‘parcels’. The function creates a polygon geometry, which is valid according to the Oracle rules: the co-ordinates of the outer boundary are listed counter clockwise and the co-ordinates of the enclaves are listed clockwise. In the dataset the winged edge structure is defined in both directions, since every boundary contains a reference to its four connecting boundaries.

The relevant attributes in ‘parcel’ table used in the construction of polygons are:

- object_id: the unique identifier of parcels
- line_id1: reference to one of the surrounding boundaries (stored in the boundary table)
- line_id2: reference to one of the boundaries of the first enclave (stored in the boundary table)

If there is more than one enclave the ‘parcelover’ table is used. The relevant attributes in this table are:

- object_id: the unique identifier of parcels
- line_id1: reference to one of the boundaries of the second enclave
- line_id2: reference to one of the boundaries of the third enclave
- …….
- line_id10: reference to one of the boundaries of the eleventh enclave

Also these line_id’s refer to a line in the boundary table. If a parcel has more than eleven enclaves, the parcelover table has more than one entry for that object_id.

The relevant attributes in the 'boundary' table are:

- object_id: unique identifier of boundaries
- geo_polyline: geometry of the line
- fl_line_id: reference to the first line on the left, seen from the middle point of the line
- ll_line_id: reference to last line on the left, seen from the middle point of the line
- fr_line_id: reference to the first line on the right, seen from the middle point of the line
- lr_line_id: reference to last line on the right, seen from the middle point of the line
- l_parcel: parcel that is located at the left- hand side from the directed boundary
- r_parcel: parcel that is located at the right-hand side from the directed boundary

Note that these references are different from figure 1. In figure 1 the references at the start of the edge or ‘left’ or ‘right’ are seen from the starting point from the edge. In the data set these references are ‘left’ or ‘right’ seen from the middle point of the edge and therefore they are reversed. How the function works, will be illustrated with an example in which the polygon of parcel 603 is realised (see figure 2). The attributes of line_id1 and line_id2 are:

```
select object_id, parcel, line_id1, line_id2 from parcel where parcel=603;

 OBJECT_ID PARCEL   LINE_ID1   LINE_ID2
---------- ------ ---------- ----------
 310148953    603  310439663          0
```

Figure 2: Parcel 603 and 973 are used in the examples

The parcel has one reference to its outer boundary (line_id1) and no enclaves (line_id2=0). The polygon of the parcel can now be constructed by starting with the first boundary, with object_id=310439663. The boundary table is queried to look for the co-ordinates of this boundary and to look for the boundaries that are connected in counter clockwise direction. The boundaries also contain information about the parcels that are left and right located from the specific boundary. First we select all boundaries together with the relevant attributes, which have parcel 603 on their right-hand or left–hand:

```
select object_id, fl_line_id, fr_line_id, ll_line_id, lr_line_id, l_parcel, r_parcel from
  boundary where l_parcel=603 or r_parcel=603;

 OBJECT_ID FL_LINE_ID FR_LINE_ID LL_LINE_ID LR_LINE_ID L_PAR R_PAR
---------- ---------- ---------- ---------- ---------- ----- -----
 310547374  310419672 -310419673  310594168  310439663   973   603
 310419672 -310419673  310547374  310518755  310419671   603   960
 310518755  310419671 -310419672 -310439663  310439732   603   605
 310439663 -310547374  310594168  310439732 -310518755   960   603
```

The polygon construction starts with boundary with object_id 310439663. The data set is structured in such a way that the first reference is positive when the boundary is directed clockwise. Since the first reference in the example is positive, the boundary is directed clockwise. The boundary connected to it in anti-clockwise direction is the boundary that is connected 'first left' to it: 310547374.
This is a negative directed boundary (anticlockwise), now we have to take the 'first left' connected boundary: 310419672. If we continue with this till the first boundary-id is met again, the complete list looks as follows: 310439663, –310547374, 310419672, 310518755. For a positive directed boundary (anti clockwise directed), the 'last left' boundary is taken and for a negative directed (clockwise directed) boundary, the 'first left' connected boundary is taken.
The last boundary connects to the first boundary again. The polygon can now be constructed by connected all the linestrings of these boundaries. The minus means that the specific linestring needs to be reversed, because the boundary is directed clockwise. The polygon geometry is realised in such a way that the co-ordinates at connection pints are stored only once, and that polygons are closed (first and last point is repeated).
The collected geometry information is inserted in the spatial data type of Oracle to create the polygon geometry of the parcel as a spatial data type:

```
select return_polygon(object_id) shape from rw_parcel where parcel=603;

SHAPE(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z), SDO_ELEM_INFO, SDO_ORDINATES)
--------------------------------------------------------------------------------
SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1), SDO_ORDINATE_ARR
AY(81830492, 449867128, 81814766, 449894053, 81753306, 449858028, 81768982, 4498
31090, 81769341, 449830603, 81769786, 449830215, 81770209, 449829933, 81770781,
```

```
449829736, 81771417, 449829623, 81771962, 449829639, 81772573, 449829777, 817730
92, 449830011, 81829420, 449863026, 81829963, 449863446, 81830285, 449863849, 81
830562, 449864296, 81830783, 449864833, 81830897, 449865434, 81830863, 449866024
, 81830749, 449866553, 81830492, 449867128))
```

According to Oracle, the generated geometry is a valid geometry:

```
select sdo_geom.validate_geometry(return_polygon(object_id),0.01) from rw_parcel
  where parcel=603;

SDO_GEOM.VALIDATE_GEOMETRY(RETURN_POLYGON(OBJECT_ID),0.01)
------------------------------------------------------------
TRUE
```

Now we will look at a polygon with enclaves: parcel 973 (see figure 2). As can be seen from line_id2, this parcel has at least one enclave, starting with boundary with object_id = 310376490 (line_id2):

```
select object_id, parcel, line_id1, line_id2 from rw_parcel where parcel=973;

 OBJECT_ID    PARCEL      LINE_ID1     LINE_ID2
---------- ----- ---------- ----------
 310152502    973 -310419676  310376490
```

Again, we first select all boundaries that have parcel 973 on the left or right hand side:

```
select object_id, fl_line_id, fr_line_id, ll_line_id, lr_line_id, l_parcel, r_parcel from
  rw_boundary where l_parcel=973 or r_parcel=973;

 OBJECT_ID FL_LINE_ID FR_LINE_ID LL_LINE_ID LR_LINE_ID L_PAR R_PAR
---------- ---------- ---------- ---------- ---------- ----- -----
 310309774  310309775 -310276856  310276863  310276862   956   973
 310309775 -310276856  310309774  310309773 -310354600   973   578
 310309773 -310354600 -310309775  310419688  310419787   973   961
 310419688  310419787 -310309773  310419676 -310419677   973   579
 310376488 -310376490  310376489 -310376489  310376490   971   973
 310419676 -310419677 -310419688  310419673  310419675   973   542
 310376490 -310376488 -310376489  310376489  310376488   970   973
 310379237 -310379237 -310379237  310379237  310379237   973   972
.....
 310323936 -310547377 -310758683  310484543 -310205689   973   706
 310676405 -310597737  310597742  310597741  310666423   606   973
 310597737 -310323958  310323957  310597742  310676405   593   973
 310205706  310594167 -310594168  310758683 -310547378   973   602
 310758683 -310547378 -310205706  310323936 -310547377   973   705
 310323958 -310323943  310323915  310323957  310597737   959   973
 310758688 -310254553 -310379527 -310758687  310340471   609   973
 310340471 -310758688 -310758687 -310379484  310758690   586   973
 310758690 -310340471 -310379484 -310240245  310240247   608   973
```

The realisation of the outer boundary of the polygon is performed in the same way as in the first example. In this example the parcel contains one or more enclaves (line_id2<>0), and therefore the enclaves need to be constructed (in clockwise direction). The first enclave starts with boundary with object_id 310376490 (line_id2). Since the boundaries of enclaves are defined in opposite direction than the outer boundaries in the winged edge structure (anticlockwise order), we can follow the same procedure as in the case of the outer boundary: find the list of all connecting arcs (this time in clockwise order), to realise the geometry of the enclave. For a positive directed boundary (anticlockwise directed), we take the 'last left' boundary and for a negative directed (clockwise directed) boundary, we take the 'first left' connected boundary.
In this example, two boundaries form the first enclave: 310376490 directs to 310376488, since this is a negative reference, the connected boundary of the enclave is the 'first left' boundary, which is 310376490 again. The geometry of enclaves is constructed in the same way as the geometry of outer boundaries: linestrings are connected, double co-ordinates are removed, linestrings in counter clockwise direction are reversed and the polygon is closed. To see if this parcel has more than one enclave the 'parcelover' table is checked:

```
select * from parcelover where object_id in (select object_id from rw_parcel
  where parcel=973);

  LINE_ID1   LINE_ID2   LINE_ID3   LINE_ID4   LINE_ID5  LINE_ID6   LINE_ID7   LINE_ID8
---------- ---------- ---------- ---------- ---------- --------- ---------- ----------
  LINE_ID9  LINE_ID10
---------- ----------
-310379237 -310205718          0          0          0         0          0          0
         0          0
```

The result is two more enclaves. The enclaves are generated in the same way as the first one. Again, the collected geometry information of enclaves together with the geometry of the outer boundary is inserted in the spatial data type of Oracle to create the polygon geometry of the parcel as a spatial data type:

```
select return_polygon(object_id), sdo_geom.validate_geometry(return_polygon(object_id), 0.01)
  from parcel where parcel=973;

RETURN_POLYGON(OBJECT_ID)(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z), SDO_ELEM_INFO
--------------------------------------------------------------------------------
SDO_GEOM.VALIDATE_GEOMETRY(RETURN_POLYGON(OBJECT_ID),0.01)
--------------------------------------------------------------------------------
SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1, 209, 2003, 1, 227
, 2003, 1, 241, 2003, 1), SDO_ORDINATE_ARRAY(81719739, 449884875, 81740070, 4498
50270, 81753306, 449858028, 81814766, 449894053, 81831869, 449904078, 81819109,
449925889, 81818800, 449926521, 81818732, 449927103, 81818697, 449927764, 818190
42, 449928865, 81819360, 449929310, 81819713, 449929680, 81820173, 449929988, 81
……..
21013, 449962385, 81733667, 449903365, 81787769, 449935141, 81805522, 449904858,
 81762913, 449879833, 81764144, 449877738, 81752648, 449871027, 81733667, 449903
365, 81851044, 449971293, 81807612, 449945704, 81778409, 449995014, 81804062, 45
0010168, 81823270, 449977885, 81841194, 449988382, 81851044, 449971293))

TRUE
```

Section 3.2 Realising geometry based on left-right information

An alternative version of the 'return_polygon' function uses the left-right information stored with every parcel boundary. Here only the boundaries that have the given parcel to the left or to the right are selected. By repeatedly joining boundaries that end in the same endpoint, we end up with the boundary of the complete parcel. Enclaves are realised in the same way.

The attributes in the 'boundary' table that are used by the algorithm are:

- geo_polyline: geometry of the line
- l_parcel: parcel that is located at the left- hand side from the directed boundary
- r_parcel: parcel that is located at the right-hand side from the directed boundary

The function is implemented in the Java programming language and is integrated in the database server. The function accesses the tabular data via an internal JDBC connection.

The first step is to retrieve all boundary lines that are part of the parcel:

```
select geo_polyline from boundary where l_parcel = 973 or r_parcel = 973;
```

This query results in a collection of LineStrings. What needs to be done now is to glue these LineStrings together in such a way that they form an ordered collection of rings. This is done using two data-structures:

- rings: In this variable we collect the LinearRings (LineStrings that form a loop) that are formed during the algorithm.
- graph: The graph structure contains all LineStrings that still need to be combined to form loops. The graph contains vertices and edges. The endpoints of the LineStrings form the vertices of the graph. The edges in the graph are formed by the LineStrings and runs between two nodes, being the startpoint of the LineString and the endpoint of the LineString.

The algorithm first fills the graph structure and then tries to move all LineStrings in the graph to the rings structure from which the result is constructed.

```
//
// 1. Initialization.
//
for (all LineStrings that belong to the parcel boundary)
{
   Insert the LineString into the graph.
}

//
// 2. Main Loop.
//
while (graph contains a node with two edges)
```

```
{
   Delete the node and the two edges from the graph.

   if (the two edges at the node are the same edge)
   {
      we have found a loop and add the edge to the rings.
   }
   else
   {
      glue the two LineStrings together to form one big LineString.
      Insert the new LineString into the graph.
   }
}

//
// Now the graph should be empty. If this is not the case, the input
// data was incorrect.
//

//
// 3. Construct Polygon from rings.
//
Find the ring with encloses the largest area. This is the boundary.
The rest of the rings are enclaves.
Construct a Polygon using the boundary an the enclaves and return.
```

Implementation details are not further described in this paper. For example the orientation of the rings (clockwise or counter-clockwise) does not follow from the algorithm and must be calculated afterwards.

Section 3.3 ISO/OGC and our own implementations

The implementations of planar partitions described above differ in the underlying geometrical model. In the winged-edge implementation a face can touch itself in the outer boundary in exactly one point and in the left-right implementation this is not possible. This difference can be illustrated by the polygon as shown in figure 3: a polygon that has an island that touches the boundary in exactly one point. The winged-edge algorithm will generate a polygon with one self-intersecting outer ring, while the left-right algorithm will return a polygon with an outer boundary and an island. As was described in section 2, a self-intersecting boundary is not allowed according to the OpenGIS Specifications for SQL.
Therefore, the winged-edge algorithm results in a non-valid geometry. Post-processing the invalid polygons is possible, but requires so much geometric and topological calculations that it is easier to use the left-right topology.

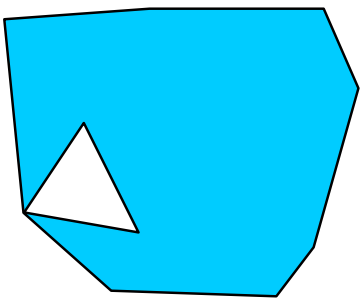

Figure 3: A polygon with a hole that touches the boundary

Section 4 Topology software

Compared to the self-implemented software, the implementation of topology in Radius Topology (Louwsma, 2003) is much more extensive, it is a 'complete' implementation of topology with support for linear networks and planar topology. All required topological references are stored explicitly: the winged edge representation (in the edge-to-edge table) makes up just a small part of the complete system (see figure 4). Topological primitives are stored in the NODE, EDGE and FACE tables while faces are only stored by references to edges. A number of reference tables are used to store various types of topological references. The TOPO table is the link between the features and the topological structures. Topology is organised in 'manifolds'. Associated with each manifold and with the system as a whole are some meta data and error tables. Before topologically structuring data in Radius Topology, the user can specify rules in order to control the way the structuring works (snap tolerances, which features/primitives are moved and which stay put while snapping, etc.).

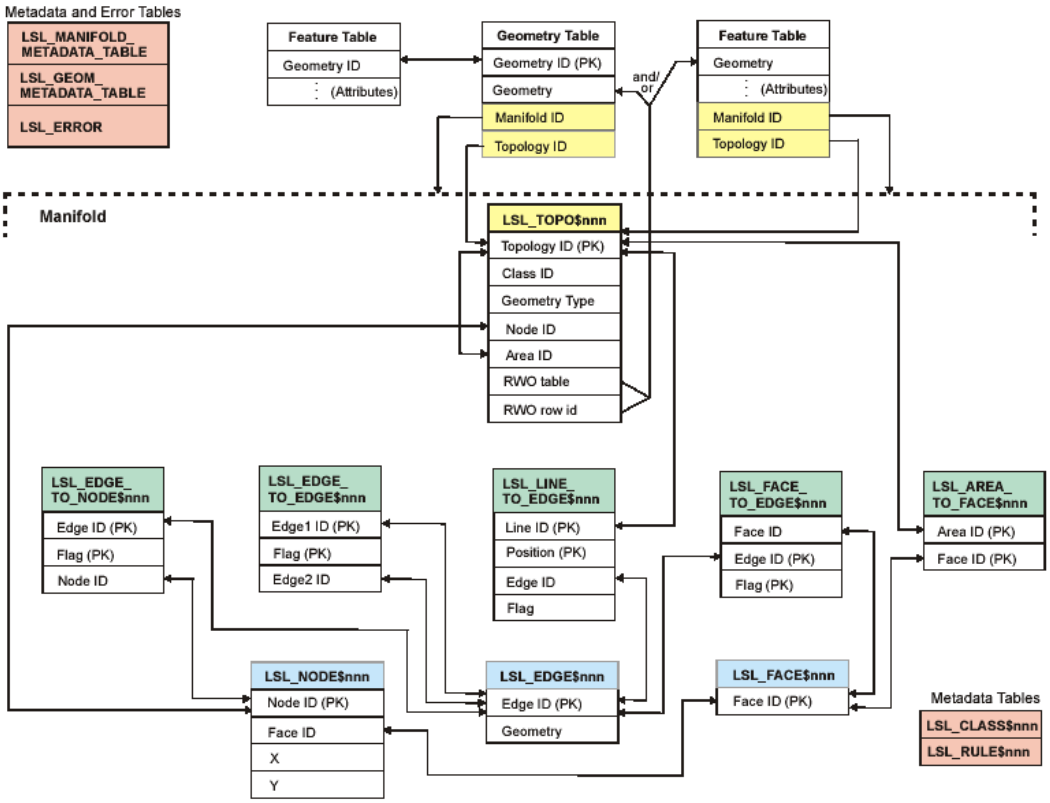

Figure 4: Radius Topology database tables (version 1.0).

To retrieve geometry from a topologically structured dataset Radius offers a 'get_geom' function that is equivalent to the 'return_polygon' function of our own implementation. But also support for topological querying (containment, adjacency, connectivity, overlap) is available by means of a TOPO_RELATE operator. This is the topological 'equivalent' of the geometrical SDO_RELATE operator of Oracle Spatial.

This functionality can be illustrated by an example query to find all features within a certain rectangular area (e.g. for display purposes). The query is performed both with a geometrical version of the query and a topological version of the query. In the geometrical version the geometry is explicitly stored (shape column). The get_geom. function in the topological version cannot retrieve the geometry directly, it must be constructed on-the-fly by means of topological references.

Geometry version:

```
select oid from areas
  where mdsys.sdo_relate (shape,
    mdsys.sdo_geometry(2003,NULL,NULL,
      mdsys.sdo_elem_info_array(1,1003,3),
```

```
      mdsys.sdo_ordinate_array(177450.82,495672.131,179811.475,499672.131)),
  'MASK=ANYINTERACT QUERYTYPE=WINDOW') = 'TRUE';
```

Topology version:

```
select oid from areas
  where mdsys.sdo_relate (lsl_topo_rwo.get_geom(mani_id,topo_id),
    mdsys.sdo_geometry(2003,NULL,NULL,
      mdsys.sdo_elem_info_array(1,1003,3),
      mdsys.sdo_ordinate_array(177450.82,495672.131,179811.475,499672.131)),
  'MASK=ANYINTERACT QUERYTYPE=WINDOW') = 'TRUE';
```

To learn more about Radius Topology several large scale cadastral (as described above) and topographical data sets were converted into topological primitives. While structuring the data some errors were detected in data sets that were supposed to be clean, an example of the way topology helps to improve data quality. The test data set presented here is a 1:50.000 data set containing area features of the Dutch Topographic Service. After structuring, the Oracle table of this data set, the table has the following columns:

OID	number(11)	-- unique object identifier
SHAPE	sdo_geometry	-- original feature geometry (closed polygons)
TOPO_ID	number(38)	-- topological ID, reference to topological structures
MANI_ID	number(38)	-- manifold ID

Radius Topology adds the last two columns to maintain the references to the topological structures, the SHAPE column contains the original, 'simple' geometry. The characteristics of the data set are:

```
Polygon geometry (with redundant 'internal' boundaries):
    41239  number of area features
   661596  total number of points
      732  maximum number of points in feature
    16.04  average number of points per feature
       31  maximum number of elements (polygon rings) in feature
     1.07  average number of elements per feature

Topology edges:
   106372  number of edges
   421084  total number of points
      290  maximum number of points in edge
     3.96  average number of points per edge
Topology nodes:
    68185  number of nodes (= points)
```

In the topology case about 17% less points are stored (by avoiding storing 'common' boundaries twice), but the disk space required is 30% bigger due to the increased number of topology primitives compared to the number of area features (and the way geometry is implemented in Oracle Spatial, small objects have relatively much overhead) :

```
     polygon geometry:  15.28 Mb  (R-tree index: 2.61 Mb)
     edge geometry:     17.94 Mb
     node geometry:      1.89 Mb
```

On top of that, the total storage requirements for topology are vastly increased by the references, ID's and associated indexes that are required by the Radius Topology model (see table 1 and figure 3). The storage requirements will be much more favourable towards topology in the case of smaller scale data and data with a relatively high number of intermediate points in the boundaries.

Table	Table in Mb	Indexes in Mb
node	2.34	7.41
edge	26.03	12.48
face	0.75	3.36
topo	1.69	1.26
edge_to_node	5.38	16.04
edge_to_edge	5.73	15.46
face_to_edge	5.69	16.37
line_to_edge	5.69	14.62
area_to_face	0.69	2.71
	53.39	97.73

53.99 89.71

Table 1: Disk space requirements of topology tables (including primitives) and associated indexes.

It can be expected that the performance of geometrical querying (queries that require the complete geometry of a feature, e.g. the polygons that represents parcels) on a data set structured with Radius Topology, will suffer to some extent from the large disk space requirements. How much depends on the type of query, most queries will only require access to a (small) part of all data stored. On the other hand, topological querying (queries that only require explicitly stored relationships) is much faster than in the situation where only simple geometry is available.

Section 5 Conclusions

The geometrical model described in the OpenGIS Specifications for SQL has been implemented in mainstream DBMSs. The next step is support for topology. In this paper we described two methods of supporting topology in a DBMS based on planar partition, which we implemented in Oracle Spatial 9i. Also our initial experiments with Radius Topology 1.0 (software that recently has been released and supports topology on top of the geometrical model in Oracle) were described.
It can be expected that spatial queries only relying on topological references are performing very well in the topological model compared to the geometrical model, e.g. to find all features that are adjacent to a certain feature. On the other hand, our experiments in Radius Topology 1.0 with large-scale spatial data, showed that storage requirements (and probably also performance) of the plain geometry approach are superior in many cases. For the time being, the most important advantage of topology maintained within a geo-DBMS is the improvement in data quality and consistency, the ease with which errors can be detected and corrected.
To make full and generic support of topology in DBMSs possible, storage and performance should be improved. Future research at our department will focus on comparing the three approaches describe in this paper on storage requirements and performance issues in order to be able to give recommendations on improvement of support of topology in DBMSs.

References

B.G. Baumgart (1975): A polyhedron representation for computer vision. In National Computer Conference, pages 589-596, 1975.

IBM (2000): IBM DB2 Spatial Extender User's Guide and Reference. special web release edition.

Informix (2000): Informix Spatial DataBlade Module User's Guide. Part no. 000-6868, URL: www.informix.com, 2002

Ingres (1994): INGRES/Object Management Extension User's Guide, Release 6.5 (1994). CA-OpenIngres.

Laser-Scan Radius Topology (2003): URL: http://www.radius.laser-scan.com/

Louwsma, J., T.Tijssen and P. van Oosterom (2003), A comparison between topologically structured and 'plain' spatial data*, Geoconnexion, June 2003*

OGC (1999): OpenGIS Simple Features Specification for SQL. Revision 1.1, OpenGIS Project Document 99-049.

OGC (1998): The OpenGIS Guide, third edition. An introduction to Interoperable Geo-processing. The OGC Project Technical Committee of the OpenGIS Consortium, edited by Buhler and K. McKee, L., Wayland, Mass., USA.

Oosterom, P.J.M. van and C.H.J. Lemmen (2001): Spatial data management on a very large cadastral database, Computers, Environments and Urban Systems (CEUS). Volume 25, number 4-5, 2001.

Oracle (2001): Oracle Spatial User's Guide and Reference Release 9.0.1 Part Number A88805-01, June 2001.