Oregon Relational Spatial Topology (ORST) Representation

by

David Hunt

Submitted to Oregon State University

In partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in Computer Science

Computer Science Department

Oregon State University

Corvallis, OR

August 2006

Abstract

We designed a concise way to store and manipulate GIS *coverage* data in a *geospatial* database. Our geospatial database is implemented with PostgreSQL and PostGIS. PostgreSQL is an object-relational database, and PostGIS supports various geospatial operations as an SQL extension. In our Oregon Relational Spatial Topology (ORST) approach, *topological relationships* among polygons, arcs, and points are represented explicitly. With this explicit representation of polygon data, such spatial operations as *moving a point*, *merging a polygon*, and *splitting a polygon* can be supported with relative ease. In order to populate this database, we developed a process for converting polygon data stored as an ESRI shapefile first to the ESRI E00 coverage format and then to the ORST representation. We also implemented the spatial operations discussed above.

TABLE OF CONTENTS

1. Introduction

A *shapefile* is a popular binary data format for *geospatial vector data*. Spatial data stored in a shapefile does not explicitly represent *topological relationships* among polygons. Without the topological relationships, it is not easy to manipulate polygons while maintaining correct topological relationships among them. One way to obtain the topological relationships is to convert the shapefile data into a *coverage*, which is a proprietary format developed by ESRI, a company specializing in GIS software. ESRI provides another text-based data format called E00 for coverage data. This format is proprietary. However, a specification, although incomplete, has been published by a third party.

The *Oregon Relational Spatial Topology* (ORST) representation allows topological relationships among polygons to be specified within a database. For this purpose, we used relational tables in the *object-relational* DBMS PostgreSQL augmented with PostGIS [1]. PostGIS supports spatial operations as an SQL extension compatible with the OGC (Open Geospatial Consortium) specification [2] [3].

In converting coverage data in the E00 format to the ORST representation, specific sections of the E00 file are parsed and SQL insert statements are generated. These insert statements can be executed on a database supporting PostGIS. The parcel data for the City of Corvallis, Oregon was converted to the ORST representation as a test case.

We then implemented the spatial operations *moving a point*, *merging a polygon*, and *splitting a polygon* for the coverage data in the ORST representation. Moving a point allows a user to move an end point of an arc to a new location. When an end point of an arc is moved, the geometry of the arc is recomputed with the new coordinates of the moved point. Then the geometries of the polygons sharing the arc are recomputed. To merge two polygons into a single polygon, the user can delete an arc shared by those two polygons. To split a polygon into two new polygons, the user can draw an arc through a polygon.

Section 2 gives an overview of the ORST representation. We discuss the E00 format in Section 3, concentrating on the parts needed to create coverage data in the ORST representation. In Section 4 implementation of the spatial operations *Move Point*, *Merge Polygon*, and *Split Polygon* are discussed. Section 5 concludes this report.

2. Overview of Oregon Relational Spatial Topology (ORST) Representation

The ESRI *shapefile* format is a popular binary data format for geospatial *vector* data [4]. A shapefile stores such features as *line strings*, *points*, and *polygons*, and the attributes associated with them in multiple files. When polygons are stored in a shapefile, *topological relationships* among the polygons are not explicitly represented. On the other hand, a *coverage*, which is another ESRI GIS data format, can represent topological relationships among polygons explicitly. Figure 1 shows three polygons (a) in a shapefile and (b) a coverage.

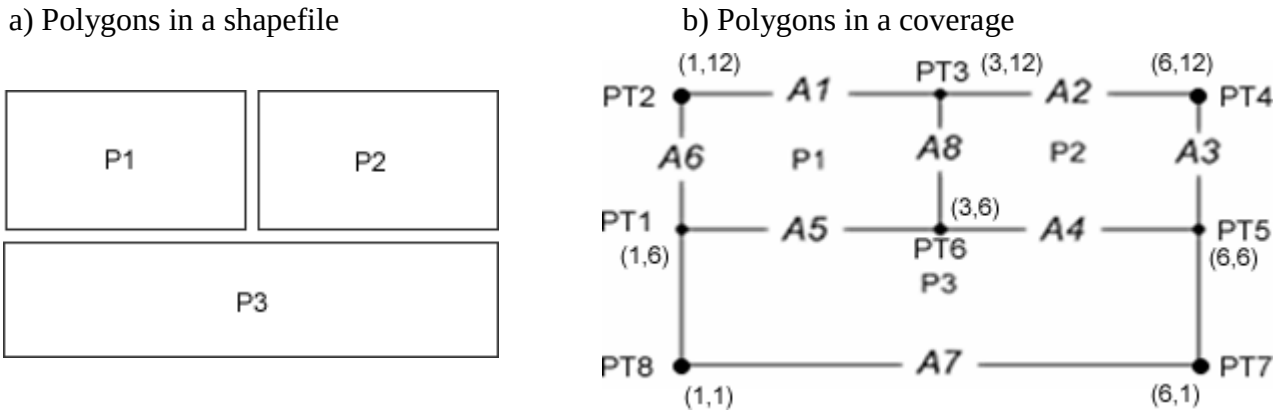


Figure 1: Shapefile vs. Coverage representations.

We now explain the major difference between a shapefile and a coverage by using the example in Figure 1. In the shapefile representation, the right edge of polygon P1 and the left edge of polygon P2 are different edges. On the other hand, in the coverage representation, polygon P1 and P2 share the edge A8, which is called an *arc*.

1. Assume that the user moves the right edge of polygon P1. In the shapefile representation, the left edge of polygon P2 does not move, even though the right edge of polygon P1 is identical to the left edge of polygon P2. In the coverage representation, arc A8 is moved, with both polygon P1 and P2 being reshaped as they share arc A8.

2. Assume that the user wants to merge polygon P1 to P2. In the shapefile representation, the user needs to delete polygons P1 and P2 and create a new polygon by merging them. In the coverage representation, the user can remove arc A8 to merge polygons P1 and P2.

In the ORST representation, we follow the standard approach for storing coverage data in relational tables, whose schema is given in Figure 2.

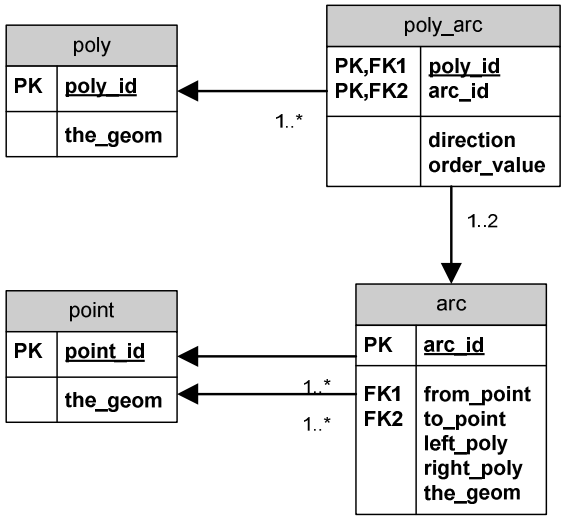


Figure 2: ORST representation of coverage data.

The end points of the arcs are stored in table `point`. The records in table `arc` designate arcs. The start point and the end point of an arc are specified by attributes `from_point` and `to_point`, respectively. The polygon records are stored in table `polygon`. Each arc has one polygon on its left side and right side. These polygons are designated by attributes `left_poly` and `right_poly`, respectively, of an arc record. Table `poly_arc` define the arcs constituting each polygon. The `order_value` of a `poly_arc` record designates the sequence number of the arc designated by the `arc_id`, where the arc is an edge of the polygon designated by `poly_id`. The `direction`, `f(orward)` or `r(everse)`, tells whether the arc should be traversed from the `from_point` to the `to_point` or from the `to_point` to the `from_point`, respectively.

We now list the attributes in each table shown in Figure 2.

Table `point` stores data for the points:

- `point_id` – Unique identifier for the point.
- `the_geom` – Geometry of the point represented as an `(x, y)` pair.

Table `arc` stores the geometry and data for the arcs connecting points:

- `arc_id` – Unique identifier for the arc.
- `from_point`- Identifier for the start point of the arc.
- `to_point` – Identifier for the end point of the arc.
- `left_poly` – Identifier for the polygon to the left of the arc.
- `right_poly` – Identifier for the polygon to the right of the arc.
- `the_geom` – Geometry of the arc represented as a line string. The coordinate values of the start and end points of an arc must match the coordinate values of those points in table `point`. An arc may contain intermediate points in between the start and end points.

Table `poly` stores data for polygons:

- `poly_id` – Unique identifier for the polygon.
- `the_geom` – Geometry of the polygon represented as a multi-polygon. The geometry of the polygons is redundantly stored so that a GIS application can render the polygon.

Table `poly_arc` stores the relationship data between polygons and arcs:

- `poly_arc_id` – Unique identifier for the `poly_id` and `arc_id` combination.
- `poly_id` – Unique identifier of the polygon formed by the arc designated by `arc_id`.
- `arc_id` – Unique identifier of the arc which is an edge of the polygon designated by `poly_id`.
- `direction` – Direction in which arc is traversed, where the arcs forming the polygon are visited clockwise order.
- `order_value` – Sequence number of the arc forming the polygon.

Figure 3 shows the content of the tables for the coverage specified in Figure 1 (b). The values -1 in the `left_poly` and `right_poly` attributes of table `arc` designates the *universal polygon*, which is the surrounding area of all the polygons defined in table `poly`.

point	
point_id	the_geom
PT1	. . .
PT2	. . .
PT3	. . .
PT4	. . .
PT5	. . .
PT6	. . .
PT7	. . .
PT8	. . .

Arc					
arc_id	from_point	to_point	left_poly	right_poly	the_geom
A1	PT2	PT3	-1	P1	. . .
A2	PT3	PT4	-1	P2	. . .
A3	PT4	PT5	-1	P2	. . .
A4	PT5	PT6	P3	P2	. . .
A5	PT6	PT1	P3	P2	. . .
A6	PT1	PT2	-1	P1	. . .
A7	PT5	PT1	-1	P3	. . .
A8	PT6	PT3	P1	P2	. . .

poly	
poly_id	the_geom
P1	. . .
P2	. . .
P3	. . .

poly_arc			
poly_id	Arc_id	direction	order_value
P1	A6	f	1
P1	A1	f	2
P1	A8	r	3
P1	A5	f	4
P2	A8	f	1
P2	A2	f	2
P2	A3	f	3
P2	A4	f	4
P3	A7	f	1
P3	A5	r	2
P3	A4	r	3

Figure 3: ORST representation of coverage data.

A major objective of this project was to devise a process to convert a shapefile to the ORST representation, and we came up with the process shown in Figure 4 as one that required the least amount of our work.

Figure 4: Process to convert a shapefile into an ORST representation.

The process consists of the following two steps.

1. A shapefile is first converted to the ESRI proprietary E00 coverage format with the conversion tool in the ArcToolbox of ESRI ArcMap. This coverage can be converted to the E00 format again with the conversion tool in ArcToolbox. The sequence of the commands used by this step can be found in Appendix A.
2. The coverage in the E00 format is parsed with a PHP script, and the SQL `INSERT` statements for the ORST representation are generated [5]. The script parses only the `ARC` and `RPL` section in the E00 coverage file.

Figure 5 gives examples of the `ARC` and `RPL` sections of an E00 file. The formats of these sections are explained in Section 3.

a) `ARC` section for points and arcs

```
  1       1      5      1      -1          3          3
 6.000000000000000E+06 6.000000000000000E+05
 6.000000000000000E+06 1.000000000000000E+05
 1.000000000000000E+06 1.000000000000000E+05
 1.000000000000000E+06 6.000000000000000E+05
```

b) `RPL` section for a polygon

```
3  1.000000000000000E+06  1.000000000000000E+05   6.000000000000000E+06   6.000000000000000E+05
      7          0          0          -5          0          0
     -4          0          0
```

Figure 5: Examples of `ARC` (a) and `RPL` (b) sections in an E00 file.

The SQL INSERT statements generated from the E00 data given in Figure 5 are shown in Figure 6.

a) Insert for table point.

```
Insert into e00_point(point_id,the_geom)values
(1, GeomFromText('POINT(1272815.12498623 3328705.62537365)', 32026));
```

b) Insert for table arc.

```
Insert into e00_arc(arc_id,from_point, to_point, left_poly, right_poly,
the_geom)values(1,1,3,1,-1, GeomFromText('LINESTRING (1272815.12498623
3328705.62537365, 1272752.87502497 3328387.18707713, 1272731.75015514
3328279.68742550)', 32026));
```

c) Insert for table poly.

```
Insert into e00_rpl(poly_id, the_geom) values(8,
GeomFromText('MULTIPOLYGON(((1272815.12498623 3328705.62537365, 1272752.87502497
3328387.18707713, 1272731.75015514 3328279.68742550, 1269840.443398000000
331915.382004284000,1269932.322300500000 331912.755438105000, 1272815.12498623
3328705.62537365)))', 32026));
```

d) Insert for table poly_arc.

```
Insert into e00_poly_arc_test(poly_arc_id, poly_id, arc_id, order_value,
direction)values(1, 1,1, 1,'f');
```

Figure 6: SQL INSERT statements generated from the E00 file.

3. ESRI E00 Format for GIS Data

The E00 format for coverage data is a proprietary data format of ESRI. However, the specification of the E00 format has been published, although not complete, by a third party [6]. An E00 file consists the following sections:

- ARC –sets of data on the arcs
- CNT – center points of the polygons
- LAB – sets of labels for polygons
- LOG – coverage edit history
- PAL – repeating sets of data on the polygons
- PRJ – map projection data

- SIN – spatial index
- TOL – multiple lines defining the tolerance type, the tolerance status, and the tolerance value
- TXT – annotations
- TX6/TX7 – annotations
- RXP – associations of polygons within a region and the PAL section
- RPL - sets of data on the polygons for regions

Coordinate values contained in a E00 file are either 8-digit floating point (single precision) values or 15-digit floating point (double precision) values. In the current version of the ORST implementation, we only support double precision values. In converting data to the ORST representation, only the ARC and RPL sections of the E00 file are used.

We now explain the format of an ARC section by using an example given in Figure 7.

```
1       1      5      1      -1          3          3
 6.00000000000000E+06 6.00000000000000E+05
 6.00000000000000E+06 1.00000000000000E+05
 1.00000000000000E+06 1.00000000000000E+05
 1.00000000000000E+06 6.00000000000000E+05
```

Figure 7: ARC section listing.

The attributes in the top row of the ARC section have the following meaning:

1. 1: coverage number – not converted
2. 1: coverage ID – not converted
3. 5: from node – id of the first point in the current arc
4. 1: to node – id of the last point in the current arc
5. -1: left polygon – id of the polygon on the left of the arc
6. 3: right polygon – id of the polygon on the right of the arc
7. 3: number of coordinates – number of point geometries in the arc

Attributes 3 – 6 are the unique identifiers of the points and the polygons. These values are used also in tables `point` and `poly`. Each row after the first contains `(x, y)` values of a point contained in the arc. The second row is for the `from node,` and the final row for the `to node`.

We now explain the format of an `RPL` section by using again an example given in Figure 8.

```
3  1.00000000000000E+06  1.00000000000000E+05   6.00000000000000E+06   6.00000000000000E+05
     7          0          0          -5          0          0
    -4          0          0
```

Figure 8: `RPL` section listing.

The attributes in the top row of the `RPL` section have the following meaning:

1. 3: `number of arcs` in the current polygon
2. 1.00000000000000E+06: `x min` of the current polygon
3. 6.00000000000000E+05: `y min` of the current polygon
4. 6.00000000000000E+06: `x max` of the current polygon
5. 6.00000000000000E+05: `y max` of the current polygon

The subsequent lines contain one or two sets of three values. In our example, the second line contains two sets and the third one set. Each set contains the following values:

1. 7: `Arc_Id` – ID of the arc contained in the current polygon
2. 0: `From_Node_Id` - not converted
3. 0: `Adjacent_Polygon_Id` - not converted

A coverage produced from a shapefile contains data on a single region. The `RPL` section contains for each polygon a complete list of the ids of the arcs forming the polygon. If one of the arc IDs is zero, then the polygon contains a hole as shown in Figure 9. The arc IDs following the zeros represent the inner polygon, which is a hole.

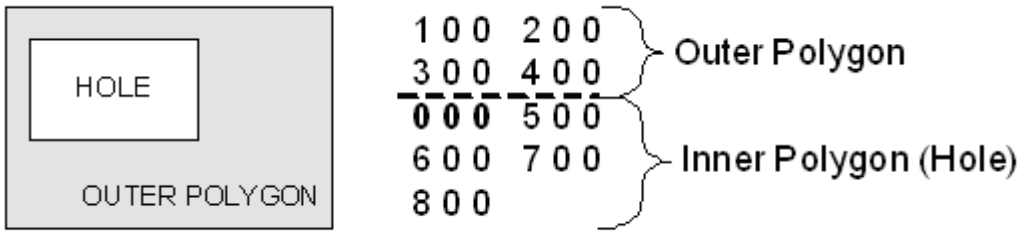


Figure 9: Polygon containing a hole

The RPL section defines the same polygons as those in the PAL section in the E00 file. We used the data in the RPL section instead of those in the PAL section because of the following reasons.

1. The polygons are listed in the RPL section in the same order as in the original shapefile.
2. If the PAL section is used, we need to parse the RXP section to determine the correct polygon ID ordering.

The PAT section of the E00 file contains the attributes of the polygons. However, we use the attribute data in a table created directly from a shapefile, and hence the PAT section is not parsed.

4. Advanced Polygon Manipulation Features

4.1 Move Point Operation

We want to allow the user to move a point node with a Move Point operation. The current implementation of the operation supports moving only the `from point` and the `to point` of an arc. A future release may support a Move Point operation for any point within an arc. When a point is moved, the arcs attached to the point and the geometry of the polygon containing those arcs must be updated as shown in Figure 10.

a) Before moving point PT3 **b) After moving point PT3**

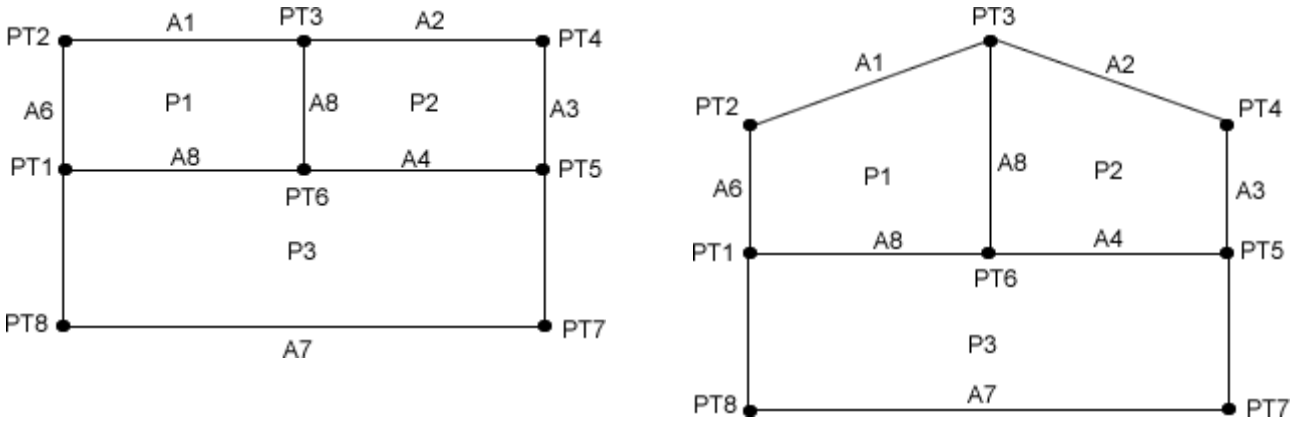


Figure 10: Move Point operation on point PT3.

To update a point, the user selects the point on the map and drags it to a new location. The point is updated in the point `table` with the new coordinates.

The code for updating the arcs is shown in figure 11.

```
1)  $affected_polys = array();  //holds ids of polygons that need to be updated
2)
3)  //get all arcs that contain moved point
4)  $sSQL_arcs = sprintf("Select arc_id, from_point, to_point, " .
5)"left_poly, right_poly, astext(the_geom) as the_geom" .
6)" from arc where from_point = %s or to_point = %s", $point_id, $point_id);
7)
8)  $result = pg_query($link, $sSQL_arcs);
9)
10)  //update each arc with updated point
11)  while($row = pg_fetch_assoc($result)){
12)  $new_arc = ""; //holds updated geometry of arc
13)
14)  //check if point matches from point of arc
15)  if($row[from_point] == $point_id){
16)    //overwrite old from point with new point
17)    $new_arc = "LINESTRING(" . $point . "," .
18)  substr($row[the_geom], strpos($row[the_geom], ",") + 1);
19)    }
20)  //point matches arcs to point
21)  else{
22)    //overwrite old to point with new point
23)    $new_arc =  substr($row[the_geom], 0,
24)  strrpos($row[the_geom], ",") + 1) . "$point)";
25)    }
26)
27)  //check if polygon containg the arc has already been added to affected_polys
28)  if(!in_array($row[left_poly], $affected_polys)){
29)    array_push($affected_polys, $row[left_poly]);
30)  }
31)  if(!in_array($row[right_poly], $affected_polys)){
32)    array_push($affected_polys, $row[right_poly]);
33)  }
32)
33)  //update arc with new geometry
```

```
34)  $sSQL_update_arc = sprintf("Update arc set the_geom = geomfromtext('%s',%s) "
35)        . "where arc_id = %s", $new_arc, $data_srid, $row[arc_id]);
36)
37) pg_query($link, $sSQL_update_arc);
```

Figure 11: Updating the arcs affected by a Move Point operation.

Once the arcs adjacent to the point moved are retrieved, (lines 4-8), then all the end points of arcs that match the point moved are updated (lines 9-25 and 34-37). When each of the affected arcs are updated, the IDs of the polygons that are on the left or right of that arc are also saved for future processing (lines 27-33). The geometry of those polygons need be updated with the new arc data.

After the arcs affected by the modified point are updated, table `poly_arc` is queried with each ID stored in `$affected_polys`, to get the list of all the arcs forming the polygon and the geometry of the polygon is updated. The code for this process is shown in Figure 12.

```
38)  //this loop updates each polygon with the new arcs
39)  foreach($affected_polys as $poly_id){
40)    //if polygon id is -1 then no update needed
41)   if($poly_id <> -1){
42)    $begin_poly = true; //indicates if arc should be the first arc in the polygon
43)    $poly_geom = "MULTIPOLYGON(((''; //holds new polygon geometry for updating
44)
45)    //gets the order and direction of the arcs for the polygon
46)    $sSQL_poly_arc = sprintf("Select poly_arc.arc_id, order_value, direction," .
47)        "astext(the_geom) as the_geom from poly_arc " .
48)       "join arc on arc.arc_id = poly_arc.arc_id " .
49)        " where poly_id = %s order by order_value", $poly_id);
50)
51)   $result = pg_query($link, $sSQL_poly_arc);
52)
53)   //add each arc to the updated polygon
54)    while($row = pg_fetch_assoc($result)){
55)     $hole = false; //indicates if polygon has a hole
56)      $arc_geom = ""; //holds geometry of arc to add to polygon
57)
58)     //check if arc should be reversed
59)     if($row[direction] == "r"){
60)       $arc_geom = TrimGeom(ReverseArcGeom($row[the_geom], $link));
61)      }
62)      //check if arc does not need to be reversd
63)     elseif($row[direction] == "f"){
64)       $arc_geom = TrimGeom($row[the_geom]);
65)      }
66)     //polygon contain a whole since direction value is h
67)      else{
68)        $hole = true;
```

```
69)       $poly_geom = $poly_geom . "),(";
70)       $begin_poly = true;
71)     }
72)     //if polygon did not have a hole then add arc to polygon
73)     if(!$hole){
74)       //if arc is first arc in polygon or hole
75)       if($begin_poly){
76)         $poly_geom = $poly_geom . $arc_geom . ",";
77)         $begin_poly = false;
78)
79)       //arc is not first arc in polygon or hole
80)       else{
81)         $poly_geom = $poly_geom . TrimFirstPoint($arc_geom) . ",";
82)       }
83)     }
84)  }
85)  //remove extra , at end of polygon
86)  $poly_geom = rtrim($poly_geom, ",") . ")))";
87)
88)  //update polygon with geometry
89)  $sSQL_update_poly = sprintf("update poly set the_geom = geomfromtext('%s', %s) " .
90)    "where poly_id = %s", $poly_geom, $data_srid, $poly_id);
91)
92)   pg_query($link, $sSQL_update_poly);
93)  }
```

Figure 12: Code for updating the polygons when a point is moved.

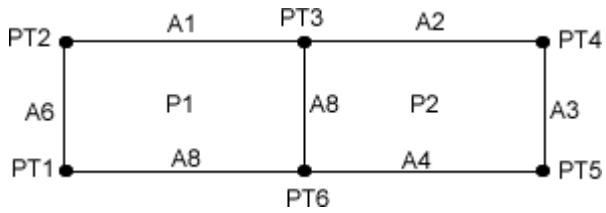
The process of updating the polygons consists of the following steps.

1. The values of attributes the `arc_id`, `order_value`, `direction`, and `the_geom` of the arcs in the polygon are retrieved from tables `poly_arc` and `arc` (lines 46-51). The `direction` value is checked for a `f`, `r`, or `h` value (lines 59, 63, and 67).
2. If a `direction` value was a `f` or `r`, then the arc is added to the polygon. Otherwise, a hole must be created (line 69). If the arc is the first arc in the polygon, then it can be added to the set of the nodes for the polygon (line 76). If the arc is not the first edge of the polygon then the first point of the arc must be removed before the rest of the points of the arc arc added to the polygon (line 81).
3. The geometry of the polygon is updated (lines 89-92).
4. Steps 1-4 are repeated for each polygon whose ID is in `$affected_polys` (line 39).

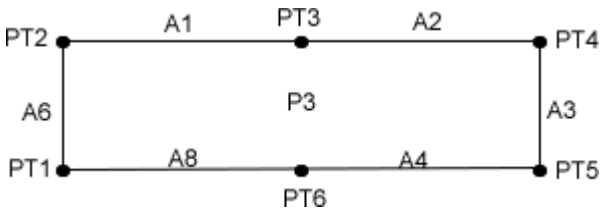
4.2 Merge Polygon Operation

We want to allow the user to merge two polygons with a Merge Polygon operation. After selecting this operation, the user can select an arc to be deleted. Then the polygons on the left and right sides of the arc are merged into a single polygon. A special handling is needed when a polygon is merged with the universal polygon. In this case, the polygon is actually deleted, since the universal polygon is the surrounding area. Figure 12 illustrates the normal and special cases for merging polygons.

a) Polygons before merging



b) Removing arc A8



c) Removing arc A3

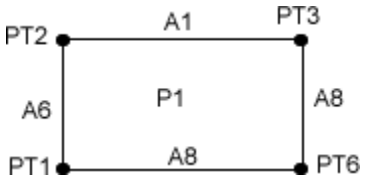


Figure 12: Merge Polygon operation.

Polygons *P1* and *P2* can be merged into new polygon *P3* as follows.

1. Let the sets of the arcs of *P1* and *P2* be *arcs*(*P1*) and *arcs*(*P2*), respectively. Compute the set of arcs of *P3* as *arcs*(*P3*) = *arcs*(*P1*) $\cap$ *arcs*(*P2*) – *arcs*(*P1*) $\cup$ *arcs*(*P2*). The set *arcs*(*P1*) $\cap$ *arcs*(*P2*) represents all the arcs each of which is used by *P1* or *P2*. The set *arcs*(*P1*) $\cup$ *arcs*(*P2*) represent the arcs shared by *P1* and *P2* along the common boarder.
2. Delete the arcs in the set *arcs*(*P1*) $\cup$ *arcs*(*P2*) from tables `arc` and the related entries in table `poly_arc`. Also, the points adjacent to those arcs need be deleted from table `point` if they are not adjacent to any of the remaining arcs.

3. Insert polygon *P3* into table `poly` and update the entries in tables `arc` and `poly_arc` for *arcs*(*P3*) . The geometry of *P3* need be computed from *arcs*(*P3*).

4. Delete polygons *P1* and *P2* from table `poly`.

Step 1 is accomplished by the following SQL statement.

```
arcs(P3) = select arc.arc_id, from_point, to_point,
astext(the_geom) as the_geom,
order_value, direction  from arc, poly_arc
where (poly_id = P1 or poly_id = P2) and arc.arc_id = poly_arc.arc_id
except
select arc.arc_id, from_point, to_point,
astext(the_geom) as the_geom,
order_value, direction  from arc, poly_arc,
(select p1.arc_id from poly_arc as p1, poly_arc as p2
where ((p1.poly_id = P1 and p2.poly_id = P2)
and (p1.arc_id = p2.arc_id and p2.arc_id = p1.arc_id))) as arc1
where (poly_arc.arc_id = arc1.arc_id)
and (poly_arc.arc_id = arc.arc_id)
```

Step 2 is accomplished by the following sub-steps repeated for each arc.

i. The following SQL statement removes all the arcs that *P1* and *P2* have in common.

```
delete from arc where arc_id in (select p1.arc_id, from_point,
to_point from arc, poly_arc as p1, poly_arc as p2
where (p1.poly_id = P1 and p2.poly_id = P2)
and (p1.arc_id = p2.arc_id) and (p1.arc_id = arc.arc_id))
```

ii. This following SQL statement removes the `poly_arc` entries for the arcs deleted by the previous step.

```
delete from poly_arc where arc_id = id_of_deleted_arc
```

iii. The following pseudo-code removes the end points that are not used by any of the remaining arcs.

```
from_point_count = select count(arc_id) from arc where
   from_point = deleted_arc[from_point] or
   to_point = deleted_arc[from_point]

to_point_count = select count(arc_id) from arc where
```

```
    from_point = deleted_arc[to_point] or
    to_point = deleted_arc[to_point]

  if (from_point_count  <= 1) {
    delete from point where point_id = deleted_arc[from_point]
    }
    if (to_point_count <= 1) {
      delete from point where point_id = deleted_arc[to_point]
```

Step 3 is accomplished by the following pseudo-code shown in Figure 13.

```
1)  foreach(merge_arc) {
2)    if(merge_arc[direction] == f) {
3)      forward_arcs[merge_arc[from_point]] = merge_arc
4)      next_point = [merge_arc[from_point]
5)    } else
6)      reverse_arcs[merge_arc[to_point]] = merge_arc
7)    }
8)  }
9)
10) first_arc = true;
11) for(i = 0; i < sizeof (forward_arcs) + sizeof (reverse_arcs) ; i++) {
12)   if (exists (forward_arcs, next_point) {
13)     if(first_arc) {
14)     poly_geom = forward_arcs[next_point][the_geom]
15)     first_arc = false
16)   } else {
17)       poly_geom = poly_geom .
18)         remove_first_point (forward_arcs[next_point][the_geom])
19)   }
20)  update poly_arc set order_value = $i where poly_id
21)  forward_arcs[next_point][poly_id]and arc_id = forward_arcs[next_point][arc_id]
22)
23)  next_point = forward_arcs[to_point]
24)   } else {
25)       if (first_arc) {
26)         poly_geom = reverse_arc (reverse_arcs[next_point][the_geom])
27)       } else {
28)           poly_geom = poly_geom .
29)           remove_first_point (reverse_arc (reverse_arcs[next_point][the_geom]))
30)       }
31)       update poly_arc set order_value = $i where poly_id
32)       reverse_arcs[next_point][poly_id]and
33)       arc_id = reverse_arcs[next_point][arc_id]
34)
35)       next_point = reverse_arcs[from_point]
36)   }
37)  P3_id = Insert into poly (the_geom) values (poly_geom)
38)
39)  foreach (forward_arcs as arc) {
40)   Update arc set right_poly = P3_id where arc_id = arc[arc_id]
41)  }
42)  foreach (reverse_arcs as arc) {
43)    Update arc set left_poly = P3_id where arc_id = arc[arc_id]
44)  }
45)  update poly_arc set poly_id = P3_id where poly_id = P1_id or poly_id = P2_id
46) }
```

Figure 13: Pseudo-code to merge two polygons.

The arcs in *arcs*(*P3*) are split into two arrays (line 1-8). All arcs with a direction value of `f` are placed into array `forward_arcs,` using the arcs `from_point` ID as the index value (lines 2 and 3). The arcs with a direction value of `r` are placed into array `reverse_arcs`, using the arcs `to_point` ID as the index value (lines 5 and 6). Variable `next_point` contains the end point ID of the first arc to be added to *P3* (line 4). The `forward_arcs` and `reverse_arcs` are checked to find which array contains the point ID contained in `next_point` (lines 12 and 21). When the array with the index value matching `next_point` is found, the arc is added to the geometry of *P3* (lines 13-18 and 22-26). If the added arc was from the `forward_arcs` array then the new `next_point` will be the arcs `to_point` (line 20). Otherwise, the `next_point` will be the arcs `from_point` ID (line 28). As each arc is added to *P3*, the sequence number is updated in table `poly_arc` (lines 20 and 31). After *P3* has been inserted and its ID returned, the `left_poly` or `right_poly` IDs are updated for each arc (lines 32 – 37). For each arc contained in `forward_arcs,` the `right_poly` ID is updated with *P3*'s ID (line 33). For each arc contained in `reverse_arcs,` the `left_poly` ID is updated with *P3*'s ID (line 36). The IDs of polygons *P1* and *P2* are updated with *P3*'s ID in table `poly_arc`.

Step 4 is accomplished by the following SQL statement.

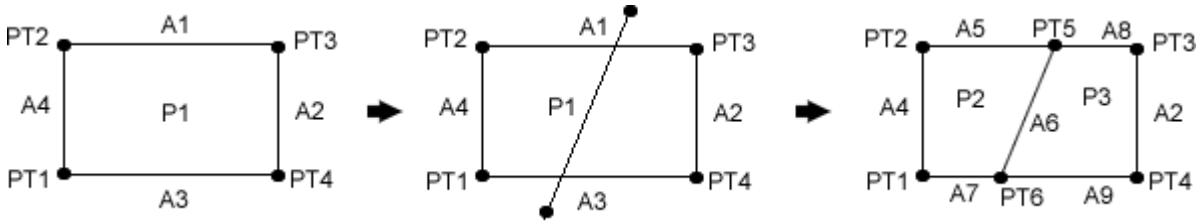
```
delete from poly where poly_id = P1 or poly_id = P
```

4.3 Split Polygon Operation

The Split Polygon operation allows the user to split a polygon into two new polygons. After selecting this operation, the user can draw an arc to split a polygon, with the splitting arc as the common boarder. Figure 14 illustrates the polygon P1 being split. The user can choose to draw an arc that intersects one or two arcs used by a single polygon. Figure 14 (a) illustrates a case when the arc splitting the polygon intersects two edges of the polygon. In this case, each of the intersected arcs is divided into two new

arcs. Figure 14 (b) illustrates a case where one arc is intersected. When this happens, the intersected arc is divided into three new arcs.

a) Splitting polygon P1 by intersecting arc A1 and A3.



b) Splitting polygon P1 by intersecting arc A1 twice.

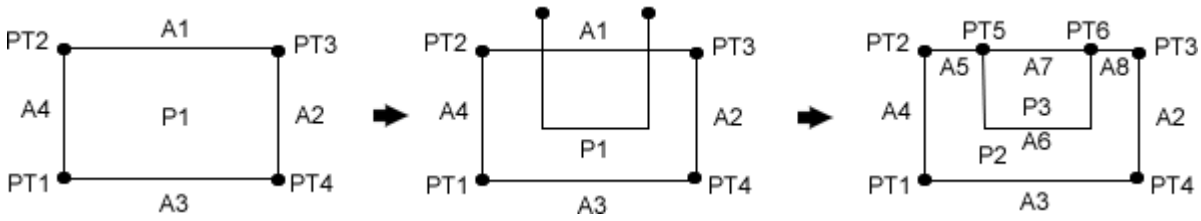


Figure 14: Split Polygon operation.

Polygon P can be split into polygons P_1 and P_2 as follows.

1. Use the split arc A_s to compute the arcs intersected and the intersection points. Then use the intersection points as the end points of A_s. In Figure 14 (a), PT5 and PT6 are intersection points and A6 is A_s.
2. Obtain the arcs of P that were not intersected by A_s as the set *arcs*(P) . In Figure 14 (a), P = P1 *arcs*(P) = {A2, A4}.
3. If two arcs are intersected, let them be A_i and A_j, compute the arcs A_{i1} and A_{i2} from A_i and A_{j1} and A_{j2} from arc A_j. In Figure 14 (a), A_i = A1, A_j = A3, A_{i1} = A5, A_{i2} = A8, A_{j1} = A7, and A_{j2} = A9. If only one arc, Ai is intersected, compute the arcs A_{i1}, A_{i2}, and A_{i3} from arc A_i. In Figure 14 (b), A_i = A1, A_{i1} = A5, A_{i2} = A7, and A_{i3} = A8.
4. Create P_1 and P_2 with the arcs contained in *arcs*(P), A_{i1}, $A_{i2,}$ A_{j1}, A_{j2}, and A_s. Insert P_1 and P_2 into table `poly`. In Figure 14 (a), P_1 = P2 and P_2 = P3.

5. Insert the new arcs into table `arc` and add the entries for P_1 and P_2 into table `poly_arc`.
6. Delete the entries for P in table `poly_arc` and P from table `poly`.
7. Update the entries in table `poly_arc` for the `left_poly` or `right_poly` contained in A_i and A_j. The entries for A_i or A_j are deleted and replaced with entries for A_{i1} and A_{i2} or A_{j1} and A_{j2}. When the new arcs are added the values of `order_value` for the arcs after A_i or A_j are reassigned.
8. Delete arcs A_i and A_j from table `arc`.

The following pseudo-code shown in Figure 15 describes the outline of Step 1.

```
1) intersected_arcs = select arc_id, left_poly, right_poly,from_point, to_point,
2) astext(the_geom) as the_geom,astext(intersection(the_geom, As) as intersect_geom
3) from arc where intersects(the_geom, As)
4)
5) if (sizeof(intersected_arcs) == 1) {
6) intersect_pt1 = get_first_point (intersected_arcs[0][intersect_geom])
7) intersect_pt2 = get_second_point (intersected_arcs[0][intersect_geom])
8) } else {
9) intersect_pt1 = intersected_arcs[0][intersect_geom]
10) intersect_pt2 = intersected_arcs[1][intersect_geom]
11) }
12)
13) split_arc_points = get_points(As)
14)
15) if (split_arc_points == 2) {
16)  As= intersect_pt1 . intersect_pt2
17)   } else {
18)   if (GetPointDistance (intersect_pt1, split_arc_points[1])
19)   < GetPointDistance (intersect_pt1,
20)      split_arc_points[sizeof(split_arc_points)-2])) {
21)   for(i = 1; I < sizeof (split_arc_points) – 1; i++) {
22)  As= split_arc_points[i]
23)   }
24)  As= intersect_pt1 . As.intersect_pt2
25) } else {
26)   for (i = sizeof (split_arc_points) - 2; i >= 1; i--) {
27)  As= split_arc_points[i]
28)   }
29)  As= intersect_pt1 . As.intersect_pt2
30) }
```

Figure 15: Pseudo-code to create split arc.

The arcs intersected by A_s are stored in variable `intersected_arcs` (lines 1-3). If only one arc is contained in `intersected_arcs`, then only one arc is intersected, with the two intersection points being stored in variables `intersect_pt1` and `intersect_pt2` (lines 5-7). If two arcs are intersected, then the intersection point for each arc is stored in variables `intersect_pt1` and `intersect_pt2` (lines 8-11). The points contained within A_s are stored in variable `split_arc_points` (line 13). If A_s contains only two points, then `intersect_pt1` and `intersect_pt2` are the end points (line 15 and 16). If A_s contains more than two points, the new split arc A_s is created with the internal points in the correct order. If the second point in `split_arc_points` is closer to `intersect_pt1`, then all the points in `split_arc_points` except for those at the ends are added between `intersect_pt1` and `intersect_pt2` (lines 18-24). If the point next last is closer to `intersect_pt1`, then all points except the end points of `split_arc_points` are added between `intersect_pt1` and `intersect_pt2` (lines 25-30). Step 2 is implemented with the following SQL statements.

i. Two arcs are intersected by the splitting arc:

```
Select arc.arc_id, arc.left_poly, arc.right_poly,
arc.from_point, arc.to_point, order_value, direction,
astext(the_geom) as the_geom
from poly_arc, arc where poly_id = P
and poly_arc.arc_id = arc.arc_id
and (arc.arc_id != A_i and acr.arc_id != A_j)
```

ii. One are is intersected by the splitting arc:

```
Select arc.arc_id, arc.left_poly, arc.right_poly,
arc.from_point, arc.to_point, order_value, direction,
astext(the_geom) as the_geom
from poly_arc, arc where poly_id = P
and poly_arc.arc_id = arc.arc_id and arc.arc_id != A_i
```

The following pseudo-code shown in Figure 15 describes the outline of Step 3.

The following pseudo-code describes the algorithm employed when two arcs are intersected by a splitting arc.

```
1)  points = split("," , arc_data["the_geom"]);
2)  current_arc = 1;
3)  for (c = 0; c < sizeof(points); c++) {
4)    if(c <> sizeof(points)-1) { //check if we are not at the last point
5)    intersect = get_arc_intersect (point[c] . "," . points[c+1],
6)    split_arc);
7)
8)    if (intersect <> null) {
9)      arc1["from_point"] = arc_data["from_point"];
10)     arc2["to_point"] = arc_data["to_point"];;
11)
12)        if(inter == intersect_point1_data["the_geom"]) {
13)        arc1["to_point"] = intersect_pt1["point_id"];
14)        arc2["from_point"] = intersect_pt11["point_id"];
15)      } else {
16)        arc1["to_point"] = intersect_pt2["point_id"];
17)        arc2["from_point"] = intersect_pt2["point_id"];
18)      }
19)      arc1["the_geom"] = arc1["the_geom"]. points[c] . "," . inter . ",";
20)      arc2["the_geom"] = intersect
21)
22)      current_arc++;
23)    } else {
24)      if(working_arc == 1) {
25)        arc1["the_geom"] = arc1["the_geom"] . points[c] . ",";
26)    } else {
27)       arc2["the_geom"] = arc2["the_geom"] . points[c] . ",";
28)      }
29)    }else {
30)      arc2["the_geom"] = arc2["the_geom"] . points[c];
31)     }
```

Figure 16: Pseudo-code for creating arcs A_{i1}, A_{i2}, A_{j1}, and A_{j2} from arcs A_i and A_j.

The code in Figure 16 is executed once for A_i and again for A_j. The current arc being split is represented by the variable `arc_data`. Variable `points` stores all points that are contained within the given arc (line 1). Variable `current_arc` indicates if the first or second arc being created from A_i or A_j is being created. Since an arc can be have many points between the end points, it must be determined between which two points the intersection point should be inserted (lines 5 and 6). If the intersection location has been found it is stored in variable `intersect,` then the end of the first new arc of A_i or A_j is found and the beginning of the second new arc is created (lines 8–22). The `from point` of the first new arc

is the same from point of A_i or A_j and the `to point` of the second new arc is the same as the `to point` of A_i or A_j (line 9 and 10). The point stored in `intersect` is checked against the first intersection point that was inserted into table `point` from step 1. If the points match then the id of the intersection points contained in `intersect_pt1` and `intersect_pt2` are used so set the `from point` and `to point` IDs (lines 12-14). If the first intersection point does not match then the second point is used to set the `from point` and `to point` IDs (lines 15-18). Then the geometry of the intersection point is added to the two new arcs and the (line 19 and 20). If the intersection point is not found between two points then the points are added to the current arc be created (lines 24-31). The process to create the three new arcs when only one arc is intersected is similar to the above process and will not be discussed.

The following pseudo-code shown in Figure 17 and 18 describes the outline of Step 4.

The following pseudo-code describes the algorithm employed when one arc is intersected.

```
1) if (arc_i1["direction"] == "f") {
2)   new_poly1 = arc_i1["the_geom"] . ",";
3)   $next_point = arc_i1["to_point"];
4)
5)   if (next_point == split_arc["from_point"]) {
6)   new_poly1 = new_poly1 . trim_first_point(split_arc["the_geom"]) . ",";
7)   } else {
8)       new_poly1 = new_poly1
9)        .trim_first_point(reverse_arc_geom($split_arc["the_geom"])) . ",";
10)    }
11)   new_poly1 = new_poly1 . trim_first_point(arc_j1["the_geom"]) . ",";
12)    next_point = arc_j1["to_point"];
13)
14)    done = false;
15)
16)    while (!done) {
17)    if (array_key_exists(next_point, forward_arcs)) {
18)      new_poly1 =
19)      $new_poly1.trim_first_point(forward_arcs[next_point]["the_geom"]). ",";
20)      next_point = forward_arcs[next_point]["to_point"];
21)    } else if (array_key_exists(next_point, reverse_arcs)) {
22)        new_poly1 = new_poly1 . trim_first_point(reverse_arc_geom
23)        (reverse_arcs[$next_point]["the_geom"])) . ",";
24)
25)      next_point = reverse_arcs[next_point]["from_point"];
26)      } else {
```

```
27)       done = true;
28)       }
29)     }
30)     new_poly2 = arc_i2["the_geom"] . ",";
31)     next_point = arc_i2["to_point"];
32)
33)     if (next_point == split_arc["from_point"]) {
34)     new_poly2 = new_poly2 . trim_first_point(split_arc["the_geom"]);
35)     } else {
36)     new_poly2 = $new_poly2 .
37)     trim_first_point(reverse_arc_geom($split_arc["the_geom"])));
38)     }
39)   }
```

Figure 17: Pseudo-code for creating polygons P_1 and P_2 when one arc is intersected.

Variables `arc_i1`, `arc_i2` and `arc_j1` contain data on the arcs created from the single intersected arc A_i. Variable `next_point` indicates which end point of an arc to find next. To create P_1 , `arc_i1` is added and `next_point` is computed (lines 2 and 3). The next two arcs added are `split_arc` and `arc_j1` (lines 5-12). The arcs that were not affected by the split were divided into the two arrays `forward_arcs` and `reverse_arcs` by the algorithm that is used also by the Merge Polygon operation. The arcs in `forward_arcs` and `reverse_arcs` are added to the new polygon geometry until the arc with a matching `next_point` is not found (lines 16-29). After all the arcs are added to *P1*, *P2* is created with `arc_i2` and `split_arc` (lines 30-38). The above code shows the case when the direction of `arc_i1` is `f`. If the direction is `r`, then `_i1`, `arc_i2`, and `arc_j1` need to be reversed first. As each arc is added, the polygon to which the arc belongs to and the sequence value for the arc are computed.

The following pseudo-code describes the algorithm employed when two arcs are intersected by a splitting arc.

```
1) if (start_arc["direction"] == "f") {
2) next_point = start_arc["to_point"];
3)
4) new_poly = TrimGeom(start_arc["the_geom"]) . ",";
5) } else {
6) next_point = start_arc["from_point"];
7) new_poly = reverse_arc_geom(start_arc["the_geom"]) . ",";
8) }
```

```
9) done = false;
10)
11)   while (!done) {
12)   if (array_key_exists(next_point, forward_arcs)) {
13)   new_poly = new_poly .
14)   trim_first_point(forward_arcs[next_point]["the_geom"]) . ",";
15)
16)   next_point = forward_arcs[next_point]["to_point"];
17)   } else if( array_key_exists(next_point, reverse_arcs)) {
18)
19)   new_poly =  new_poly .
20)   trim_first_point(reverse_arc_geom(reverse_arcs[next_point]["the_geom"])).",";
21)
22)   next_point = reverse_arcs[next_point]["from_point"];
23)
24)   } else {
25)     done = true;
26)   }
27)   }
28)   foreach (split_arcs as arc) {
29)   if (arc[“direction”] == “f” && nex_point == arc[“from_point”]) {
30)   next_point = arc["to_point"];
31)   new_poly = new_poly . trim_first_point(arc["the_geom"]) . ",";
32)   break;
33)   }
34)   else if (arc[“direction”] == “r” && next_point == arc[“to_point”]) {
35)   next_point = arc["from_point"];
36)   new_poly = new_poly .
37)                  . trim_first_point(reverse_arc_geom(arc["the_geom"])) . ",";
38)   break;
39)   }
40)   }
41)   if (next_point == split_arc["from_point"]) {
42)   new_poly = new_poly . trim_first_point (split_arc["the_geom"]);
43)   } else {
44)   new_poly = new_poly . trim_first_point (reverse_arc_geom
45)   (split_arc["the_geom"]));
46)   }
```

Figure 18: Pseudo-code for creating polygons P_1 and P_2 when two arcs are intersected.

The code in Figure 18 is executed once for P_1 and again for P_2. The first arc of the new polygon's geometry is stored in variable `start_arc` (lines 1-8). Variable `next_point` indicates which end point of an arc to find next (line 2 and 6). The next set of arcs to connect to `start_arc` are the arcs that were not affected by the splitting of *P* (lines 11-27). The arcs that are not affected by the split are divided into the two arrays `forward_arcs` and `reverse_arcs` by the algorithm that is used also by the Merge Polygon operation. The arcs in `forward_arcs` and `reverse_arcs` are added to the new polygon geometry until the arc with a matching `next_point` is not found. Next, the arcs that are

created from the split operation are checked to find which arc has the matching end point id contained in `next_point` (lines 28-40). The found arc is then added to the new polygon. The last arc to add is the arc that split *P* (lines 41-45). Variable `next_point` is used to determine if the split arc is reversed before being added. As each arc is added, the polygon to which the arc belongs to and the sequence value for the arc are computed.

Step 5 is accomplished by the following sub-steps.

i. The following SQL statement is executed for each new arc created by the split. After each insertion, the ID of the arc is stored for future use in table `poly_arc`.

```
arc[“arc_id”]  =  insert into arc (from_point, to_point, left_poly,
right_poly, the_geom) values(arc[“from_poin”t], arc[“to_point”],
arc[“left_poly”], arc[“right_poly”], arc[“the_geom”]);
```

ii. The following SQL statement is executed for each polygon after the arcs have been inserted into table `arc` and the polygons into table `poly`.

```
insert into poly_arc values (arc[“poly_id”], arc[“arc_id”],
arc[“order_value”], arc[“direction”]);
```

Step 6 is implemented with the following SQL statements.

```
delete from poly where poly_id = P
delete from poly_arc where poly_id = P
```

The following pseudo-code shown in Figure 19 describes the outline of Step 7.

```
1) poly_arc_data = select arc_id, order_value, direction from poly_arc
2)                 where poly_id = update_poly_id order by order_value;
3)
4) found = false;
5) foreach (poly_arc_data as arc) {
6)   if (!found) {
7)     if (arc[“arc_id”] = Ai) {
8)       if (arc[“direction”] = “f”) {
9)         update poly_arc set arc_id = Ai1 where poly_id = update_poly_id
10)         and arc_id = arc[“arc_id”];
11)
12)         insert into poly_arc values (update_poly_id, Ai1,
13)          arc[“order_value”] + 1, “f”);
14)
15)         found = true;
16)     } else if (arc[“arc_id”] = Aj) {
17)         update poly_arc set arc_id = Aj2 where poly_id = update_poly_id
18)         and arc_id = arc[“arc_id”];
```

```
19)
20)         insert into poly_arc values (update_poly_id, A_j1,
21)          arc[“order_value”] + 1, “f”);
22)
23)         found = true;
24)   } else {
25)       update poly_arc set order_value = arc[“order_value”] + 1 where
26)       poly_id = update_poly_id and arc_id = arc[“arc_id”];
27)     }
28) }
```

Figure 19: Updating of table `poly_arc` for the `left_poly` or `right_poly` entries for A_i or A_j.

The polygon ID to be updated is stored in variable `update_poly_id`. Each record in table `poly_arc` for the ID contain in `update_poly_id` is retrieved and stored in `poly_arc_data`, ordered by the records `order_value` attribute (lines 1 and 2). The entries in `poly_arc_data` are searched to find the arc ID matching A_i or A_j (lines 5-29). When a match is found, the new arcs IDs are inserted in place of the old arc and the `order_value` for the second new arc must be incremented by one (lines 8-23). All entries after the matching entry has its `order_value` incremented to account for the new arcs inserted (lines 24-27)

Step 8 is implemented with the following SQL statements.

`delete from arc where arc_id =` A_i `or arc_id =` A_j

5. Conclusion

With the Oregon Relational Spatial Topology (ORST) approach, we can represent topological relationships among polygons explicitly within an object-relational database. We made it possible to convert GIS coverage data in the ESRI E00 coverage format into the ORST representation. The ORST representation allows various spatial operations to be implemented with relative ease.

For our test case we converted the parcel data for the City of Corvallis, Oregon into the ORST representation. By using this data, we created a WebGD (Web-based GIS/database) application. A WebGD application allows its user to *insert*, *query*, *update*, and *delete* geographical features, and it can be rapidly created with the WebGD framework [7]. The advanced spatial operations were implemented within the WebGD framework. Our application performs spatial operations and allows the user to see their results with a web browser.

By using the ORST representation, coverage data can be maintained with an open-source DBMS. With this coverage data the spatial operations implemented can help reduce errors that occur when performing moving, merging, and splitting against data not maintained in a coverage representation such as ORST.

References

[1] PostgreSQL: The world's most advanced open source database. http://www.postgresql.org.

[2] PostGIS: http://postgis.refractions.net.

[3] OpenGIS Simple Feature Specification for SQL Revision 1.1, Open GIS Consortium, Inc. http://www.opengeospatial.org/

[4] ESRI: GIS and Mapping Software. http://www.esri.com.

[5] PHP: Hypertext Preprocessor. http://www.php.net.

[6] Arc/Info Export (E00) Format Analysis. http://avce00.maptools.org/docs/v7_e00_cover.html

[7] Halim, S.. 2005. WebGD: Framework for Web-Based GIS/Database Applications. Oregon State University.

APPENDIX A

Steps to convert a shape file to a coverage

1. Launch **ArcMap**
2. Right Click toolbar and add **ArcToolbox** command (may have to click customize to find)
3. Open **ArcToolbox**
4. Expand **Data Management Tools**
5. Expand **Features**
6. Open Repair **Geometry**
7. Select **shape file**
8. Click **Ok**
9. Shape file is fixed and resaved
10. In ArcToolBox expand **Conversion Tools**.
11. Expand To **Coverage**
12. Chose **Feature Class To Coverage**
13. Select your **shapefile**
14. Give the output coverage a name
15. Click **Environment Settings**
16. Select **Coverage Settings**
17. Chose Highest for **Precision For Derived Coverages**
18. Chose Double for **Precision For New Coverages**
19. Click **Ok**
20. Chose a **Cluster Tolerance** or select Unknown from the drop down
21. Click **OK**

You now have a coverage.

Converting coverage to E00 data format

1. In ArcToolBox expand **Coverage Tools**
2. Expand **Conversion**
3. Chose **Export to Interchange File**
4. Select Cover for **Feature Type**
5. Select your coverage
6. Select **name** and **location** of ouput
7. Click **OK**

You now have an .e00 file.