Efficient Polygon Amalgamation Methods for Spatial OLAP and Spatial Data Mining

Xiaofang Zhou[1], David Truffet[2], and Jiawei Han[3]

[1] Department of Computer Science and Electrical Engineering
University of Queensland, Brisbane QLD 4072 Australia
zxf@csee.uq.edu.au

[2] CSIRO Mathematical and Information Sciences
GPO Box 664, Canberra ACT 2601 Australia
david.truffet@cmis.csiro.au

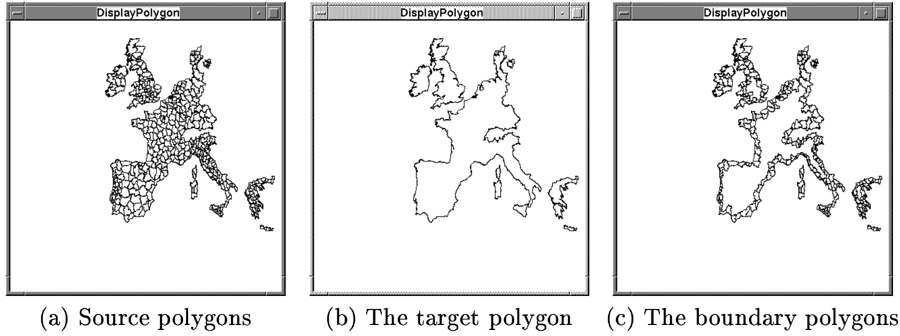
[3] School of Computing Sciences
Simon Fraser University, Burnany, B. C., Canada V5A 1S6
han@cs.sfu.ca

Abstract. The polygon amalgamation operation computes the boundary of the union of a set of polygons. This is an important operation for spatial on-line analytical processing and spatial data mining, where polygons representing different spatial objects often need to be amalgamated by varying criteria when the user wants to aggregate or reclassify these objects. The processing cost of this operation can be very high for a large number of polygons. Based on the observation that not all polygons to be amalgamated contribute to the boundary, we investigate in this paper efficient polygon amalgamation methods by excluding those internal polygons without retrieving them from the database. Two novel algorithms, adjacency-based and occupancy-based, are proposed. While both algorithms can reduce the amalgamation cost significantly, the occupancy-based algorithm is particularly attractive because: 1) it retrieves a smaller amount of data than the adjacency-based algorithm; 2) it is based on a simple extension to a commonly used spatial indexing mechanism; and 3) it can handle fuzzy amalgamation.

Keywords: spatial databases, polygon amalgamation, on-line analytical processing (OLAP), spatial indexing.

1 Introduction

Following the success and popularity of on-line analytical processing (OLAP) and data mining in relational databases and data warehouses, an important direction in spatial database research is to develop spatial data warehousing, spatial OLAP and spatial data mining mechanisms in order to extract implicit knowledge, spatial relationships, and other interesting patterns not explicitly stored in spatial databases [7, 14]. Huge amounts of spatial data have been accumulated in the last two decades by government agencies and other organizations for various purposes such as land information management, asset and facility



(a) Source polygons (b) The target polygon (c) The boundary polygons

Fig. 1. An example of polygon amalgamation.

management, resource management, and environment management. With the maturity of commercial spatial database management systems (SDBMS), it is a trend to migrate spatial data from proprietary file systems to an SDBMS. Thanks to various national spatial data initiatives and international standardization efforts, it is now both feasible and cost-effective to integrate spatial databases from different sources. Dramatic improvements have been made on the accessibility to extensive and comprehensive data sets in terms of geographical coverage, thematic layers and time. With huge amounts of integrated spatial data available, it is an imminent task to develop powerful and efficient methods for the analysis and understanding of spatial data and utilize them effectively.

For efficient OLAP and mining of spatial data, a spatial data warehouse needs to be built [15]. The cost of building a spatial data warehouse is intrinsically higher than building a relational data warehouse. Spatial operations are both I/O-intensive (for retrieving large amounts of spatial objects from database) and CPU-intensive (for performing complex spatial operations). Design of efficient spatial indexing structures and algorithms for processing various spatial operations and queries have been the focus themes in spatial database research [1, 3, 6, 10–13, 20, 21, 23, 24, 26]. Satisfactory performance has been achieved for many spatial database operations. However, in the process of evolving SDMS towards spatial OLAP and spatial data mining, the performance of spatial data processing becomes the bottleneck again since these new applications analyze very large amounts of complex spatial data using costly spatial operations.

Among many spatial operations, we have found that special attention needs to be paid to one particular operation, the *polygon amalgamation* operation. Given a set of *source* polygons, this operation computes a new polygon (called the *target* polygon) which is the boundary of the union of the source polygons. Figure 1 (b) shows the target polygon from merging the source polygons shown in Figure 1 (a). While both intersection and union are basic operations on polygon data, the polygon amalgamation problem, unlike the polygon intersection problem, has received little attention so far in the context of spatial databases. This operation, however, becomes a fundamental operation for emerging new applications such as spatial OLAP and spatial data mining.

Consider a typical scenario in spatial OLAP. A region is partitioned into a set of areas (represented as polygons), where each area is described by some non-spatial attributes (for example, area name, time, temperature and precipitation). Using the spatial data warehouse model proposed in [15, 17], a spatial data cube can be constructed with a spatial dimension (e.g., area) and several non-spatial dimensions (e.g., area name, time, temperature, precipitation). The measures used here can be non-spatial such as daily, weekly or monthly, or spatial such as combined areas according to certain concept hierarchy. Typical OLAP operations such as *drill-down* and *roll-up* can be applied to both spatial and non-spatial dimensions. A roll-up operation, such as generalizing temperature from degrees as recorded in the database into broader categories such as 'cold', 'mild' and 'hot', requires to merge areas according to their temperature degrees. Target polygons generated from such a generalization operation can be used for further operations (e.g., overlay with another spatial layer such as soil types). To spatial data mining, the polygon amalgamation operation is also important. For example, the user may want to group similar or closely related spatial objects into clusters, or to classify spatial objects according to certain feature classes (such as highly developed vs. poorly developed regions) [8, 17, 18]. Such mining will lead to combining polygons into large groups for high level description or inductive inference, using the polygon amalgamation operation.

The above discussion shows that an OLAP operation on a spatial dimension or a clustering operation on a group of spatial objects can result in new polygons at a high level of abstraction. Because of high processing cost associated with the polygon amalgamation operation, it is desirable to pre-compute target polygons and store them in the data cube in order to support fast on-line analysis. Obviously, it is a trade-off between the on-line processing cost for computing target polygons and the storage cost for materializing them. A similar problem in relational OLAP has been investigated by several researchers (e.g., [4, 16]). While materializing every view requires a huge amount of disk space, not materializing any view requires a great deal of on-the-fly and often redundant computation. Therefore, the cuboids (which are sub-cubes of a data cube) in a data cube are typically *partially* materialized. Even when a cuboid is chosen for materialization, it is still unrealistic to pre-compute *every* possible combination of source polygons when there are a large number of source polygons due to a prohibitive amount of storage required for newly generated polygons [15]. In other words, some polygon amalgamation tasks have to be performed on the fly. Moreover, certain types of multi-dimensional analysis require dynamic generation of hierarchies and dynamic computation of polygon amalgamation. In the previous example, different users may define different temperature ranges for 'cold', 'mild' and 'hot'. It might be necessary for some spatial data mining algorithms to try different classification (e.g., adding two more categories 'very cold' and 'warm' in order to find relationships between temperature and vegetation distribution). This type of analysis also demands dynamic polygon amalgamation.

Efficient polygon amalgamation is crucial for both building a spatial data warehouse and on-line processing. A straightforward method for polygon amal-

gamation is to retrieve *all* source polygons from database, and then merge them using some computational geometry algorithm such as described in [22]. Such a simplistic approach can be very time-consuming when the number of source polygons is large. We observe that there exist some *internal* polygons which do not contribute to the boundary of the target polygon. For example, if it is sufficient to use only the polygons shown in Figure 1 (c) to compute the target polygon in Figure 1 (b), a saving can be made by not to fetch and process other internal polygons. Savings from such an optimization can be significant as the number of polygons to be processed is reduced to be proportional to the perimeter of the target polygon, as opposed to its surface area. Obviously, the CPU cost of polygon amalgamation can be reduced by processing a smaller number of polygons. Whether the I/O cost can also be reduced depends on if those internal polygons can be identified *without* retrieving them from the database.

In this paper, we propose two novel methods for identifying internal polygons without retrieving them from the database. The first method uses the information about polygon adjacency, and the other takes an advantage of spatial indexing. Both algorithms are highly effective in reducing CPU and I/O costs. The latter, however, is particularly attractive for several reasons. Comparing with the adjacency-based algorithm, it takes less time in identifying internal polygons. More importantly, it is based on a simple extension to a popular spatial indexing mechanism that is supported by many SDBMSs. Thus, this algorithm can be incorporated easily and efficiently into the SDBMSs supporting that type of indexing mechanism. Another advantage comes when there are holes in target polygons. For spatial OLAP and spatial data mining applications, it is sometimes desirable to ignore holes smaller than certain threshold. These small holes might be insignificant to applications, or caused by imperfect data quality (for example, a small area with a high temperature surrounded by areas with low temperatures can be a 'noise'). We call it *fuzzy amalgamation* if the holes smaller than a specified threshold are to be ignored when merging polygons. Unlike the adjacency-based algorithm, the occupancy-based method can handle fuzzy amalgamation without incurring extra overhead for removing small holes.

The remaining of the paper is organized as follows. In Section 2, we give a basic amalgamation algorithm that processes all source polygons. The adjacency-based and occupancy-based algorithms are discussed in Section 3. A performance study of these algorithms is reported in Section 4. We conclude our discussion in Section 5.

2 A Simple Approach

In this section, we give a simple amalgamation algorithm that processes all source polygons. This algorithm provides a reference for evaluating the performance of other algorithms.

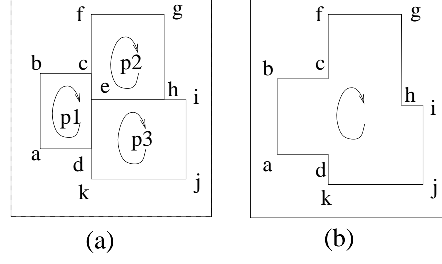


Fig. 2. Polygon representation

2.1 Polygon Representation

Let $\mathcal{D}$ be the data space, and $P = \{p_i | i = 1..n\}$ be a set of polygons inside $\mathcal{D}$ where p_i also denotes the identifier of polygon p_i. Polygon identifiers are unique in P. The boundary of a polygon may be disconnected. For example, the State of New South Wales (NSW) encloses the Australian Capital Territory (ACT), and ACT consists of two disconnected areas. We call a connected component of a polygon a *ring*. Thus, ACT is defined by two rings, and NSW is defined by three rings (one for its outer boundary and two for excluding ACT). In this paper we assume that a ring does not intersect with itself. We also assume that polygons in P do not overlap with each other. We use $t(S)$ to denote the target polygon amalgamated from a set $S \subseteq P$ of source polygons. A polygon p is a *boundary polygon* of S if it shares its boundary with that of $t(S)$. All boundary polygons of S are denoted as ∂S. The polygons in $(S - \partial S)$ (i.e., in S but not in ∂S) are *internal* polygons.

A point is represented by its coordinates (x, y). A line segment l is represented by its start and end points $(l.s,\ l.e)$. A polygon is represented as a sequence of points. For a polygon with k points $v_i \cdots v_k$, its boundary is defined by $k + 1$ connected line segments $(v_1,\ v_2),\ \cdots\ (v_{k-1},\ v_k),\ (v_k,\ v_1)$. After a polygon is fetched from the database, it is unfolded into the form of line segments following, for example, the clockwise order (as in Figure 2). In other words, we view a polygon as a sequence of line segments in this paper. Thus, the polygons in Figure 2 (a) are represented as:

$$p_1 = \langle (a, b), (b, c), (c, d), (d, a) \rangle,$$
$$p_2 = \langle (e, f), (f, g), (g, h), (h, e) \rangle,$$
$$p_3 = \langle (e, i), (i, j), (j, k), (k, e) \rangle.$$

All the line segments in a polygon p are said to be in S if $p \in S$. We use $|S|$ and $||S||$ to denote the number of polygons and the number of line segments in S.

Among many possible relationships between two line segments l and l' on $\mathcal{D}$, we are interested in two relationships which are informally defined as:

1. *congruent*: l and l' are congruent if they are between the same pair of points (i.e., $l.s = l'.s$ and $l.e = l'.e$, or $l.s = l'.e$ and $l.e = l'.s$). In this case we also say these two line segments are *identical*.

2. *touching*: l touches l' if there exists one and only one common point v (termed the *joint point*) between l and l', such that v is an end point of l (i.e., $v = l.s$ or $v = l.e$) and v is not an end point of l' (i.e., $v \neq l'.e$ and $v \neq l'.s$). For example, (b, c) touches (e, f) at point c in Figure 2 (a).

The boundary of $t(S)$ consists of some existing line segments in S, and possibly some new line segments each of which is defined with at least one joint point. The polygon in Figure 2 (b) consists of a set of existing line segments in $S = \{p_1, p_2, p_3\}$ in Figure 2 (a), plus three new line segments (c, f), (h, i) and (k, d) where c, h, and d are joint points. For any algorithm to compute a target polygon, it is necessary to find at least the joint points that define the target polygon. In order to avoid costly operations of splitting lines at joint points, we apply a pre-processing step such that whenever a line segment l of polygon p touches l' of polygon p' at point v, l' in p' is replaced by two new line segments $(l'.s,\ v)$ and $(v,\ l'.e)$. Note that how lines split here depends only on data set P, regardless which subset of P is to be amalgamated. Thus, this operation can be done at the time of building spatial data cubes on P. After such a pre-processing step, the polygons in Figure 2 are represented in the database as

$$p_1 = \langle (a, b), (b, c), (c, e), (e, d), (d, a) \rangle,$$
$$p_2 = \langle (e, c), (c, f), (f, g), (g, h), (h, e) \rangle,$$
$$p_3 = \langle (e, h), (h, i), (i, j), (j, k), (k, d), (d, e) \rangle.$$

2.2 Removing Identical Line Segments

Under the above assumptions, it is clear that there are no identical line segments in a polygon. Further, a line segment l is on the boundary of $t(S)$ if and only if it has no identical line segments in S. Therefore, a straightforward algorithm to amalgamate polygons is to remove all identical line segments in S. Below is a sketch of such an algorithm:

Algorithm SIMPLE
Given a set of source polygons S, find $t(S)$.

1. (*Retrieve data*) fetch all the polygons in S into a set L of line segments, with the middle point of each line segment calculated.
2. (*Remove identical line segments*) sort line segments by their middle points (by x then y), and remove all line segments whose middle points appear more than once in L.
3. (*Finish*) return the remaining line segments in L as $t(S)$.

Since we assume that there are no overlapping polygons or self-intersecting rings in source data, in this algorithm we use the middle point to represent a line segment for identification of identical line segments. It is simpler and more efficient to process points than line segments. An additional advantage of representing a line segment by a point is that it becomes possible to apply a hash function to avoid sorting all line segments together [26]. That is, space $\mathcal{D}$ can be

divided into cells, and a data bucket is associated with each cell. A line segment is mapped into the bucket whose corresponding cell contains the middle point of the line segment. After line segments are mapped into buckets, all identical line segments must be inside the same bucket. Thus, it is sufficient to sort the line segments bucket by bucket, instead of sorting them all together. This hashing-based method is particularly useful when the memory is not large enough for storing all line segments of S as it is now possible to apply those well-known methods developed in relational databases to handle similar problems (such as the hybrid join algorithm [5]).

3 Identifying Boundary Polygons

In this section we investigate two methods of identifying a subset $\overline{S} \subseteq S$ such that $t(\overline{S}) \equiv t(S)$. The performance of these algorithms will be discussed in Section 4.

3.1 Using Adjacency Information

Two identical line segments must come from two adjacent polygons. Strictly speaking, polygons can be adjacent to each other by edge or by point. There is no need to consider the latter because our interest here is to identify identical line segments. Using the data structures discussed in Section 2, one can simply define that two polygons are adjacent to each other if they have at least one pair of identical line segments. Moreover, the *adjacency table* of a set P of polygons is defined as a two column table $\texttt{ADJACENCY(p,p}'\texttt{)}$ where p, p' are identifiers of polygons in P. A tuple (p, p') is in table ADJACENCY if and only if polygon p is adjacent to polygon p'. The adjacency relation is reflective. For two adjacent polygons p and p', one can record the adjacency information redundantly (i.e., recording both (p, p') and (p', p)). Alternatively, a total order among polygon identifiers can be imposed (e.g., the alphabetical order if the identifiers are character strings) such that (p, p') in the adjacency table only if $p < p'$. As to be discussed in Section 4, this decision is an implementation issue, which has implications on the efficiency of the database queries to identify boundary polygons. For presentation simplicity, we assume the redundancy approach hereafter. Below is the adjacency table for the four polygons in Figure 3 (a) where $P = \{p_1, p_2, p_3, p_4\}$:

$$\{(p_1,p_2),(p_1,p_3),(p_1,p_4),(p_2,p_3),(p_2,p_4),(p_3,p_4),(p_2,\mathcal{D}),(p_3,\mathcal{D}),(p_4,\mathcal{D})\}$$

where $(p, \mathcal{D})$ means p has at least one line segment adjacent to no P polygons (in this case we say p is adjacent to a dummy polygon also labeled as $\mathcal{D}$). For $S \subseteq P$, by definition we have

$$\partial S = \{p | p \in S, \exists (p,p') \in \texttt{ADJACENCY}, p' \notin S\}$$

That is, a polygon is on the boundary of $t(S)$ if and only if it is inside S and it has at least one adjacent polygon which is not in S. Using the adjacency table, ∂S can be easily identified using SQL queries.

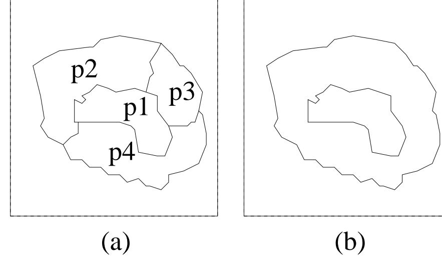


Fig. 3. An example of shadow ring

Unfortunately, it is not sufficient to use only ∂S to produce $t(S)$. When $\partial S \subset S$ (i.e., a true subset), two rings can be produced by removing all identical line segments in ∂S. For example, when merging the four polygons in Figure 3 (a) using $\partial S = \{p_2, p_3, p_4\}$, two rings are produced as shown in Figure 3 (b) where the inner ring is actually not part of $t(S)$. We call such an unwanted ring a *shadow ring*. The reason of producing shadow rings is simple: some line segments which would be otherwise found as identical cannot be recognized because their counterpart line segments are from polygons not in ∂S. In general, as a result of using only a subset of polygons, for each ring r which is part of $t(S)$ (called a *boundary ring*), there may exist a corresponding shadow ring r'. Note that r can either enclose or is enclosed by its shadow ring r' (the latter happens when r defines a hole of the target polygon). A shadow ring can be a boundary ring at the same time (e.g., when $S = \{p_2, p_3, p_4\}$ in Figure 3). The following example illustrates that it is not possible to tell whether a ring is a shadow ring by only looking at ∂S. While the inner ring in Figure 3 (b) is a shadow ring when $S = \{p_1, p_2, p_3, p_4\}$, it defines a hole in $t(S)$ when $S = \{p_2, p_3, p_4\}$. In order to identify possible shadow rings, we need to use a supplementary data set $\partial S^+ \subseteq S$ where ∂S^+ contains those S polygons adjacent to but not in ∂S polygons. That is,

$$\partial S^+ = \{p | p \in (S - \partial S), \exists p' \in \partial S, (p, p') \in \texttt{ADJACENCY}\}$$

We call the polygons in ∂S and ∂S^+ the *boundary polygons* and the *sub-boundary polygons* respectively. For a line segment $l \in \partial S$, if there exists a line segment $l' \in S$ and l' is identical to l, l' must be in either ∂S or ∂S^+. In other words, no ∂S line segments can form a shadow ring if ∂S polygons are processed together with ∂S^+ polygons. Of course, after removing identical line segments in $\partial S \cup \partial S^+$, all line segments from ∂S^+ need to be discarded.

Algorithm ADJACENCY

Given the adjacency table `ADJACENCY` for a set P of polygons and $S \subseteq P$, find $t(S)$.

1. (*Find* ∂S) $\partial S = \emptyset$; for each $p \in S$, add p to ∂S if $(p, p') \in$ `ADJACENCY` and $p' \notin S$.

2. (*Find* ∂S^+) $\partial S^+ = \emptyset$; for each $p \in \partial S$, add $p' \in (S - \partial S)$ to ∂S^+ if $(p, p') \in$ `ADJACENCY`.
3. (*Retrieve data*) retrieve all polygons in $\overline{S} = \partial S \cup \partial S^+$ into a set L of line segments, and mark the line segments from ∂S^+ as 'auxiliary'.
4. (*Remove line segments*) remove identical line segments in L (as in Algorithm SIMPLE).
5. (*Remove* ∂S^+ *line segments*) remove all the 'auxiliary' line segments from L.
6. (*Finish*) return the remaining line segments in L as $t(S)$.

Algorithm ADJACENCY computes $t(S)$ from $\overline{S} = \partial S \cup \partial S^+$, where ∂S and ∂S^+ are found using the adjacency table which contains no spatial data. That is, the internal polygons can be excluded from further processing without fetching and examining their spatial descriptions. It is not necessary to fetch all polygon identifiers for the database, as the adjacency table is a simple relational table and the first two steps in algorithm ADJACENCY can benefit from indices on both columns of the table. Note that the adjacency table only needs to be built once, independent of S. However, ∂S and ∂S^+ have to be built when S is given.

Like the *filter-and-refine* approach which is a standard approach in spatial data processing [3, 11, 26], algorithm ADJACENCY uses the adjacency table to perform a non-spatial filtering step to reduce the number of spatial objects to be processed. Because of the extra cost in constructing ∂S and ∂S^+, Algorithm ADJACENCY is efficient only when $|\overline{S}| << |S|$. As we will show in Section 4, even though $|\partial S^+|$ can be many times larger than $|\partial S|$, this filtering step is still very effective in improving the overall performance.

3.2 Using Occupancy Information

The adjacency-based approach above can be described as object-centric as it focuses on identifying boundary polygons. Now we propose a space-centric approach, which decomposes space $\mathcal{D}$ into small irregular regions and identifies a set of boundary regions.

Z-values All SDBMSs support one or several spatial data access methods for fast retrieval of spatial objects. Spatial access methods have received extensive attention in the past from the spatial database research community (see [10] for a survey). Represented by the spatial indexing mechanisms based on the R-trees [13], R$^+$-trees [24] and the z-values [2], a spatial data access method establishes certain relationship between the data space and spatial objects or their approximations such as minimum bounding rectangles. The z-ordering technique is one of the most widely used spatial indexing mechanisms [2, 20, 23]. It approximates a given object's shape by recursively decomposing the embedding data space into smaller sub-spaces known as *Peano cells*. The z-ordering decomposition works as follows. The whole space $\mathcal{D}$ (represented as a rectangle) is divided into four smaller rectangles of the same size. The four quadrants are numbered as 1 to 4

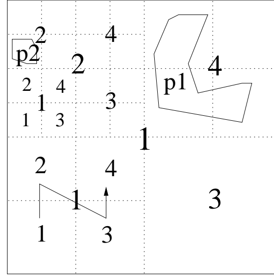


Fig. 4. Z-order and object approximation using z-values

following certain order (e.g., the z-order as shown in Figure 4). These quadrants can be further divided and numbered recursively. In such a way, $\mathcal{D}$ can be decomposed into a set of quadrants, or Peano cells, of varying sizes. Each Peano cell has a *z-value*, which can be determined elegantly by bit-interleaving [23]. The following is a simple description of one way to assign z-values (see Figure 4):

1. The z-value of the initial space, $\mathcal{D}$, is 1;
2. The z-values of the four quadrants of a Peano cell whose z-value is $z = z_1 ... z_n$, $1 \leq z_i \leq 4$, $1 \leq i \leq n$, are $z1$, $z2$, $z3$ and $z4$ respectively following the z-order;

Thus, the z-values of the minimum Peano cells containing polygon p_1 and p_2 in Figure 4 are 14 and 1221 respectively. The z-values can have different length, reflecting different Peano cell sizes. The maximum number of decomposition level, also called resolution, determines the maximum length of z-values. In order to simplify processing, a number of '0's are often appended at the end of shorter z-values to make the length of all z-values identical (i.e., the maximum length).

Spatial objects can be approximated using a set of Peano cells. A polygon can be assigned with the z-value of the minimum Peano cell which fully encloses the polygon (so the z-values of polygons p_1 and p_2 in Figure 4 are 14 and 1221 respectively). The accuracy of approximation can be improved by assigning multiple z-values to a polygon (e.g., p_1 in Figure 4 can be approximated by three Peano cells whose z-values are 141, 142 and 143). The issue of approximating polygons by multiple z-values is a subject of several previous studies [2, 9, 19]. From a mathematical viewpoint, this decomposition is a transformation of a two- (or higher) dimensional object into a set of one-dimensional points (i.e., the z-values) which can be represented as numbers and therefore can be maintained by a ubiquitous one-dimensional access method such as the B$^+$-tree [1].

Let c and c' be two Peano cells. If c is nested inside c' and the z-value of c' has k non-zero digits, then c must have at least k non-zero digits, and they are digit-wise identical to the first k digits of the z-value of c. In other words, the containment relationship among Peano cells can be easily recognized by looking at their z-values. This property has a wide range of applications. For example, for selecting objects inside a given region, one can first find a set Z of z-values of the Peano cells covering the query region. Subsequently, the objects inside

the query region can be quickly identified, using a B^+-tree index on z-values for example, by comparing their z-values with the z-values in Z.

Z-values with Occupancy Let C be the set of all Peano cells with which S polygons overlap with. If $c \in C$ is not *completely* occupied by S polygons, we call c a *boundary cell.* An S polygon overlapping with a boundary cell is likely to be a boundary polygon. Now we look at how to use z-values to find boundary cells by extending the traditional z-value based spatial indices.

The spatial indices using z-values associate objects with Peano cells. That is, each index entry is of the form (z, p), stating that object p overlaps with Peano cell z. There is no information about what percentage of the cell is occupied by p, thus it is not sufficient to determine if a Peano cell is completely occupied by a set of objects. Therefore, we extend the index entry to the form of (z, p, α) where α is the *occupancy ratio.* Let $p \cap z$ be the polygon produced from clipping polygon p by the Peano cell z, then

$$\alpha = \frac{area(p \cap z)}{area(z)}$$

In other words, we record not only which polygons overlap with a Peano cell, but also the percentage of the area that each polygon occupies. The structure of the spatial indexing mechanisms based on traditional z-values, and the algorithms using such indices, need little modification to accommodate this additional piece of data, though some more efficient algorithms can be designed to take advantage of the occupancy information (e.g., the spatial join algorithm in [25]).

Identifying Boundary Peano Cells With the occupancy ratio for a polygon in a Peano cell, a boundary cell with respect to S can be identified simply by adding up the occupancy ratios of all S polygons overlapping with the cell. On the other side, we want to ensure that all S polygons which do not overlap with any boundary cells are internal polygons thus can be ignored by our amalgamation algorithm. While this is in general true, there is an exception when polygon p has a line segment l which coincides with the boundary of a Peano cell. One extreme case is a polygon that is of the same shape and size with a Peano cell. To solve this problem, we introduce zero-occupancy. That is, we use $(p, z, 0)$ if p is adjacent to but not inside cell z. In such a way, if p is a boundary cell but cannot be recognized by other cells, it can be picked up by cell z if z is a boundary cell. Now the problem of finding boundary polygons of S can be solved by finding all Peano cells which are not fully occupied by S polygons.

The algorithm for finding boundary cells can be complex because Peano cells are typically of different sizes, and some Peano cells may be nested inside others. Assume that c is nested inside c'. If c is an internal cell (i.e., fully occupied by S polygons), this fact needs to be propagated to its parent cell c' in order to find if c' is also fully occupied. This upwards propagation can be done using an algorithm of controlled-traversal similar to the one used in [20]. However, if a parent cell c'

is not fully occupied but one of its sub-cell c might be, it is difficult to translate the occupancy ratio from c' to c. This translation requires polygon clipping and re-calculation of the occupancy ratios in c for the polygons approximated in c'. In addition to a more complex algorithm to identify boundary cells, allowing nesting cells may lead to another disadvantage. That is, it is no longer possible to use simple SQL queries to find boundary cells; rather, all spatial index entries need to be pulled out and processed outside of the underlying DBMS.

On the other side, it is not efficient if there are too many polygons associated with a Peano cell. All source polygons associated with a boundary cell will be processed as possible boundary polygons. Thus, a number (termed *high watermark* (HWM)) is chosen such that a cell is to be further decomposed into quadrants when the number of polygons overlapping with the cell exceeding the high watermark. To avoid nesting cells, once a cell is decomposed all polygons approximated in that cell will be re-approximated at the lower level. In other words, while cells in the spatial index can be different sizes, there are no nesting cells.

Algorithm OCCUPANCY
Given a set $S \subseteq P$ of source polygons and a z-value-with-occupancy index $I = \{e_1, \ldots e_n\}$ for P where each e_i is of the form (z, p, α), find $t(S)$.

1. (*Identify boundary cells*) select $Z = \{e_i.z | e_i \in I, e_i.p \in S, \sum_{e \in I, e.z = e_i.z} e.\alpha < 100\%\}$.
2. (*Identify* $\overline{S}$) $\overline{S} = \emptyset$; add p to $\overline{S}$ if $(z, p, \textit{do-not-care}) \in I$ and $z \in Z$.
3. (*Fetch* $\overline{S}$ *polygons and do line-clipping*) fetch polygons whose id in $\overline{S}$, and add the line segments which intersect with at least one cell in Z into L.
4. (*Remove duplicate line segments*) remove identical line segments in L (as in Algorithm SIMPLE).
5. (*Finish*) return L as $t(S)$;

Algorithm OCCUPANCY identifies boundary Peano cells and computes the target polygon using the occupancy information. Note that the first two steps in this algorithm can be implemented using a single SQL query with a group-by clause (i.e., "group by z having $sum(\alpha) < 100\%$"). Because a boundary cell tells not only which polygons to be fetched but also which parts of these polygons are to be used (that is, a line segment l of polygon p which overlaps with a boundary cell z contributes to the part of $t(S)$ in that cell only if l intersects with z). Based on this property, Algorithm OCCUPANCY can discard some line segments in step 2 and subsequently improve the performance of step 3.

In algorithm OCCUPANCY $\overline{S}$ is a superset of ∂S because an internal polygon can also overlap with a boundary cell. In general, it is likely to have more internal polygons in $\overline{S}$ if the Peano cells are large. This performance issue will be discussed in Section 4. Because of these internal polygons, on the other side, algorithm OCCUPANCY does not have the problem of producing shadow rings. One can see this from two aspects:

1. Any line segment which is not in any boundary cell is not part of the target polygon, thus can be discarded;

2. Any line segment inside a boundary cell either is part of the target polygon, or its counterpart from another polygon must also be fetched as this polygon also overlaps with the boundary cell.

Fuzzy Amalgamation Algorithm OCCUPANCY has a desirable advantage over the other two algorithms — the definition of boundary cells (thus boundary polygons) can be easily adjusted by the user. Instead of defining the internal cells as those with 100% aggregate occupancy, one can adjust to a lower *threshold* (for example, 95%). This is useful when the user wants to ignore data noises in polygon amalgamation (e.g., holes smaller than certain size, caused by either some abnormal or insignificant attribute values, or caused by inaccuracy in polygons definitions known as *sliver* polygons). If this threshold percentage is defined relatively to a Peano cell, one can simply replace 100% in algorithm OCCUPANCY with the threshold. In general, however, the user may want to use a threshold related to the data space $\mathcal{D}$. In this case one need to translate this threshold value to each Peano cell by considering actually the size of the cell. Let $d = length(z)$ be the number of non-zero digits of z-value z. The area of this cell is $1/2^d$ of that of $\mathcal{D}$. Thus, a threshold of t percent of the total data space is equivalent to $t \times 2^d$ percent in cell z. Algorithm OCCUPANCY can skip these insignificant holes easily and safely, with little extra overhead. This type of fuzzy amalgamation, often found as useful for spatial OLAP applications, cannot be achieved by the other two algorithms without forming those small polygons and calculating their sizes.

4 Performance Study

In this section we compare the three polygon amalgamation algorithms discussed in this paper. The primary performance index used in this section is the response time, which is measured as the elapsed time from when a predicate describing the source polygons is submitted to all the line segments of the target polygon are found. As pointed out in [10], the I/O cost-based measure such as the number of disk pages accessed is not necessarily a suitable performance indicator because the CPU cost can be equally important. Not measured are those once-off costs, including pre-processing of original polygons (to comply with the polygon data structures in Section 2), building the adjacency and occupancy tables and other necessary indices.

4.1 Cost Analysis

From a set $S \subseteq P$ of source polygons, an amalgamation algorithm takes three phases to produce the target polygon $t(S)$:

1. *the query phase* which identifies a subset $\overline{S} \subseteq S$ (where $\overline{S}$ is represented as a set of polygon ids);

2. *the fetch phase* which retrieves the polygons whose ids are in $\overline{S}$, and unfolds polygons from a sequence of points into an array L of line segments; and
3. *the merger phase* which computes the target polygon by removing duplicate line segments in L.

Let C_{query}, C_{fetch} and C_{merge} be the response times for the three phases respectively. The response time for an amalgamation algorithm, C_{total}, is the sum of these three components. That is:

$$C_{total} = C_{query} + C_{fetch} + C_{merge}$$

Two factors need to be considered for C_{query}. First, these three algorithms find an $\overline{S}$ with different numbers of polygons. A larger $|\overline{S}|$ may affect C_{query} as well as C_{fetch} (since more polygon ids and polygons need to be fetched). Second, these three algorithms use methods of different complexity to select $\overline{S}$. Algorithm SIMPLE simply uses $\overline{S} \equiv S$, thus it uses a straightforward selection query for this step. Algorithm ADJACENCY selects $\overline{S}$ in two steps: one to identify the boundary polygons and one to identify the sub-boundary polygons. Algorithms OCCUPANCY also uses two steps: identification of the boundary cells using a query with an aggregate function, followed by a set of queries to retrieve ids of the source polygons overlapping with the boundary cells. For both algorithms, it is possible to combine these two steps into one SQL query. However, such a complex query is far less efficient to execute than executing two separate queries for the two steps in an application program. For algorithm OCCUPANCY, both the final size of $\overline{S}$ and the cost for identifying $\overline{S}$ vary with HWM.

All the three algorithms retrieve the polygons whose ids are in $\overline{S}$. The cost for fetching objects (i.e., C_{fetch}) depends not only on how many polygons to be fetched but also on the sizes of these polygons. During this phase, algorithm ADJACENCY tags each line segment with its source (i.e., from a boundary or sub-boundary polygon). For algorithm OCCUPANCY, it clips line segments against boundary Peano cells such that only the line segments which intersect with at least one boundary Peano cell are kept for the next phase. With larger Peano cells (because of higher HWM), more internal polygons are included in $\overline{S}$ and less line segments can be dropped out by clipping. The middle point of each line segment is calculated in this phase for all the three algorithms.

The line segments are sorted by their middle points and the line segments that appear more than once are removed. Algorithm ADJACENCY needs to have an additional step to discard all line segments from the sub-boundary polygons. The remaining line segments form the target polygon. We do not include the time to order the line segments for the target polygon in C_{merge} as this cost is identical across all the three algorithms.

4.2 Databases and Parameters

Each polygon is stored as one object in the database (i.e., we do not consider object decomposition), whose schema is:

$$\texttt{POLYGON(pid : INTEGER, boundary : POLYGON)}$$

where data type POLYGON, which is essentially a sequence of points, can have different implementations according to different underlying DBMS. We use Oracle 8 in our tests (its object-relational features are not used, nor is the Spatial Data Cartridge, so POLYGON is simply implemented as a BLOB). Other attributes for a polygon are stored in a separate table and are linked to table `POLYGON` through `pid`.

A subset of the TIGER/LINE data (census blocks in California) is used for our performance testing (see http://www.census.gov/ftp/pub/geo/www/tiger/). A census block has an attribute county id, which is used for grouping source polygons in our experiments. A county consists of from 9 to 6,022 polygons. There are 21,648 polygons with 1,618,950 points in total. The number of points in a polygon ranges from 4 to 3,846, with an average of 75 points. We merge census blocks into counties, and adjacent counties into larger polygons. The preprocessing for splitting line segments at joint points and resolving data inconsistency problems are done using a GIS package ARC/INFO.

The adjacency table has the schema:

$$\texttt{ADJACENCY}(\texttt{pid}:\texttt{INTEGER}, \texttt{next_to}:\texttt{INTEGER})$$

If polygon p and p' are adjacent to each other, we chose to store both (p, p') and (p', p) in table `ADJACENCY`. The number of tuples in the table is twice more than necessary, but the query to identify adjacent polygons is simpler and runs faster. The adjacency table for the data set we used has 137,978 rows. Both the boundary polygons and sub-boundary polygons for algorithm ADJACENCY are identified using this table, in two steps as mentioned before.

The occupancy information is recorded as

$$\texttt{OCCUPANCY}(\texttt{z}:\texttt{INTEGER}, \texttt{pid}:\texttt{INTEGER}, \texttt{occupancy}:\texttt{NUMBER})$$

Since all we need to find here is whether a cell is fully occupied by a set of polygons or not, we do not compute exact percentage of a polygon in a cell. Instead, we count the number of polygons in each cell and simply give each polygon an equal share of occupancy. For example, if there 8 polygons in a cell, the occupancy for each polygon in that cell is $1/8 = 0.125$, regardless their actual occupancy ratios. When the total occupancy for a given set of source polygons in the cell is 1, we know that the cell is not a boundary cell. (However, an accurate calculation of occupancy ratio is necessary in order to support fuzzy amalgamation.) The number of rows in table `OCCUPANCY`, as shown in Table 1, varies depending on HWM.

4.3 Experimental Results

Now we compare the performance of the three amalgamation algorithms empirically. For algorithm OCCUPANY, we also test with different HWMs. The algorithms are implemented using Microsoft Visual C++ and Oracle OCI interfaces. Both development and testing are done using a DELL notebook (Pentium II/266) with 128 MB memory. Indexes are created wherever necessary for all the

$HWM(bytes)$	Average num. of polygons per cell	Num. of rows in table OCCUPANCY
512	4.6	99638
1024	7.1	65405
2048	11.4	48168
4096	18.9	38336

Table 1. High water marks (HWM) for the occupancy table.

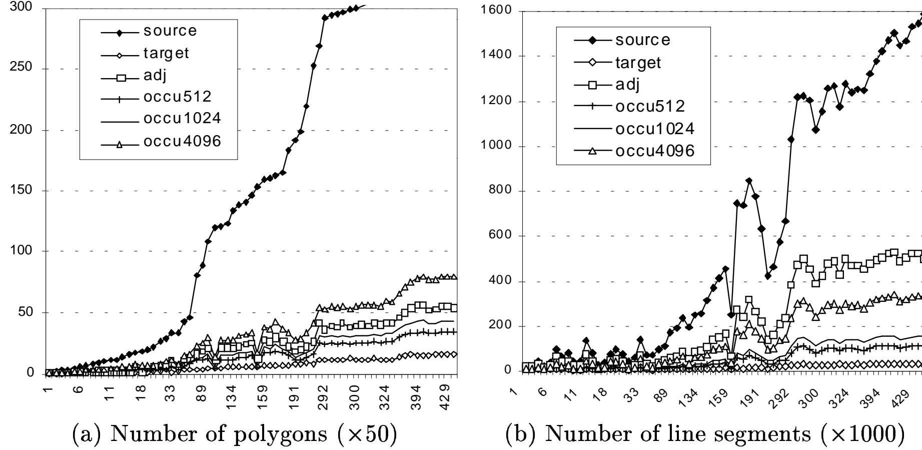


Fig. 5. Number of spatial objects processed (x-axis = $|S|/50$)

tables used. Oracle's array fetch function is used whenever possible to improve the performance of data retrieval.

Data Size Reduction First, we look at the effectiveness for these algorithms in reducing the number of polygons to be processed. Figure 5 (a) shows the number of polygons actually fetched by different algorithms, where the x-axis in Figure 5 (and Figure 6) is the number of source polygons to be amalgamated (ranging from 17 to 21,528). Obviously, the maximum and minimal numbers of polygons to be fetched by any amalgamation algorithm are $|S|$ (i.e., all source polygons) and $|\partial S|$ (i.e., only the boundary polygons) respectively. These two numbers are labeled as 'source' and 'target' respectively in Figure 5.

Figure 5 (a) reveals three facts: (1) in comparison with algorithm SIMPLE which retrieves all the source polygons, both adjacency-based and occupancy-based algorithms fetch a much smaller number of polygons; (2) the performance of algorithms ADJACENCY and OCCUPANCY scales well when the number of source polygons increases (note that the number of polygons to be fetched depends not only on the number of source polygons but also the complexity of target polygon such as its shape and if there exist holes or not); and (3)

the differences among the adjacency-based algorithm and the occupancy-based algorithms with different HWMs are significant (the differences among them may look deceptive in Figure 5 (a) due to much bigger differences between them and algorithm SIMPLE). The occupancy algorithm with HWM = 4096 performs consistently worse than the adjacency algorithm, which needs to retrieve about 50% more polygons than the occupancy algorithm with HWM=512. For the occupancy algorithm, when HWM increases, more polygons overlap with a Peano cell; thus, it is more likely to fetch internal polygons. The implication of different HWMs on response times will be examined later.

For algorithm ADJACENCY, $|\partial S^+|$ can be derived as the difference between the actual number of polygons fetched by the algorithm and $|\partial S|$. One can see that ∂S^+ contains about twice more polygons than ∂S.

Figure 5 (b) shows the total number of lines to be processed. Here the maximum and minimum numbers of line segments to be processed by any amalgamation algorithm are the total number of line segments in all source polygons (i.e., $||S||$) which is what algorithm SIMPLE has to process, and the number of line segments in the final target polygon (i.e., $||t(S)||$). The occupancy algorithm discards those line segments not overlapping with any boundary Peano cells. As a result, the number of line segments processed by the algorithm (after clipping) is much smaller than that by the adjacency algorithm, which in turn processes mush less line segments than the simple algorithm.

Response Times Figure 6 shows the response time for each phase as well as the total elapsed time. The query time is the elapsed time from when the predicate describing source polygons is submitted to when the polygons to be fetched are identified. Three factors may affect the query time. First, the size of the table used to produce $|\overline{S}|$ (i.e., the size of the adjacency and occupancy tables for algorithms ADJACENCY and OCCUPANCY respectively). Second, $|\overline{S}|$ itself. Third, the complexity of the query used for identifying candidate polygons. We avoid to use inefficient join query for the adjacency and occupancy algorithms (using two-passes as mentioned in section 4.1); thus the complexity of the queries used for these three algorithms are similar. Figure 6 (a) illustrates clearly that the response time for the query phase is primarily determined by the first factor. $|\overline{S}|$ is insignificant because of the use of array fetch. Algorithm SIMPLE is the fastest since it uses only a simple selection query on the base table that has a smaller number of rows than the adjacency or occupancy tables. The adjacency algorithm is the slowest, even when it results in smaller $|\overline{S}|$ in comparison with the occupancy algorithm with HWM=4096. For the occupancy algorithm, a smaller HWM results in a larger occupancy table (see Table 1) due to a higher probability for one polygon overlapping with many cells [19].

The time for fetching polygons from the database, as shown in Figure 6 (b), clearly dominates the whole polygon amalgamation process. It is our main motivation in this paper to reduce this time. We have achieved this goal by reducing this time by approximately 80% in comparison with algorithm SIMPLE.

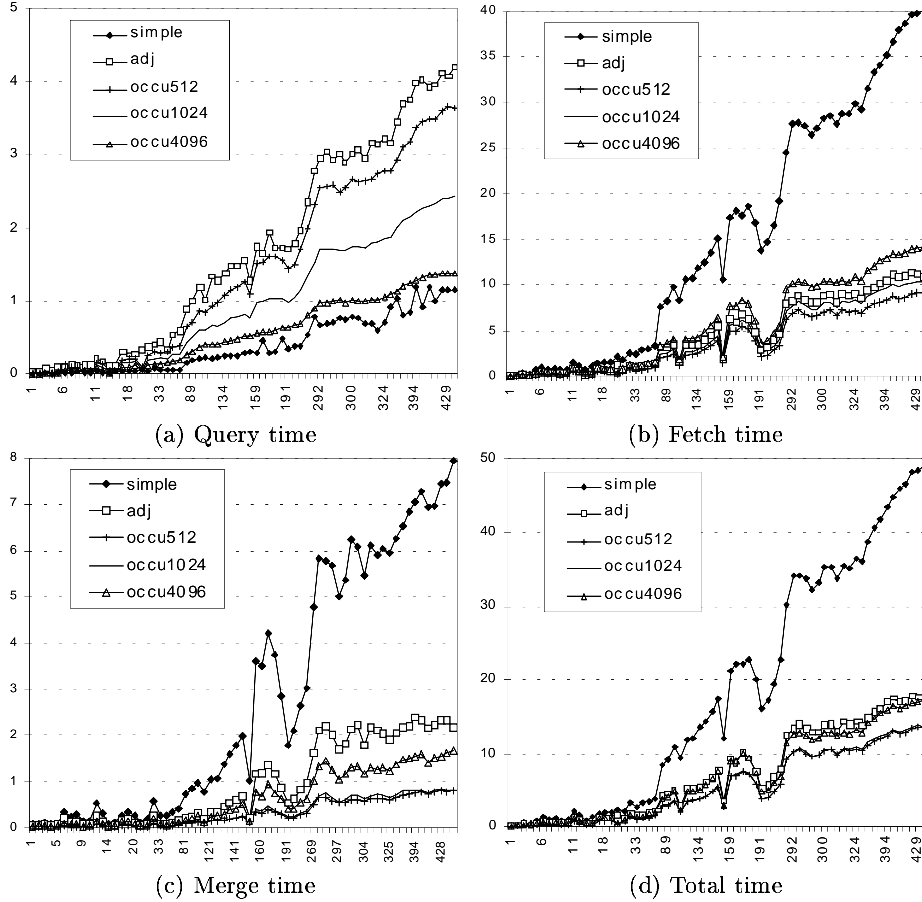


Fig. 6. Response time (seconds) (x-axis = $|S|/50$)

A clear winner is the occupancy algorithm with HWM=512, which is the most selective one as shown in Figure 5 (a).

The biggest time reduction in terms of percentage has been achieved for the merger phase. The cost of this phase, obviously, is determined by the number of lines to be processed (compare Figure 6 (c) with Figure 5 (b)). The occupancy algorithm, with all three HWMs, performs significantly better because of a smaller $|\overline{S}|$, and more importantly, because of discarding line segments by boundary cells at the end of the fetch step.

Figure 6 (d) shows the total response time. It is clear that the occupancy algorithm has achieved a remarkably better overall performance, in particular with an HWM which puts an average of 5 - 7 polygons to a Peano cell. That is, HWM = 512 or 1024 for the data set we use. A higher HWM degrades the performance because it becomes inefficient in filtering out internal polygons and

line segments. On the other side, a too low HWM (i.e., less than 512 for the data set we used) increases the query cost with little benefit to other steps.

Finally, we briefly discuss the memory requirement for the amalgamation algorithms. Algorithm SIMPLE is the hungriest in terms of memory requirement among the three algorithms, as it needs to hold all line segments in the memory. Algorithm ADJACENCY consumes much less memory, only for holding line segments from the boundary and sub-boundary polygons. Algorithm OCCUPANCY needs the least amount of memory, as it stores only part of line segments for the polygons overlapping with boundary Peano cells. We assume in this paper that the memory is large enough for holding all line segments to be processed in memory. This assumption might become unrealistic when there are a large number of polygons to be processed (which is common in spatial OLAP and spatial data mining applications). However, based on the fact that only the polygons adjacent to each other are to be processed together for the purpose of removing duplicate line segments, those algorithms in spatial databases (such as the plane-sweep algorithm [3, 22]) or in relational databases to handle the similar problems (such as the hybrid hashing [5]) can be used when the memory is not big enough.

5 Conclusions

With emerging new applications such as spatial OLAP and spatial data mining, certain spatial operations such as polygon amalgamation have become increasingly popular and its efficient implementation becomes crucial in the realization of new spatial applications. In this paper, we have studied efficient algorithms for polygon amalgamation. This operation is intrinsically time-consuming. However, with the observation that only boundary polygons are playing crucial roles in polygon amalgamation, a set of interesting algorithms have been proposed and studied in this paper. Starting from improving a simplistic polygon amalgamation algorithm, we have proposed two methods, adjacency-based and occupancy-based, which exclude a large subset of polygons from being considered in the amalgamation algorithm without retrieving the spatial description of these polygons. The performances of these algorithms have been compared using real spatial data sets. With the support of a more sophisticated data storage structure, the occupancy-based method outperforms the adjacency-based method, whereas both methods are significantly more efficient than the algorithm which requires to fetch all objects to be merged.

The performance of the occupancy-based algorithm can be further improved by decomposing spatial objects. As implemented in some SDBMSs, a spatial object can be decomposed with the Peano cells approximating the object. In such a case, the occupancy-based algorithm only needs to fetch the parts of a spatial object that are inside a boundary cell, instead of the whole object. Such object decomposition can be done off-line when a spatial index is built. The online processing performance for the occupancy-based algorithm can be improved greatly as the amount of data to be retrieved is reduced and there is no need to

do polygon clipping on-the-fly. Our work on this improvement will be reported in a separate paper. In the future we also plan to integrate our new polygon amalgamation algorithms with the research results in selective materialization for data cube construction [15] for supporting spatial OLAP and spatial data mining applications.
Acknowledgment: The first author conducted part of this research while in CSIRO. The work of the third author was partially supported by Natural Science and Engineering Research Council (NSERC) of Canada and Networks of Centres of Excellence (NCE) of Canada. The authors would like to thank Dr. Dave Abel, Dr. Volker Gaede and Professor Maria Orlowska for many helpful discussions.

References

1. D. J. Abel. SIRO-DBMS: A database toolkit for geographical information systems. *Int. J. Geographical Information Systems*, 4(3):443 – 464, 1989.
2. D. J. Abel and J. L. Smith. A data structure and algorithm based on a linear key for a rectangle retrieval problem. *Computer Vision, Graphics and Image Processing*, 24(1):1–13, 1983.
3. T. Brinkhoff, H. P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 237–246, Washington, D. C., 1993.
4. S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *SIGMOD Record*, 26(1), 1997.
5. D. J. DeWitt and R. H. Gerber. Multiprocessor hash-based join algorithms. In *Proc. 1985 Int. Conf. on Very Large Data Bases*, pages 151–164, Austin, Texas, 1985.
6. M. J. Egenhofer. Reasoning about binary topological relationships. In *LNCS 552: Proceedings of 2nd Int. Symp. on Spatial Databases (SSD'91)*, pages 143–160. Springer-Verlag, 1991.
7. M. Ester, H.-P. Kriegel, and J. Sander. Spatial data mining: A database approach. In *LNCS 1262: Proceedings of the 5th Int. Symp. on Spatial Databases (SSD'97)*, pages 47–66, Berlin, Germany, 1997. Springer-Verlag.
8. M. Ester, H.-P. Kriegel, J. Sander, and X. XU. Density-connected sets and their application for trend detection in spatial databases. In *Proc. 3rd Int. Conf. Knowledge Discovery and Data Mining (KDD'97)*, pages 10–15, Newport Beach, California, 1997.
9. V. Gaede. Optimal redundancy in spatial database systems. In M. J. Egenhofer and J. R. Herring, editors, *LNCS 951: Proc. 4th Int. Symp. on Spatial Databases (SSD'95)*, pages 96–116, Portland, Maine, 1995. Springer-Verlag.
10. V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.
11. O. Günther. Efficient computation of spatial joins. In *Proceedings of 9th International Conference on Data Engineering*, pages 50–59, Vienna, Austria, 1993.
12. R. H. Güting. An introduction to spatial database systems. *VLDB Journal*, 3(4):357–399, 1994.
13. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 47–54, Boston, Massachusetts, 1984.

14. J. Han, K. Koperski, and N. Stefanovic. GeoMiner: A system prototype for spatial data mining. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 560–563, Tucson, Arizona, 1997.
15. J. Han, N. Stefanovic, and K. Koperski. Selective materialization: An efficient method for spatial data cube construction. In *Proc. Pacific-Asia Conf. on Knowledge Discovery and Data Mining*, pages 144–158, Melbourne, Australia, 1998.
16. V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data*, pages 206–216, Montreal, Canada, 1996.
17. W. Lu, J. Han, and B. C. Ooi. Knowledge discovery in large spatial databases. In *Proc. Far East Workshop Geographic Information Systems*, pages 275–289, Singapore, 1993.
18. R. Ng and J. Han. Efficient and effective clustering method for spatial data mining. In *Proceedings of 1994 International Conference on Very Large Data Bases*, pages 144–155, Santiago, Chile, 1994.
19. J. Orenstein. Redundancy in spatial databases. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 294–305, Portland, Oregon, 1989.
20. J. Orenstein and F. A. Manola. Probe spatial data modeling and query processing in an image database application. *IEEE Trans. on Software Eng.*, 14(5):611–629, 1988.
21. J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 259 – 270, Montreal, Canada, 1996.
22. F. P. Preparata and M. I. Shamos. *Computational Geometry: an introduction.* Springer-Verlag, 1985.
23. H. Samet. *Applications of Spatial Data Structures.* Addison-Wesley, 1990.
24. T. Sellis, N. Roussopoulos, and C. Faloutsos. The R$^+$-tree: a dynamic index for multi-dimensional objects. In *Proc. 13th Int. Conf. on Very Large Data Bases*, pages 3–11, Brighton, England, 1987.
25. D. Truffet and M. E. Orlowska. Two phase query processing with fuzzy approximations. In *Proc. 4th ACM International Workshop on Advances in Geographic Information Systems (ACM-GIS'96)*, Rockville, USA, 1996.
26. X. Zhou, D. J. Abel, and D. Truffet. Data partitioning for parallel spatial join processing. In *LNCS 1262: Proc. 5th Int. Symp. on Spatial Databases (SSD'97)*, pages 178–196, Berlin, Germany, 1997. Springer-Verlag.