Spatial Query Processing in an Object-Oriented Database System

Jack A. Orenstein

Computer Corporation of America

Abstract

DBMSs must offer spatial query processing capabilities to meet the needs of applications such as cartography, geographic information processing and CAD. Many data structures and algorithms that process grid representations of spatial data have appeared in the literature. We unify much of this work by identifying common principles and distilling them into a small set of constructs. (Published data structures and algorithms can be derived as special cases.) We show how these constructs can be supported with only minor modifications to current DBMS implementations. The ideas are demonstrated in the context of the range query problem. Analytical and experimental evidence indicates that performance of the derived solution is very good (e.g., comparable to performance of the kd tree.)

1. Introduction

It is widely recognized that existing database management systems (DBMSs) do not address the needs of many non-traditional applications such as automated cartography, geographic information processing and computer-aided design. The underlying data models, query languages and access paths were designed to deal with simple datatypes such as integers and strings, while these applications are characterized by spatial data, temporal data and other forms of data with complex structure The subject of this paper is the handling of spatial data in a database management system.

This work was supported by the Defense Advanced Research Projects Agency and by the Space and Naval Warfare Systems Command under Contract No N00039-85-C-0263. The views and conclusions contained in this paper are those of the author and do not necessarily represent the official policies of the Defense Advanced Research Projects Agency, the Space and Naval Warfare Systems Command, or the U.S. Government.

There have been many attempts at combining DBMS capabilities and spatial processing capabilities. The simplest approach is to build on top of an existing DBMS. In [CHOC84, LIEN77, SMIT84] a grid representation of geographic data is used. Cells of the grid are represented by database records. Additional operators (i.e., not usually found in query languages) are added to provide basic spatial capabilities such as the computation of distance [CHOC84]. [CHAN77] describes a system that provides both cell and vector representations. In [CHAF81] relations are used to store images using a vector representation. A QBE-like language handles non-spatial parts of queries while a set of built-in spatial operators handles the spatial parts of queries. This system, unlike the others, has quite sophisticated spatial searching capabilities.

A more general approach is to provide a DBMS that can be customized by the addition of abstract datatypes (ADTs). Extensions to System R provide a "long" field that can be used to store an arbitrary amount of uninterpreted data. This provides the foundation for an ADT facility [HASK82, LORI83]. INGRES has also been extended to allow the addition of ADTs [STON83]. In addition, support for new access methods is provided [STON85] (e.g , access methods for spatial searching can be added).

Neither of these approaches is satisfactory. Assumptions that limit generality (e.g., assumptions about dimensionality of the data, the objects that can be described, and the operations that can be performed) are hard-wired into systems built on top of DBMSs. On the other hand, the ADT approach described above gives very little support to users that need to manipulate spatial data. Both approaches avoid the central issue: What can the DBMS do to aid in the processing of spatial data? Can the DBMS do more than just deliver spatial data to the spatial operators? Adding a fixed set of spatial capabilities (whether by hard-wiring or through an ADT mechanism) leaves the *semantics* of spatial query processing outside the DBMS.

The goal of the PROBE research project is to investigate data modeling, architecture, query processing and optimization issues in the construction of an "extensible" DBMS. This was motivated by the needs of non-traditional applications where spatial and temporal data are prevalent. (For an overview of the PROBE project see [DAYA85].) Our approach to spatial data is as follows. The DBMS should be concerned with manipulating collections of objects The detailed manipulations of individual spatial objects should be left to "specialized processors" encapsulated in object classes (i.e. ADTs). This requires optimizations of set-at-a-time operators to be done by the DBMS.

One object class to be built in, as part of the PROBE research project, will support spatial query processing. To allow wide applicability, the object class cannot be tailored to a particular dimension or representation. Our intention is not to eliminate the need for specialized processors, but to simplify them (by not requiring them to do set-at-a-time processing), and to reduce the amount of work that gets passed to them.

Section 2 describes the PROBE approach to spatial query processing and surveys the relevant literature. Section 3 describes the basic ideas behind our approach and in Section 4 it is shown that, with very minor modifications, existing DBMS implementations can be adapted to support spatial query processing as developed here. In Section 5, space and time requirements are examined analytically and experimentally. Section 6 describes related PROBE work on spatial query processing that is beyond the scope of this paper.

2. Spatial Query Processing in PROBE

We will support spatial query processing by providing an object class that implements *approximate geometry* (AG). The query processor will invoke operators of this object class to obtain fast but approximate answers to spatial queries. If more precise answers are needed, the approximate answers will be refined by the specialized processors.

Approximate geometry, as developed here, relies (conceptually) on a grid representation of spatial data. This is where the approximation comes in - precision is limited by the resolution of the grid. The AG *algorithms* are exact. I.e., if the grid representation is considered to be precise (as might be the case for LANDSAT data), then AG would provide precise results.

It is not feasible to store high-resolution grids explicitly. The space and time requirements are too high. Therefore data structures and algorithms that optimize the processing of grids are important. Work along these lines has been conducted by researchers in 1) databases and 2) image processing and vision (IPV). This work is surveyed below.

The work presented here has three new contributions:

1. **Unification of published results.** A large number of published data structures and algorithms (from database and IPV research) are special cases of the AG techniques described here. As an example of this we will focus on the range search problem (which is described below). However, AG also supports other kinds of searches and other spatial operations (see Section 6). We have developed new algorithms and re-explained published algorithms.
2. **Integration with DBMSs.** We show how AG can be supported with very minor modifications of current DBMS implementations. Common techniques for file organization (e.g. B-trees) and buffer management provide excellent support for AG. We are not aware of any other approach to spatial data that can make this claim.
3. **Analysis of performance.** We have carried out mathematical and experimental analyses of a range query algorithm that can be derived from AG. The analyses show that performance is comparable to that of other practical solutions (e.g. the kd tree [BENT75]).

Recent work from database researchers on the processing of "range queries" resembles techniques being used in IPV. This has occurred because the range query problem can be transformed into a spatial searching problem. Given a set of tuples with k attributes $(A_1, \dots, A_k)$, a range query asks for all tuples such that $L_i \leq A_i \leq U_i$, $i = 1, \dots, k$. I.e., a range, defined by L_i and U_i, is specified for each attribute, A_i. This is a very broad and useful class of queries.

If each A_i is an integer, then a tuple can be viewed as a point in k-dimensional (kd) space or as a pixel in a k-dimensional grid. Each attribute forms one axis of the space. A "black" point or pixel represents a tuple. A "white" point or pixel indicates the absence of a tuple. Usually when we talk about points or pixels, we mean the black ones representing tuples. In this view, a range query is a k-dimensional box in the space (whose sides are parallel to the axes). The range query problem is now a spatial searching problem: Find all the (black) points in a given box (see Figure 1).

In order to support range queries efficiently in a DBMS, three issues must be considered:

1. How should the (black) points be allocated to partitions of the kd space? Each partition will be stored on one disk page. The partitioning should preserve proximity. As the distance between two points decreases, the chance that they will be retrieved together in response to a range query increases. By preserving proximity, the number of page accesses will be minimized.
2. What partitions (i.e. pages) need to be retrieved in response to a range query? How can they be located? A partition must be retrieved if it overlaps the box representing the query. The data structure used for locating partitions is a "spatial index" or a "partition index".
3. How are insertions and deletions handled? The partitioning and the partition index should adapt gracefully as the number and distribution of points change.

A "multidimensional" data structure for searching (MDS) solves the first two problems. A solution to the third problem is desirable but not always necessary.

Most of the work in this field has been done in the past ten years (see [BENT79, OREN83, OREN85] for surveys.) The partitioning of a kd space or grid provided by an MDS is usually grid-like or in a "brick-wall" pattern. Grid methods [MERR78, MERR82, MERR84, NIEV84, TAMM81, TAMM82] construct a grid out of (k-1)-dimensional partitions. The brick-wall pattern is created by splitting in one direction first. These partitions are further partitioned by splits in another direction, etc. This pattern is used in [LIOU77, SCHE82, ROBI81]. The kd tree [BENT75] partitions the space with 2^k "bricks" and then, as necessary, partitions each sub-region recursively. The work presented here is based on a particularly simple splitting scheme. This scheme (or a minor variation of it) has been used in the data structures of [BURK83, OREN82, OREN84, OUKS83, TAMM81, TAMM82]. See [OREN85] for a more thorough discussion of the similarities and differences among MDSs.

Relevant work from the IPV literature is related to the "simple" partitioning methods mentioned above (see [SAME85a] for a thorough survey). A discussion of the relationships between these methods and the quadtree from IPV literature appears in [OREN85]

3. Approximate Geometry

Our approach to AG is based on a grid representation of spatial objects. The techniques to be presented can be thought of as methods that optimize the handling of these grids.

3.1 Elements

A k-dimensional spatial object is approximated by superimposing a kd grid of pixels and noting which pixels lie inside or on the boundary of the object. In what follows, when we discuss spaces and spatial objects, we will be referring to grids and approximations of spatial objects respectively. While the presentation is in terms of 2d data, all the ideas extend to higher dimensions (and to 1d) without difficulty. We will call the horizontal and vertical axes of the 2d space x and y respectively. We assume 1) that the grid has resolution $2^d \times 2^d$ where d is an integer; 2) that regions are split into equal-sized subregions; 3) that the direction of splitting alternates between x and y (as explained below).

An understanding of the partitions obtained by this splitting policy is necessary for the development of AG. The rest of this section will discuss these partitions in some detail.

We will use the following notation: $<s_1{:}n_1 \mid s_2{:}n_2 \mid \ldots \mid s_m{:}n_m>$ denotes the string

$$\underbrace{s_1\, s_1 \ldots s_1}_{n_1}\ \underbrace{s_2\, s_2 \ldots s_2}_{n_2} \ldots \underbrace{s_m\, s_m \ldots s_m}_{n_m}$$

If $n_i = 1$ then $s_i{:}n_i$ may be written as s_i. E.g. $<011{:}2 \mid 01> = 01101101$.

A pixel in the grid is specified by providing two coordinates of d bits each, $(<x_0 \mid x_1 \mid \ldots \mid x_{d-1}>, <y_0 \mid y_1 \mid \ldots \mid y_{d-1}>)$.

A vertical split through the middle of this space amounts to discriminating on the value of x_0. In the left half of the space $x_0 = 0$, in the right half $x_0 = 1$ If the resulting subregions are split horizontally, this corresponds to discrimination on the value of y_0. There are now four regions corresponding to the possible values of x_0 and y_0. In general, each split is characterized by one bit from x or y. The interleaving of these bits from x and y creates a bitstring that uniquely identifies a region. If r is a region created in the splitting process then z(r) denotes the bitstring corresponding to the region (see figure 2.)

The z value of a region is a concise description of the shape, size and position of the region. These can be derived from the z value. In general, if the z value contains the first m bits of x and the first n bits of y, then the region described extends from $<x_0 \mid \ldots \mid x_{m-1} \mid 0{:}d-m>$ to $<x_0 \mid \ldots \mid x_{m-1} \mid 1{:}d-m>$ horizontally, and from $<y_0 \mid \ldots \mid y_{n-1} \mid 0{:}d-n>$ to $<y_0 \mid \ldots \mid y_{n-1} \mid 1{:}d-n>$ vertically. All points inside the region have coordinates with the same m and n bit prefixes. Furthermore, for any region, r, obtained by recursive splitting, *the z value of any point in r is lexicographically between the z values of r's lower left and upper right corners* (see figure 3) I.e., the z values of a region (obtained by recursive splitting) are consecutive.

Although the preceding discussion applies to all regions generated during splitting, it is only the "bottom-most" regions, (those that are not split further), that would be kept in practice. These regions will be called *elements*. Figure 2 shows the elements generated in the "decomposition" of a box. The decomposition algorithm for boxes is given in [OREN84] (the first RangeSearch algorithm). It generalizes immediately to an algorithm for the decomposition of arbitrary spatial objects. All that is required is a procedure that indicates whether a given element is inside a given spatial object, outside the object, or crosses the boundary of the object.

3.2 Z order

Z values can be compared lexicographically, i.e. the bitstrings are left-justified and then compared one bit at a time. Therefore a collection of elements can be ordered by sorting lexicographically on their z values The spatial interpretation of this ordering is interesting. If each pixel of a 2d grid is treated as an element, then the z values of the elements trace out the path shown in figure 4. This ordering has been discovered many times [ABEL83, BURK83, GARG82, OREN84, OUKS83]. When we discovered it we called it z ordering [OREN83, OREN84] and we'll use that name here. The curve is recursive in that it consists of the same "N" shape (covering four points) repeated throughout the space. Groups of four Ns are connected in an N pattern; groups of four of these groups of four are connected in the same pattern, etc. (It's called z order because our first drawing of it had Zs instead of Ns.)

Z ordering is a total ordering of elements. This property, combined with the highly constrained splitting policy that produces elements, leads to the following observation. *The only possible relationships between elements are containment and precedence (in z order)* Overlap (other than containment) cannot occur. This leads to very simple algorithms based on the merging of sequences of elements.

Much of the usefulness of z order is based on the property that it preserves proximity, i.e. if two points are close in space then they are likely to be close in z order. By preserving proximity, elements are clustered for efficient access on secondary storage. (This point is discussed in Section 5.)

3.3 A Solution to the Range Search Problem

To complete this section, the ideas developed above will be used to derive a solution to the range query problem: *Given a set of points, find all points that fall in a given box*

The algorithm is based on the following idea. The box representing the range query can be decomposed into elements. An element represents a set of consecutive z values; i.e. a range of z values. Therefore, if the z value of a point falls in the range of z values of one of the box's elements, the point must satisfy the range query. The algorithm consists of three steps:

1. Compute the z value of each (black) point (by interleaving all bits of the point's coordinates). Form a sequence of points ordered by z value Call this sequence P. Conceptually, a member of P is a record of the form [z, pt] where pt is a description of the point (e.g. the identifier). This is a preprocessing step. Once completed, queries would be run starting at step 2.
2. Decompose the box to yield a set of elements. Compute the z value of each element. Form a sequence of elements ordered by z value. Call this sequence B. Conceptually, a member of B is a record of the form [zlo, zhi]. Recall that each element corresponds to a range of z values; zlo and zhi are the extreme values in this range.
3. Perform a merge of sequences P and B identifying pairs p and b such that $b.zlo \leq p.z \leq b.zhi$. Such a pair represents the fact that the point pt(p) satisfies the range query because it falls inside an element (b) of the box.

This algorithm is demonstrated in Figure 5. Note that the running time is O(length(P) + length(B)) because of the merge in step 3. However, the merge can be optimized in the following way. When $p.z > b.zhi$, instead of scanning through members of B until $p.z < b.zlo$ or $b.zlo \leq p.z \leq b.zhi$, the value of p.z can be used in a random access to B to locate the next "interesting" B record. Similarly, b.lo can be used in a random access to P. I.e., parts of the space that could not possibly contribute to the result are skipped

Note that we have said nothing about data structures so far. The algorithm was expressed in terms of the merging of sequences. Clearly, sequential access is needed so that the sequences can be merged. The optimization described makes use of random access Therefore, any data structure that supports both random and sequential accessing can be used to support this range search algorithm. This is a very modest requirement since Btrees, ISAM, VSAM, etc. are widely available. In spite of the simplicity of this approach, good performance can be obtained. Analytical and experimental results are described in Section 5.

Another optimization is that the sequence B does not have to be formed before the merge starts. Elements of the box may be generated on demand, i.e. when a sequential or random access on sequence B is performed. The method for doing this appears in [OREN84].

With these optimizations, the algorithm is essentially the same as the algorithm reported in [OREN84]. In practice the running time of this algorithm appears to be proportional to the fraction of the space covered by the query (see Section 5.3).

Throughout this section, 2d data has been discussed. Algorithms based on z order work without modification in all dimensions. This is because of the reduction to 1d.

4. Integration with a DBMS

We will now show how the ideas of the previous section can be encapsulated by a join-like operator. We will also show how this operator can be supported with trivial modifications of existing DBMS implementations. The operator, *spatial join*, supports the following very general class of queries. Given two relations, R and S, each storing a set of spatial objects (i.e., each tuple stores the identifier of one object), spatial join identifies overlapping objects from R and S. (A range query is a special case in which one of the relations represents the set of points and the other relation represents the query region.) The spatial join is denoted by

$$R\ [zr \diamond zs]\ S$$

where zr and zs are the attributes of R and S (respectively) that store the elements resulting from the decomposition of spatial objects.

Spatial join can be built into a relational DBMS implementation with very little extra machinery. One obvious addition is a domain for the "element" object class. Recall that an element is just a variable-length bitstring (that has a spatial interpretation). The following operations on elements are needed:

- shuffle(r: region) → element
  This computes the z value for a region obtained by recursive splitting.
- unshuffle(e: element) → region
  Inverse of shuffle.
- decompose(b: box) → set of elements
- precedes(e1, e2: element) → boolean
  Check for precedence in z order.
- contains(e1, e2: element) → boolean
  Check if e1 contains e2.

These are all very simple to implement. The first three are discussed in [OREN84]. If z1 and z2 are the z values of e1 and e2 respectively, then e1 precedes e2 if z1 precedes z2 lexicographically, and e1 contains e2 if z1 is a prefix of z2. (Recall that z values are bitstrings.)

The only other thing needed is an implementation of spatial join. The implementation strategies of natural join can be used. Instead of looking for equality, we're looking for containment between zr and zs. I.e., we want pairs of tuples from R and S such that contains(zr,zs) or contains(zs,zr).

Implementations of spatial join that incorporate the optimizations discussed above will be designed in the next phase of PROBE research. However, it is already clear that existing DBMS facilities provide what is needed in the way of file organizations and buffer management. Recall that the merge step (as in section 3.3) can use any data structure that supports random and sequential access. Many existing DBMS implementations already use B-trees. Z values can easily be represented as integers. Then the < predicate of any programming language can be used to test precedence in z order, so existing sort utilities can be used to create z ordered sequences. The LRU buffering strategy will work well because of our reliance on merging in AG algorithms: each page is accessed at most once, its contents are processed, and then the page will not be needed again for the rest of the merge.

Spatial join would normally be used as in the following scenario. Spatial objects (stored in relations) are decomposed. The resulting relations have attributes for the element and for the identifier (indicated by @) of the object that generated the element.

R(p@, zr, ...) := Decompose(P(p@, ...))
S(q@, zs, ...) := Decompose(Q(q@, ...))

(This notation hides some details. Decompose would have to be applied to each spatial object; i.e., to each tuple of P and Q. Each decomposition would yield a set of elements. Thus the result is a set of sets that must be "flattened" to yield the 1NF relations R and S.)

Next, the spatial join is done.

RS(p@, q@, zr, zs, ..., ...) := R [zr ◇ zs] S

Now, if (p@', q@', zr', zs', ..., ...) is a tuple of RS, then p@' and q@' overlap. The overlap of p@' and q@' may be noted many times. Projecting out the zr and zs fields eliminates this redundancy.

Result := RS[p@, q@, ..., ...]

If p@" and q@" do not occur together in a tuple of RS then they do not overlap.

This strategy leads to the following implementation of range search.

Given Points(p@, x, y) and Box(b@, xlo, xhi, ylo, yhi). Box contains one tuple representing the query region.

P(p@, zp, x, y) := Points[p@, shuffle([x:x, y:y]), x, y)]
Shuffle takes the X range and Y range of an element and produces the z value for the element. Here, the element contains a single pixel, so the X and Y ranges each contain a single value (x and y respectively).

B(zb) := Decompose(Box)
Create the set of elements representing the box.

Result := (P [zp ◇ zb] B) [x,y]
Do the spatial join and project onto the point coordinates.

5. Analysis of the Range Search Algorithm

The objects manipulated by AG algorithms are the elements generated by the decomposition of spatial objects. The time and space requirements of AG algorithms obviously depend on the number of elements generated in a decomposition. Space requirements are discussed in Section 5.1. The preservation of proximity is an important consideration when the elements are kept on secondary storage. Section 5.2 discusses the relationship between preservation of proximity and the time requirements of the range search algorithm. Only results are given here. Derivations and proofs can be found in [OREN83]. An intermediate level of presentation can be found in [OREN85] Section 5.3 summarizes experimental results on the range search algorithm. A detailed description of the experiments and results is in [OREN85].

5.1 Space Requirements

In [OREN83] we analyzed the decomposition of a 2d rectangle of size U x V whose lower left corner is at (0,0). We wanted to understand how the number of elements in the decomposition depends on U and V. (The fact that elements are of different sizes is irrelevant. The time to process one element is independent of the size of the element.) The following facts were derived: (E(U,V) is the number of elements generated in the decomposition.)

- E(U,V) is highly dependent on the number of bit positions between the first and last 1 bits in the bitwise logical OR of the binary representations of U and V. (Closed form expressions are given in [OREN83, OREN85].)
- E(U,V) is cyclic in the magnitudes of U and V: Specifically, E(U,V) = E(2U,2V).

These results sharpen our intuition about space requirements and immediately lead to an optimization. The added intuition is that small changes in the position of the border (i.e. the values of U and V) can lead to large increases in E(U,V). Also, it can be observed that E(U,V) is usually dominated by the number of elements on the border of the decomposed object. These observations suggest that E(U,V) is determined by the *surface area* of a spatial object, not the volume. Dependence on surface area was also reported in [SAME85b]. Note that for an explicit grid representation, space and time requirements are dictated by the *volume* of spatial objects. Thus AG techniques should be very hard to beat, especially at high resolution.

Optimization can be achieved by reducing the number of bit positions between the first and last 1s in U OR V. By expanding the boundaries of the spatial object (i.e., increasing U and V) appropriately, the number of elements generated can be decreased. Specifically, replace U and V by U' and V' such that $U' \geq U$, $V' \geq V$ and the last m bits of U' and V' are zero (for an arbitrary value of m). This is equivalent to using a coarser grid. The construction is trivial. E.g., if U = 01101101 and m = 4, then U' = 01110000.

In going to a coarser grid, the imprecision of the approximation grows slowly (e.g., based on a measurement of area). This is because it is the small (e.g., 1 pixel) elements that are being aggregated to form pixels of the coarser grid.

5.2 Proximity

In [OREN83] we derived a relationship between proximity in space and proximity in z order. The results can be summarized as follows. Proximity in space in any direction usually corresponds to proximity in z order. The greater the discrepancy, the less likely it is to occur. (A closed form expression is in [OREN83, OREN85].)

A disk page can be seen as storing all the points whose z values are in a certain range. Therefore, a disk page represents a set of points that are close in space. In general, the space covered by a page is not rectangular (see figure 6). However, under the assumptions of the analysis, there is a certain regularity in the partitioning. In [OREN83] we assumed that every page could represent a fixed number of pixels (black and white), regardless of the density of black pixels (i.e. data density). We will refer to this as the "fixed-size page" assumption. This assumption leads to the following results:

- The space is partitioned into rectangular blocks of the same size and shape. Each block contains several disk pages.
- The number of pages in a block is bounded by a number that depends only on the dimensionality of the space. (6 in 2d, 28/3 in 3d [OREN85].)

This result is used in the analysis of the following subsection.

5.3 Performance

The preceding discussion sets the stage for an analysis of performance under the assumption of fixed-size pages. In this section we give results for the number of pages accessed in range and partial match queries followed by a discussion of experimental results.

5.3.1 Analysis of Range and Partial Match Queries

Recall that a range query asks for all points (A_1, ... , A_k) such that $L_i \leq A_i \leq U_i$, i = 1, ... , k. In a partial match query, for each i, either $L_i = U_i$ or L_i and U_i do not restrict the value of A_i at all. As explained in [OREN83, OREN85], the two cases must be analyzed separately.

Under the assumption that each page covers the same amount of space, the following results are obtained. For a range query, the number of (data) pages accessed is O(vN) where v is the volume of space covered by the query (as a fraction of the volume of the entire space) and N is the total number of pages.

The number of pages accessed by a partial match query is $O(N^{1-t/k})$ where t is the number of attributes restricted in the query and k is the dimensionality of the data, $t < k$.

These results are obtained by counting the number of "blocks" covered by the query. (Recall that each block covers a fixed number of pages in a given dimension.) They match the performance predicted for kd trees [BENT75].

5.3.2 Experimental Results

The analysis of range and partial match queries is based on the fixed-size page assumption. There is no guarantee that the predicted behavior would be observed in practice. Experiments were conducted to test two hypotheses.

1. The general trends predicted by the analysis are robust; i.e., observable when the fixed-size page assumption is dropped. One of these trends is a dependence on query shape; that for queries of the same volume, "long and narrow" queries (e.g., partial match queries) require more work than queries with a squarish shape.
2. The results of the analysis are pessimistic. Observed behavior should be better than what is predicted. (For a rationale see [OREN85].) Worst case scenarios are easy to construct, but we are concerned with performance averaged over several queries.

For the experiments we implemented a prefix B+-tree to store points in z order. Page capacity was 20 points (in 2d) and 5000 points were generated for each experiment. Three sets of experiments were run, 1) uniformly distributed data (experiment U), 2) "clustered" data - 50 small clusters of 100 points each (experiment C), 3) "diagonally" distributed data - points uniformly distributed along the x=y line (experiment D). The partitioning induced by page boundaries is shown in figure 6. For each experiment queries of various rectangular shapes (and four different volumes) were run in five randomly selected locations. We measured 1) the number of (data) pages accessed for each query, and 2) efficiency, a measure indicating how much "relevant" data was on each retrieved page.

We obtained the following results:

- The general trends predicted by the analysis were observed in all experiments. The results for experiment U were closest to the predicted results and the results for experiment D were farthest.
- Except for a few data points, the predicted results provided an upper bound for the experimental results. I.e , the hypothesis about pessimism of the analysis was supported.
- Query efficiency increased with query volume. Low efficiency was usually accompanied by a low number of page accesses (fortunately).
- Analysis predicted that the greatest efficiency would be achieved by queries which are square or twice as tall as they are wide [OREN85]. This was observed in all experiments.

These results are very encouraging. Experiments in higher dimensions and with "real" data are still needed.

6. Other AG Algorithms

The work presented here dealt with spatial queries involving overlap. Simple modifications can be used for queries involving containment and proximity. (Containment implies overlap but not vice versa.) Proximity queries can often be translated into containment or overlap queries. We have already developed AG algorithms for several other spatial operators. Details are in [OREN85]. The operations considered are the following.

- **Overlay.** Polygon overlay is an extremely important operation in geographic information processing. The operation is simple to carry out on a grid representation, a pixel at a time. We have developed an AG algorithm that works directly on sequences of elements. The AG algorithm should be faster than the grid algorithm since performance is determined by the surface area of spatial objects, not volume (as discussed in Section 5.3).

- **Connected component labelling.** Another class of spatial queries has to do with the computing of "global" properties. E.g., how many black objects are in a given picture? What is the area of each object? Computing global properties of these objects is at least as hard as identifying them. One algorithm that works directly on quadtrees has appeared [SAME85c] but it is extremely complicated. We have developed an AG version of the algorithm that can be expressed very concisely.
- **Support for mechanical CAD.** Very recently, IPV researchers have been using quadtrees (and related structures) to support approximate algorithms for interference detection and related problems [MANT83, SAME85b]. AG, the spatial join in particular, can be of use here. We have developed an AG algorithm to speed up an implementation based on boundary representation. Our approach was to re-express the algorithms of [MANT83] using spatial join.

Acknowledgements

I am grateful to Umesh Dayal and Arnie Rosenthal for insightful comments on early versions of this paper. Thanks also to Nita Mulani for writing the software that produced Figure 6.

References

[ABEL83]
D. J. Abel, J. L. Smith. A data structure and algorithm based on a linear key for a rectangle retrieval problem. *Computer Vision, Graphics and Image Processing 27*, 1 (1983), 19-31.

[BENT75]
J. L. Bentley. Multidimensional binary search trees used for associative searching. *Comm ACM 18*, 9 (1975), 509-517.

[BENT79]
J. L. Bentley, J. H. Friedman. Data structures for range searching. *ACM Comp. Surv. 11*, 4 (1979), 397-410.

[BURK83]
W.A. Burkhard. Interpolation-based index maintenance. Proc. 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, (1983), 76-89.

[CHAF81]
N. S. Chang, K. S. Fu. Picture query languages for pictorial database systems. *COMPUTER 14*, 11 (1981), 23-33.

[CHAN77]
S. K. Chang et al. A relational database system for pictures. Proc. IEEE Workshop on Picture Data Description and Management (1977).

[CHOC84]
M. Chock et al. Database structure and manipulation capabilities of a picture database management system (PICDMS). *IEEE Trans on Pattern Analysis and Machine Intelligence 6*, 4 (1984), 484-492.

[DAYA85]
U. Dayal et al. PROBE - a research project in knowledge-oriented database systems: preliminary analysis. Technical Report CCA-85-03 (1985), Computer Corporation of America.

[GARG82]
I. Gargantini. An effective way to represent quadtrees. *Comm. ACM 25*, 12 (1982), 905-910.

[HASK82]
R. L. Haskin, R. A. Lorie. On extending the functions of a relational database system. Proc. ACM SIGMOD (1982), 207-212.

[LIEN77]
Y. E. Lien, D. F. Utter Jr. Design of an image database. Proc. IEEE Workshop on Picture Data Description and Management (1977).

[LIOU77]
J. H. Liou, S. B. Yao. Multidimensional clustering for database organization. *Information Systems 2*, 4 (1977), 187-198.

[LORI83]
R. A. Lorie, W. Plouffe. Relational databases for engineering data. IBM Research Report RJ 3847 (43914) 4/6/83 (1983).

[MANT83]
M. Mantyla, M. Tamminen. Localized set operations for solid modeling. *Computer Graphics 17*, 3 (1983), 279-288.

[MERR78]
T. H. Merrett. Multidimensional paging for efficient database querying. Proc. Int'l Conference of Management of Data, Milan (1978), 277-290.

[MERR82]
T. H. Merrett, E.J. Otoo. Dynamic multipaging: a storage structure for large shared databases. Proc. 2nd Int'l Conference on Databases: Improving Usability and Responsiveness, Jerusalem (1982).

[MERR84]
T. H. Merrett. Relational Information Systems, Reston Publishing, Reston, Virginia (1984).

[NIEV84]
J Nievergelt, H. Hinterberger, K. C. Sevcik. The grid file: an adaptable, symmetric multikey file structure. *ACM TODS* 9, 1 (1984), 38-71.

[OREN82]
J. A. Orenstein. Multidimensional tries used for associative searching. *Information Processing Letters 14*, 4 (1982), 150-157.

[OREN83]
J. A. Orenstein. Algorithms and data structures for the implementation of a relational database. Ph.D. thesis, McGill University, (1983). Also available as Technical Report SOCS-82-17 (1982), School of Computer Science, McGill University.

[OREN84]
J A. Orenstein, T. H. Merrett. A class of data structures for associative searching. Proc. 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (1984), 181-190.

[OREN85]
J. A. Orenstein. Spatial query processing in

PROBE. Working paper. To appear as a Technical Report, Computer Corporation of America.

[OUKS83]
M. Ouksel, P. Scheuermann. Storage mappings for multidimensional linear dynamic hashing. Proc. 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, (1983), 90-105.

[ROBI81]
J. T. Robinson. The K-D-B tree: a search structure for large multidimensional dynamic indexes. Proc. ACM SIGMOD (1981), 10-18.

[SCHE82]
P. Scheuermann, M. Ouksel. Multidimensional B-trees for associative searching in database systems *Information Systems* 7, 2 (1982), 123-137.

[SMIT84]
J. D. Smith. The application of data base management systems to spatial data handling. Project report, Department of Landscape Architecture and Regional Planning, University of Massachusetts, Amherst (1984).

[STON83]
M. Stonebraker et al. Application of abstract data types and abstract indices to CAD data. Proc. ACM SIGMOD conference on engineering design applications (1983).

[STON85]
M. Stonebraker. Inclusion of new types in relational data base systems. Memorandum No. UCB/ERL M85/67, Electronics Research Laboratory, College of Engineering, University of California, Berkeley (1985).

[SAME85a]
H. Samet. The quadtree and related hierarchical data structures. *ACM Comp. Surv. 16*, 2 (1984), 187-260.

[SAME85b]
H. Samet, M. Tamminen. Bintrees, CSG trees and time. SIGGRAPH (1985).

[SAME85c]
M. Samet, M. Tamminen. Computing geometric properties of images represented by linear quadtrees. *IEEE Trans. on Pattern Analysis and Machine Intelligence* 7, 2 (1985), 229-239.

[TAMM81]
M. Tamminen. The EXCELL method for efficient geometric access to data. *Acta Polytechnica Scandinavica*, Mathematics and Computer Science Series No. 34 (1981).

[TAMM82]
M. Tamminen, R. Sulonen. The EXCELL method for efficient geometric access to data. Proc 19th ACM Design Automation Conf. (1982), 345-351.

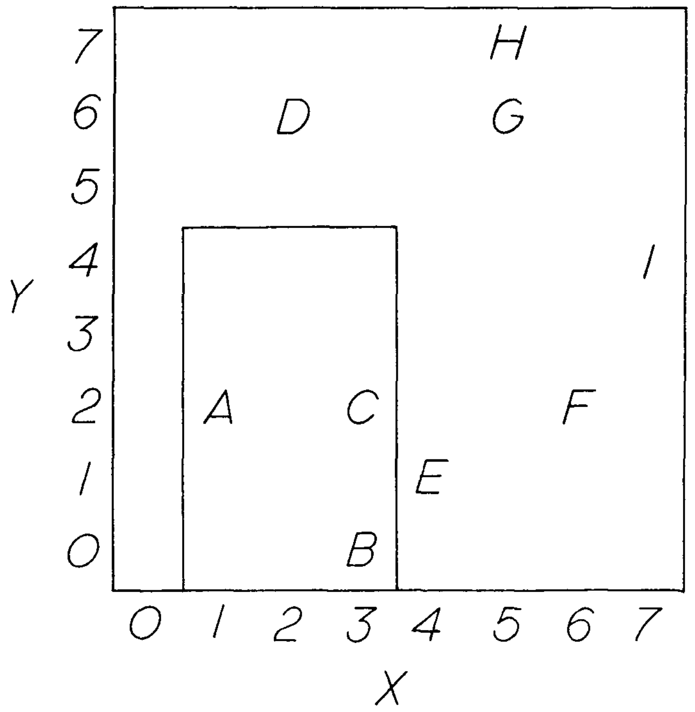


Figure 1:

Spatial interpretation of the range query
$1 \leq X \leq 3$ & $0 \leq Y \leq 4$.

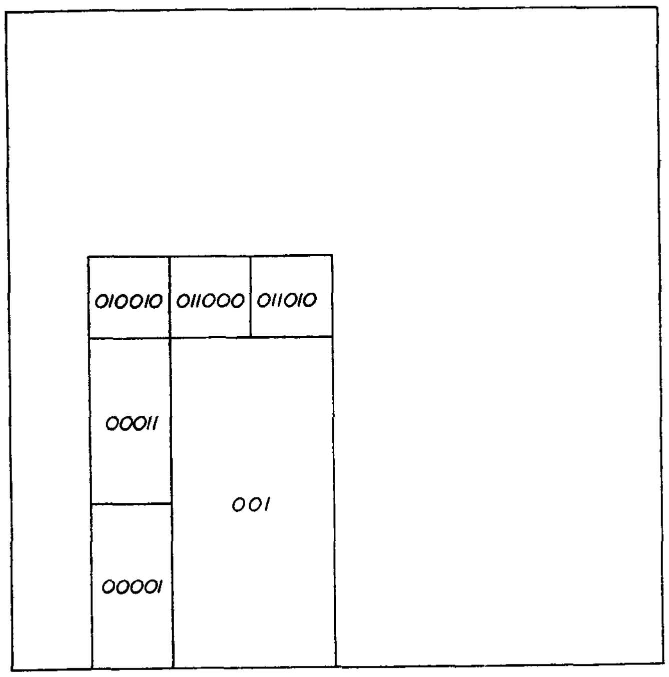


Figure 2:

The decomposition of a box. Each element is labelled with its z value. A z value is constructed by interleaving the bits describing the range of values covered by the element along each dimension. E.g. The element labelled 001 is handled as follows. The ranges of X and Y values covered are described by [2:3, 0:3] (i.e. $2 \leq X \leq 3$ and $0 \leq Y \leq 3$). In binary, these ranges are [010:011, 000:011]. The element is characterized by the common prefix of each range: [01, 0]. Interleaving these bits (starting with X) yields 001.

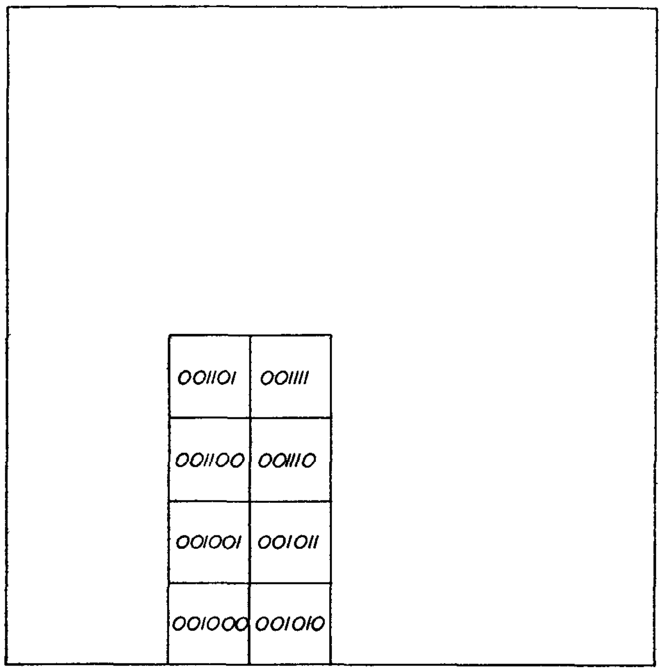


Figure 3:

The z values in an element are consecutive. The element shown is the large element from figure 2 (z value = 001). Note that all z values inside the element have the same prefix, 001.

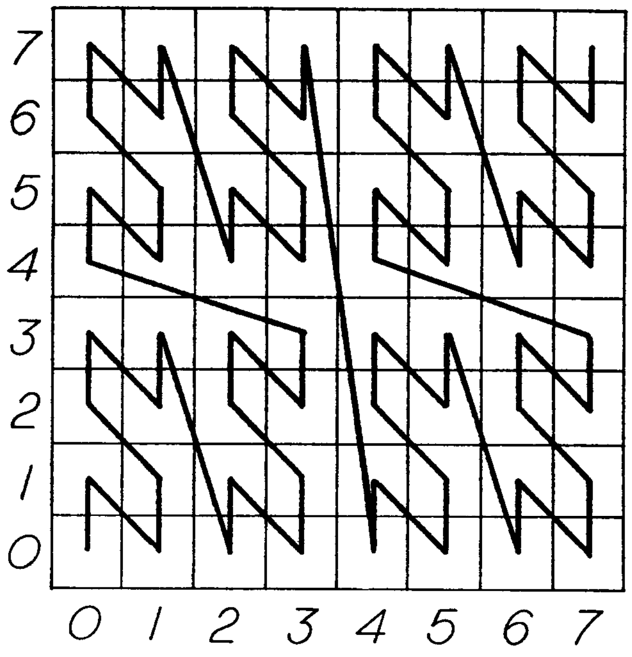


Figure 4:

Spatial interpretation of z order. The rank of a point is obtained by interleaving the bits of the coordinates and interpreting as an integer. E.g. [3, 5] → (011, 101) → 011011 = 27.

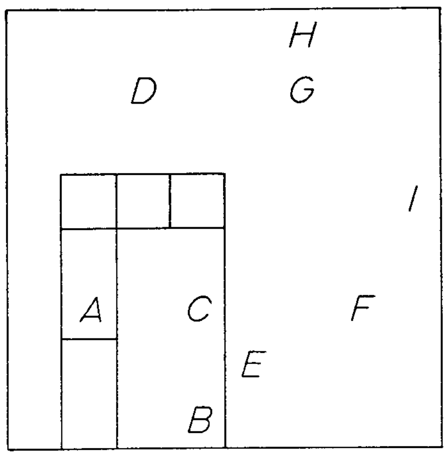


Figure 5:

Range search algorithm. Each point is a one pixel element. The box is decomposed to elements. Merge the sequences of elements, looking for containment of point elements by box elements.

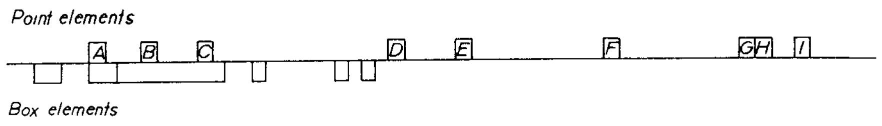
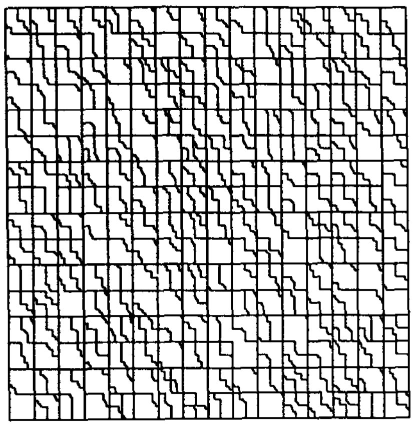
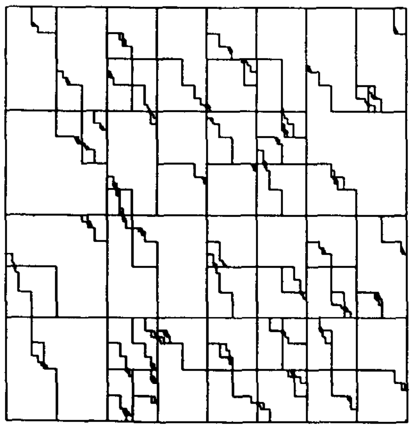
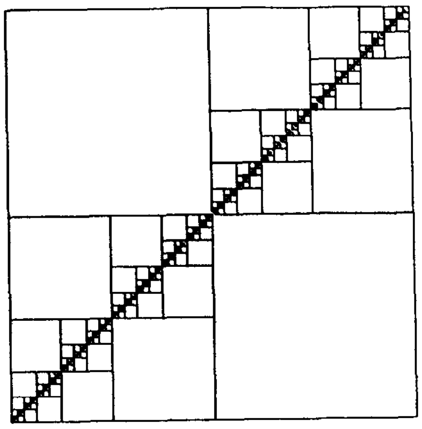


Figure 6.

Partitioning of the space induced by page boundaries in the zkd B+-tree.


a) **Experiment U.** Uniformly distributed points.


b) **Experiment C.** Uniformly distributed clusters of points.


c) **Experiment D.** Points uniformly distributed along the line X=Y.