Redundancy in Spatial Databases

Jack A. Orenstein
Object Design, Inc.*
One New England Executive Park
Burlington, MA 01803

odi!jack@talcott.harvard.edu

Abstract

Spatial objects other than points and boxes can be stored in spatial indexes, but the techniques usually require the use of approximations that can be arbitrarily bad. This leads to poor performance and highly inaccurate responses to spatial queries. The situation can be improved by storing some objects in the index redundantly. Most spatial indexes permit no flexibility in adjusting the amount of redundancy. Spatial indexes based on z-order permit this flexibility. Accuracy of the query response increases with redundancy, (there is a "diminishing return" effect). Search time, as measured by disk accesses first decreases and then increases with redundancy. There is, therefore, an optimal amount of redundancy (for a given data set). The optimal use of redundancy for z-order is explored through analysis of the z-order search algorithm and through experiments.

1 Introduction

A spatial database must be able to store spatial objects and retrieve those objects by specifying queries involving spatial predicates such as overlap, containment, and proximity. There has been much progress on spatial searching in the past fifteen years. There is a wide variety of structures for use in both main memory (e.g. kd trees [BENT75] and quadtrees [SAME84]) and on secondary storage [GUTT84, LOME87, MERR78, MERR82, NIEV84, OREN84, ROBI81, SELL87]. However, most of this work has addressed only range queries, in which the data objects are points and the goal is to locate all points that fall within a query box. (Each edge of the box is parallel to one axis of the space.)

Applications such as geographic information systems and CAD require the ability to deal with spatial objects other than points and boxes. This motivates consideration of the overlap query. The *overlap query* takes as input two sets of spatial objects, R and S, and returns a set of pairs (r, s) such that r is a member of R, s is a member of S, and r and s overlap spatially [OREN88]. The range query is a special case in which one set contains points and the other set contains a single box.

Spatial objects, other than points and boxes, often have to be approximated before any of the spatial indexes listed above can be used. The basic techniques involve placing the objects in containers, yielding a conservative approximation, and parameterization. These techniques are described below. However, because the approximations are conservative, query processing strategies based on spatial indexes must use the following two-step strategy:

Filter step: The spatial index is used to rapidly eliminate objects that could not possibly satisfy the query. The result of this step is a set of *candidates* which includes all the results and possibly some false hits.

Refinement step: Each candidate is examined. False hits are detected and eliminated.

One technique for dealing with non-point objects is parameterization. Some simple spatial objects can be described by a small number of parameters and represented by a point in parameter space (e.g., see [FALO87, HINR85]). For example, a circle can be represented by three parameters (center coordinates and radius). Similarly, a box in k dimensions is represented by a point in 2k dimensions. Objects with more complicated structure, e.g. arbitrary polygons, have to be approximated by a bounding object with a simple shape. The container can then be parameterized.

The use of parameterized containers is not entirely satisfactory. For a disjoint object or an object with holes, the volume of the container may far exceed that of the object. As a result, the object will be involved in many false hits. A poor approximation can also occur when the object is continuous and has no holes, but because of its shape, is not capable of filling the container, e.g., a line segment inside a containing circle. This can be dealt with by using containers with more parameters, (e.g., an ellipse instead of a circle), but this increases the number of parameters which leads to decreased performance as discussed below.

Another problem with parameterization is that the number of dimensions in the parameter space is greater than the number of dimensions in the original space. This is a problem because the performance of all spatial search structures for point data degenerates as the number of dimensions increase.

These problems suggest that spatial searches based on the parameterization of containers will yield increasingly inaccurate answers as the complexity of the data objects increases. In the extreme case, the filter step would return the entire set of data objects, all of which would then have to be considered by the refinement step.

Complexity of spatial data need not destroy the accuracy of the filter step. An alternative is to introduce redundancy into the spatial index. The goal is to trade space for accuracy, while retaining the speed of the filter step. Several researchers have used redundancy to deal with non-point spatial objects.

* The work reported here was performed on equipment owned by the author, and does not relate to anticipated Object Design products.

Quadtree-like structures, the grid-file, the R+-tree, and the cell tree partition objects at cell, grid or page boundaries [GUNT89, HORN87, SELL87, TAMM82]. A copy of the object is created for each partition induced by boundaries.

The goal of this paper is to explore the use of redundancy in connection with the spatial search structures based on z-order [OREN84, OREN86a, OREN88]. Quadtrees, grid files (and variants of each), and the R+-tree do not permit any flexibility on redundancy. In each case, the amount of redundancy in the index is fixed, determined by the size and shape of the data objects and by aggregate properties such as density and distribution. Objects are replicated more in densely-filled regions of the space than in sparsely-filled regions. (This phenomenon is discussed in section 3.2.)

On the other hand, in a z-order based spatial index, *the redundancy of the index can be controlled.* This is due to the fact that replication can be decided on a per-object basis, and that the density of the space near the object does not affect the decision. Because redundancy can be controlled, it is possible to fine-tune the trade-off between the time required for the filter step and for the refinement step. The optimal balance, according to a cost model, can be selected, *but only if the spatial index provides control over redundancy.* Z-order based spatial indexes provide this control.

Section 2 provides a summary of z-order based spatial search structures. Section 3 discusses the ways in which redundancy can be introduced into spatial search structures, and identifies a collection of four query processing strategies that use different approaches to redundancy. Section 4 shows how redundancy can be controlled in z-order based structures, and identifies the perils of having too much or too little redundancy. Section 5 establishes that there is an optimal amount of redundancy for a given data set. Section 6 presents experimental results that support the reasoning in sections 4 and 5, and show that the results hold for one example of real-world data. Concluding remarks are in section 7.

2 Overview of z order-based search structures

In [OREN84] a solution to the range query problem is given. The technique used is to transform the problem of finding all the points in a k-dimensional (k-d) box into an equivalent search problem in 1-d. Each data point is transformed into a 1-d interval of size 1, and the query box is transformed into a set of intervals of varying size. A data point is contained by the box iff the corresponding interval falls in one of the query's intervals. This approach yields a *family* of data structures for evaluating range queries. A member of this family is derived by providing a data structure that supports random and sequential access, e.g. a sorted array, an AVL tree, or a b-tree. The search algorithm is expressed in terms of random and sequential accesses to the underlying data structure. In spite of the generality of this approach, performance is comparable to that of more specialized structures. Furthermore, this approach permits all the theory, techniques, and even *software,* that has been developed for ordinary (one dimensional) searching problems, to be applied to spatial searching. For example, the experiments reported in section 6 were carried out using the *zkd b-tree*, an ordinary b-tree loaded with 2-d boxes transformed to 1-d intervals.

This approach can be generalized to deal with arbitrary spatial objects involved in overlap queries [OREN86a, OREN86b]. Each spatial object in each input set is transformed to a set of 1-d intervals. An algorithm called *spatial join* implements the filter step. Each resulting candidate is a pair of objects, one from each input file, that are likely to overlap. The output from spatial join has to go through a refinement step as described above. Spatial join and filtering algorithms for other spatial problems appear in [OREN88]. These algorithms comprise the *geometry filter* (GF).

2.1 Decomposition: generating the geometry filter's representation

As discussed above, the geometry filter works by transforming a k-d spatial object into a set of 1-d intervals. There are many ways to do this. Many of the modularity and performance benefits of the geometry filter derive from the particular way in which the geometry filter does this transformation.

The "conceptual" representation used by the geometry filter is a grid of fixed resolution. The representation for an object is obtained by noting which cells are completely or partially occupied by the object. This representation is a conservative approximation. Partially occupied cells are included so that the filtering property is retained. If such cells were omitted, then the approximation would not be conservative, and some positive results would be lost in the output of the filter step.

The grid representation can easily be transformed to 1-d, e.g., by listing the occupied cells in row-major order. However, the number of occupied cells depends on the volume of the object. As a result, the space and time requirements for algorithms based on this representation will be very high. Instead, an encoding of the grid is used. The space is recursively partitioned until the resolution of the grid is reached. Regions that are entirely contained in the object do not have to be split further. The space requirement for this encoding is proportional to the surface area of the object, not the volume, so the space and time requirements are much better. However, further improvements can be obtained, as described in section 4.1.

By constraining the partitioning process as in [OREN84, OREN88], a highly compact representation of the spatial object can be obtained. A region created by partitioning under these constraints is called an *element*. Together, these constraints lead to a very concise description. Typically, each element can be described by one 32-bit word. Each partition is represented by one bit, and the relationship of the element to the partition is described by the value of the bit. (In 2-d space, a 0 means to the left of or below. A 1 bit means to the right of or above.) The bit-sequence corresponding to an element is called a z *value*. Given a z value, the size, shape and position of an element can be reconstructed.

Figure 1 shows the encoding of a spatial object achieved by elements. This is a more compact representation than the explicit listing of each grid cell, since the space requirement for each element is the same, regardless of its size.

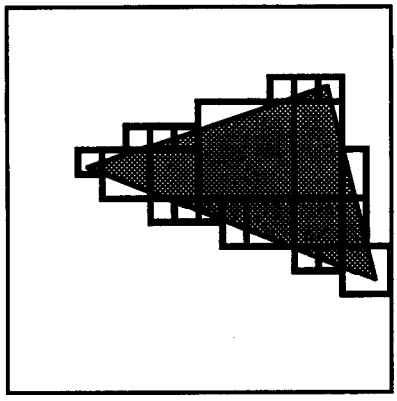


Figure 1. Decomposition of a spatial object

The *decompose* algorithm is responsible for generating the elements corresponding to a spatial object. In spite of the constraints on the partitioning process, there are actually a variety of decomposition strategies. It will be shown that these strategies provide the key to controlling redundancy and, as a result, the performance of z-order based search structures.

The geometry filter's representation for a set of spatial objects is obtained by decomposing each object, associating each z value with the object, and then merging all the (z value, object) pairs into a single 1-d search structure, keyed by z value. Redundancy occurs when the same object is associated with multiple z values. This structure will be referred to as the GF representation, GF file, or GF sequence.

The geometry filter algorithms operate by generating random and sequential accesses against GF sequences.

2.2 Spatial join: generating candidates

The spatial join algorithm performs a merge of two GF sequences, searching for situations where an element in one sequence, as represented by its z value, contains an element from the other input. (This can be determined by checking whether one z value is a prefix of the other.) When such a pair is found, a candidate, comprising the objects associated with the elements, is generated. If the sizes of the input sequences are n and m, then the time for the merge is O(n + m). However, it is often possible to do much better. For example, consider a range query. The n data points yield a GF sequence of size n. The other input set will contain some small number of elements for the query box. The merge can be optimized by taking advantage of the fact that the elements of the query box usually occur in clusters. For example, all the data points whose z values are less than that of the first query element can be skipped. Similarly, the data points between elements of the query box and those data points following the last element of the query can be skipped, (see figure 5). The ability to "skip over" elements that are clearly not of interest is the source of the random access requirement.

This optimization has been built into the spatial join algorithm. The details are in [OREN88]. With this optimization, and assuming a certain distribution of data (which is more regular than uniform), it can be shown that the expected performance for range queries is O(fN) where f is the fraction of the space covered by the query, and N is the number of data objects [OREN83]. This is comparable to the performance of other, special-purpose structures. Experiments have shown the robustness of this result for a variety of data distributions, query sizes and query shapes [OREN86b].

3 Redundancy

The traditional goal of clustering is to place objects that will be retrieved together, near one another on disk (i.e., the same page). For spatial search predicates, objects that are retrieved together are usually near one another in the space being modeled. That is, proximity in space must translate into proximity in secondary storage in order to obtain the best possible performance for spatial queries. Redundancy within a spatial index can be used to obtain good clustering when the topology, shape, or size of a spatial object are such that no single placement within the file makes sense.

Redundancy can be introduced by decomposing a spatial object into smaller, simpler objects. The spatial index would contain an entry for each component. There are two ways to do this. One way is to take advantage of "natural" object structure, e.g. the line segments that approximate a section of a road, or the convex polygons whose union comprises an arbitrary polygon [GUNT89]. This approach is described in section 3.1. The alternative, described in section 3.2, is to partition the data objects along boundaries that are natural for the spatial index.

3.1 Redundancy induced by object structure

In some applications, the spatial objects being stored will have some structure that can be easily exploited to partition the object. For example, in mechanical CAD applications, solids are often described constructively - in terms of set operations on some fundamental shapes. If a disjunctive normal form can be obtained, then there is a simple decomposition (not a partitioning). Each term of the disjunctive normal form expression can be indexed separately.

In geographical applications, the basic spatial object is usually a polygon. These polygons may be disjoint and may have holes, but they can always be partitioned into a collection of convex polygons. Convex polygons can be approximated more accurately than arbitrary polygons, so the filter step will be more accurate. Furthermore, the refinement step will spend less time on each candidate since operations on convex polygons are faster than operations on arbitrary polygons.

3.2 Redundancy induced by region boundaries

In a spatial index, each data page can be thought of as covering some region of the space. For example, in the grid cell these regions are rectangular but not necessarily uniform in size. Region boundaries induce a partitioning of the data objects, resulting in redundancy. A spatial object is replicated on every page whose space it occupies. The structure used in Geo-kernel actually materializes and stores the object partitions, instead of just replicating the object for each page [HORN87]. In the R-tree, index pages may cover overlapping regions of space, leading to ambiguity during searching [GUTT84]. The R+-tree addresses this problem by introducing redundancy [SELL87]. Index pages no longer cover overlapping regions of space, but this requires that some objects be represented redundantly. The R-tree would have avoided the redundancy, but the ambiguity has an impact on the search cost.

Z-order based structures are unique in their ability to use

arbitrary amounts of redundancy. To understand why, first consider what happens in a grid file. In the grid file, a boundary between data pages induces replication of an object that spans the regions covered by those pages. That is, redundancy depends on the size and shape of the object *and* on the density of objects in the same vicinity. If an object is inserted in a sparse region of the space, it is possible that no redundancy will be required (see the right side of the space in figure 2). However, if the same object is inserted into a dense region of the space, where there are many page boundaries, then redundancy will be required (see the left side). Furthermore, as shown in the diagram below, insertions into such a region may drive up the redundancy required by *several* objects in that region, since, as more splits are formed, more redundancy is introduced.

The situation is different for the zkd b-tree. A spatial object is partitioned into elements according to a *decomposition strategy*. Three of these strategies are described in section 4.1. Two of the decomposition strategies have parameters that control the amount of redundancy on a per-object basis. Because of this level of control, redundancy for the entire index can be controlled in a distribution- and density-independent way.

Insertion into a zkd b-tree may result in a page split, as in an ordinary b-tree. The resulting pages together cover the same space as the page that was split. That is, the split of a page results in the partitioning of the region represented by the page into sub-regions. The boundary between the sub-regions is shaped like a stair-case, and the regions are, in general, not boxes. Element boundaries are *unaffected* by these splits. The space covered by an element may actually span the boundary between the newly created sub-regions.

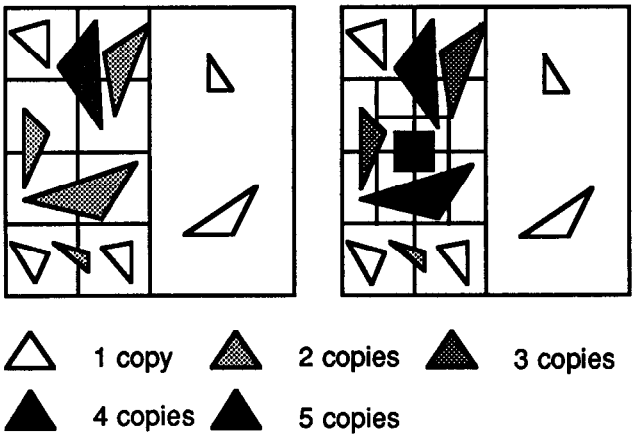


Figure 2

Grid file, before and after insertion of an object (the square). Page capacity is two objects. The insertion causes six data pages to be added, which in turn, increases the replication of several objects in the vicinity.

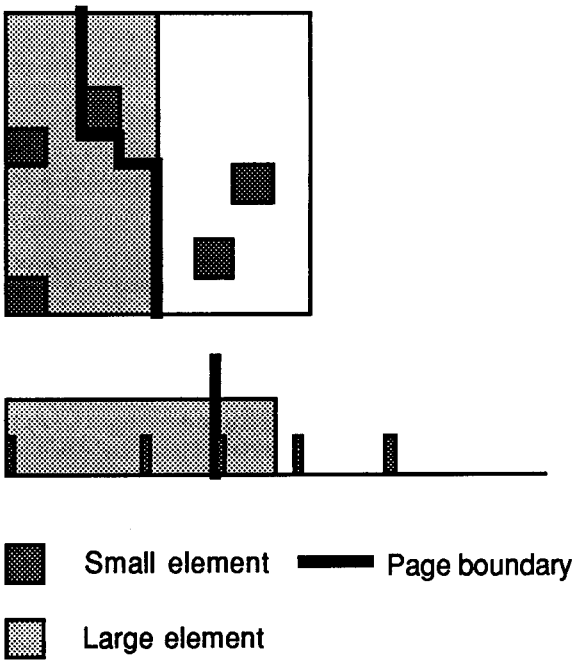


Figure 3

2-d and 1-d view of a zkd b-tree with two data pages. Page capacity is three elements. The first page has a large element and two small elements. The large element extends into the next page which has three small elements.

This occurs because an element is ordered according to the start, not the end, of its interval in 1-d space. It is important for the existence of the element to be known on all covered pages, not just the page containing (the origin of) the element. This information is encoded very compactly as described in section 4.2.2.

3.3 When is each approach to redundancy appropriate?

Each approach to redundancy requires some outside help in the form of subroutines that are specific to the spatial objects being stored. Structure-based partitioning requires a subroutine to identify the components of an object. A spatial database system would call this subroutine when spatial objects are being added to the database. Partitioning along region boundaries requires subroutines to determine the relationship between the object and a (usually rectangular) region of space, as in the PROBE spatial query processor [OREN88]. The Geo-kernel spatial query processor requires compose and join functions that partition and reconstitute spatial objects; i.e. replication involves copying only part of the object [HORN87].

These two approaches to redundancy are not mutually exclusive. Following the structure-based partitioning of a spatial object, the resulting components have to be indexed. Partitioning by region boundaries is then useful. (Structure-based partitioning doesn't make sense as a second step - the first step should yield partitions or components simple enough not to require further structure-based partitioning, no

matter which approach is used.) However, it is also possible to use each technique in isolation or to proceed without introducing any redundancy. Thus there are four general strategies to spatial query processing:

1 **No redundancy**: Each spatial object is placed in a container which is then represented by a point in a parameter space.

2 **Redundancy induced by object structure**: The spatial objects are partitioned into natural components. Each component is placed in a container and represented by a point in a parameter space.

3 **Redundancy induced by region boundaries**: The spatial objects are partitioned along region boundaries. The resulting partitions, which describe pieces of the object in the original space, are placed into a spatial index.

4 **Redundancy induced by object structure, then by region boundaries**: Partition the spatial objects into their natural components and then partition each component along region boundaries. Each piece resulting from this two-step partitioning is placed into an index which describes the original space.

There has been no systematic study of these alternatives. Experience on the PROBE project provides only anecdotal information [DAYA87]: A sample application involved a road network. The basic spatial object were road segments (line segments) and gas stations (points). One of the sample queries asked for all gas stations within five miles of a given road. In this case, strategy 3 performed better than strategy 4 on several criteria - space requirements, preprocessing time, and speed of the filter step, although the result of this step was less accurate.

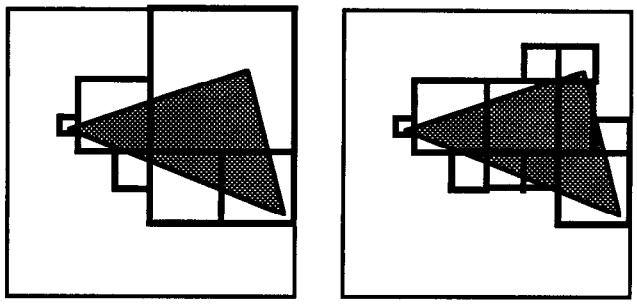
The work reported here is primarily concerned with strategy 3.

4 Controlling redundancy in z-order search structures

For point data, there is only one possible GF representation in which there is one element for each point. The size of each element is determined by the resolution of the space, i.e., the GF representation of a point is a single cell in the "conceptual" grid of the GF.

4.1 Decomposition strategies

For spatial objects other than points, a variety of representations can be derived. For example, the object shown in figure 1 can also be decomposed as shown in figure 4.



(a) SIZE-BOUND(6) decomposition (b) ERROR-BOUND(6) decompostion

Figure 4

The decomposition shown in figure 1 uses the PRECISE strategy. The algorithm appeared in [OREN86a, OREN88]. The remainder of this paper will be primarily concerned with two other decomposition strategies, to be known as SIZE-BOUND and ERROR-BOUND, shown in figures 4a and 4b respectively.

SIZE-BOUND(n) permits no more than n elements to be generated during the decomposition of a single spatial object[1]. The "exploration" of the object being decomposed is breadth-first. As regions are split, the sub-regions are processed FIFO. A depth-first exploration would reach the upper bound on elements before some very large regions (covering 1/2 of the space, 1/4 of the space, ...) had a chance to be split.

A split of a region yields either one or two non-empty regions, so a split can be described as being *one-way* or *two-way*. SIZE-BOUND(n) permits two-way splits until n elements have been formed. Once the bound is reached, only one-way splits are permitted. Each one-way split improves the accuracy of the representation without increasing the number of elements.

The result of a SIZE-BOUND(n) decomposition of an object may actually have fewer than n elements. This occurs because the elements resulting from terminal sequences of two-way splits can be merged to re-form a single element. This reduces redundancy without decreasing accuracy.

ERROR-BOUND decomposition is similar to SIZE-BOUND, but the decomposition process is terminated in a different way. The goal of ERROR-BOUND is to keep a bound on the accuracy of the representation rather than its size. This is done by placing a bound on the distance between the border of an object and the border of a containing element. In practical terms, this means that a two-way split is permitted if the resulting elements are above a threshold size. The threshold is controlled by a parameter, g. ERROR_BOUND(g) requires a two-way split of an element that covers more than 2^{-g} of the space. Equivalently, the number of two-way splits forming an element must not exceed g. The final result of an ERROR-BOUND(g) decomposition may have elements that cover more than 2^{-g} of the space. This is due to merging of elements resulting from terminal sequences of two-way splits, as above.

There are some relationships among these strategies.

[1] "SIZE" refers to the size of the output from the decomposition step, not the amount of space covered by the elements.

PRECISE is the limiting case of both SIZE-BOUND and ERROR-BOUND. Consider a space with resolution 2^d (i.e., the conceptual grid has 2^d pixels). PRECISE is equivalent to SIZE-BOUND(2^d) and to ERROR_BOUND(d). Also, SIZE-BOUND(1) is equivalent to ERROR-BOUND(0). Except for these relationships, SIZE-BOUND and ERROR-BOUND cannot be compared directly because their parameters control different qualities - size of the decomposition, and accuracy of the decomposition respectively. However, the parameter of each strategy controls redundancy. The approach pursued in section 6 is to observe how the performance and accuracy of the filter step vary with redundancy for each algorithm.

4.2 The cost of an inaccurate decomposition

Analytical results [OREN83] and experimental results [OREN86b] indicate that the number of data page accesses needed to evaluate a range query on point data is O(fN) where f is the fraction of the space covered by the query, and N is the number of data pages. (N is proportional to the number of elements since a point always decomposes to yield one element.) The exact number of pages read will be higher or lower than fN depending on the distribution of the data, the aspect ratio of the query box, the load factor of the data file, and other factors.

4.2.1 Why redundancy is bad

For data objects other than points, the number of pages retrieved for a single box-shaped query object will still be O(fN), although the value of N will be larger due to redundancy. The reasoning is as follows. As far as the filter step is concerned, the identities of the objects associated with the elements are irrelevant. It does not matter if every element comes from a different object, as is the case with point data, or if some objects gave rise to multiple elements, as would happen for non-point data with redundancy. So there is a clear advantage to minimizing redundancy: N is minimized and therefore the number of page accesses is minimized, at least according to this very non-rigorous argument.

4.2.2 Why redundancy is good

A more careful analysis of the spatial join algorithm shows that low redundancy is not a clear win. Specifically, the effectiveness of the algorithm used to optimize the merge is adversely affected by low redundancy. The explanation is somewhat involved, and it requires a discussion of how the merge is optimized.

Consider two GF files, X and Y. The spatial join of X and Y is, essentially, a merge, so if an element of X has just been processed, then it is correct to simply advance to the next item in X. However, it is often possible to do better. Under certain conditions, it is correct to skip over some consecutive elements of X starting with the one following the element just processed. The next *relevant* X element can be located by doing a random access in X with the current element of Y. The elements that are skipped would not have contributed anything to the filter output.

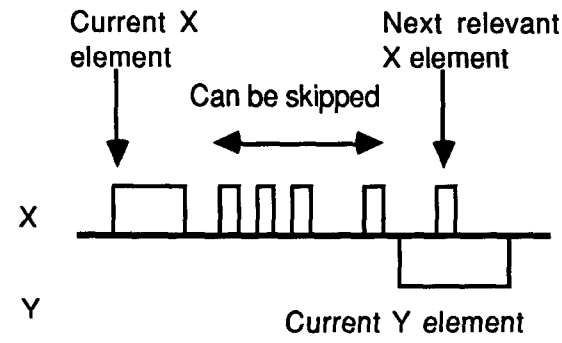


Figure 5

The logic of the optimization is fairly intricate. Details can be found in [OREN88]. For our purposes, what is important is that the random access performed during optimization is sometimes too "optimistic". The random access locates the element that starts at or following the z value used in the access. Containing elements are ordered before contained elements, so it is possible that a random access will land inside a containing element. The origin of the containing element is behind the point of access. When this happens, it is necessary to move back to the largest containing element that has not been visited yet[1].

[1] Containing elements appear before, not after, contained elements because the elements within a GF file are ordered by the lowest z value of the interval, not the highest. If the highest z value in an interval were used for ordering then the merge logic would fail.

For a given element in a GF file, the z values of containing elements can be recorded extremely concisely. One bit is sufficient to record the presence of a containing element of a given size, due to the constraints placed on the partitioning process. Therefore, the total "context" of an element can be recorded in a single word (assuming that the z value itself is a single word). The value of this word for each element can be derived in a single pass of a GF file once it is built. Insertions and deletions require maintenance of these words of context.

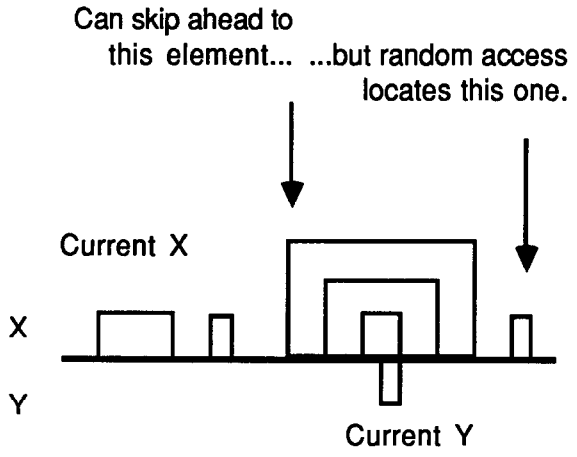


Figure 6

This sequence of events, a random access followed by another random access with a smaller key value, is a *backup*.

Large elements are more likely to result in backup than small ones since they are more likely to contain other elements. Since low redundancy results in large elements (since less accuracy is possible), the frequency of backup during a query is increased as redundancy is decreased. The cost to process a backup is one page read which may result in a page fault. The fault occurs if the origin of the element that caused the backup - the containing one - is not on the page that was read due to the original random access. Thus, large elements are not only more likely to cause backup, but when a backup occurs, they are more likely to result in a page fault.

5 Comparison of the decomposition strategies

The discussion of sections 4.2.1 and 4.2.2 establishes that, for a given data set and query, there is a non-trivial optimal amount of redundancy, R_{opt}. To summarize, as redundancy increases, so does file size - the value of N in O(fN) - so the number of pages accessed during the filter step will increase also. On the other hand, as redundancy is decreased, fN approaches its minimal value, but the probability of fault-causing backups increases. These tendencies balance one another to minimize page accesses at R_{opt}. This section and section 6 are concerned with the problem of determining which strategy attains the lowest R_{opt}, and comparing other criteria - filter execution time and accuracy - at R_{opt}.

The PRECISE strategy is not considered further, for two reasons. First, it is an extreme case of both SIZE-BOUND and ERROR-BOUND. Second, it appears to be a clear loser. While the strategy cannot be beat in terms of accuracy (as discussed in section 4.1), its performance, as measured by page accesses during the filter step, can be made arbitrarily bad by increasing the resolution of the space. As resolution increases, redundancy is likely to increase past R_{opt}, because the number of elements near the border increases (while their size decreases). The other strategies yield less accurate representations, but the size of their representations are bounded by factors having nothing to do with resolution.

By placing an upper bound on the number of elements, the SIZE-BOUND strategy addresses the most serious problem of the PRECISE strategy - redundancy gone wild in an attempt to minimize the work to be carried out in the refinement step. Section 6.3 argues that the extra precision is negligible, and not worth the greatly increased redundancy. However, SIZE-BOUND introduces redundancy where it may not be helpful. Elements that are small and close to one another can be merged into a single larger element[1]. The new element is still small since the original elements were close together. The net result is a reduction in the size of the representation, (i.e., redundancy), with only a small decrease in the accuracy of the representation. This phenomenon can be seen by comparing figures 1 and 4.

However, reasoning as in section 4.2.2, it could be argued that this replacement creates an opportunity for backup, and that this might cancel the benefit gained from the decrease in redundancy. This is certainly true if the newly-created element is large. But, according to the scenario outlined, the element created is small, and it is likely that the backup would not generate a page fault. Of course, the terms "small" and "large" are relative, but there is an objective measurement that can be used: an element can be considered to be small if it is likely to fit within one page (and therefore not cause a backup).

The rationale for the ERROR-BOUND strategy is now clear. For an element smaller than the ERROR-BOUND threshold, redundancy (i.e., more splitting) is considered to be harmful because space requirements are increased while precision is not significantly increased and the likelihood of a fault-causing backup is small. Above this threshold, the added redundancy is outweighed by increased precision and decreased likelihood of a fault-causing backup. (Note that backup affects the filter step, while precision only affects the speed of the refinement step.)

6 Experimental results and analysis

Experiments were conducted for the following reasons:

1. To verify the reasoning of section 5, the existence of R_{opt} in particular.

2. To determine which strategy, SIZE-BOUND or ERROR-BOUND makes best use of redundancy. In other words, at their respective R_{opt} values, which strategy performs best?

3. To determine which strategy is best in terms of filter speed and accuracy at R_{opt}.

It is doubtful that there is any single answer to (2) and (3). More likely, the optimal strategy and optimal amount of redundancy will depend on the size, shape and topology of the data and query objects, the distributions of size, shape and location, the denseness with which the space is covered, the amount of overlap among objects, and other factors.

The experiments reported here are preliminary, in that they ignore several of these factors. The design of the experiments is as follows:

[1] The elements must be close in the 1-d representation. Proximity in space translates to proximity in the 1-d representation with high probability, but this is not guaranteed.

- The space in which the experiments were conducted had a resolution of 1000 by 1000.
- In each experiment there were 5000 data objects and 1 query object.
- In each experiment, the data objects were squares of some uniform size, uniformly distributed. Four different edge lengths were used, 5, 10, 20 and 30 pixels, corresponding to objects covering 25, 100, 400 and 900 objects, respectively.
- In each experiment, the query objects were squares with edge lengths ranging from 20 to 200 pixels. Each query was run at five randomly selected locations.
- Data objects were decomposed according to the SIZE-BOUND strategy with upper bounds on size of 1, 2, 3 and 4, and according to the ERROR-BOUND strategy with error bounds set to 0, 4, 8 and 12. Note that SIZE-BOUND(1) is equivalent to ERROR-BOUND(0).
- Query objects were always decomposed according to the ERROR-BOUND(8) strategy.

The data and queries were loaded into zkd b-trees (b+-trees to be precise), with a data node capacity of 20 boxes. The software for the experiments was written in C and executed on a Commodore Amiga 2000, (a 68000-based system).

In each experiment, the number of data page faults was measured. Separate counts were kept for each level of the trees, but the graphs below show faults at the leaf level only.

LRU buffer management was used with 15 buffers. These buffers were dedicated to the data tree; the query tree used a separate set of buffers.

Of all the design decisions, the most dubious are those having to do with the distribution of object size, shape and especially placement. The effect of non-uniform distributions for these parameters will be reported in future papers. The effect of some highly non-uniform distributions on the performance of the zkd b+-tree for range queries on point data was reported in [OREN86b]. Also, section 6.4 reports on a second set of experiments with the same design as above, but with data obtained from a real-world application - VLSI design.

6.1 Space requirements

The space requirement of a spatial search structure is determined by several factors, including storage utilization of the structure and by redundancy. Most spatial search structures cannot guarantee lower bounds on storage utilization for non-uniform distributions of point data, for the data pages or the index pages, or both. Only the hB-tree [LOME87], the R-tree [GUTT84], and the zkd b-tree provide any guarantees. Since a zkd b-tree *is* an ordinary b-tree, the results for b-trees apply - a lower bound of 50% and an expected load factor of $\ln 2 \approx 70\%$. For all zkd b-trees created, for all decomposition strategies and data object sizes, the load factor was between 67% and 72%. As explained in section 3.2, redundancy in a zkd b-tree is a property of the objects themselves, not of the search structure or of data distribution. Also, as discussed earlier, only z-order based search structures, of which the zkd b-tree is one, can control redundancy.

The graphs show in figures 7 and 8 show how redundancy varies with object size and decomposition strategy. Redundancy is measured by (number of elements) / (number of data objects).

figure 7

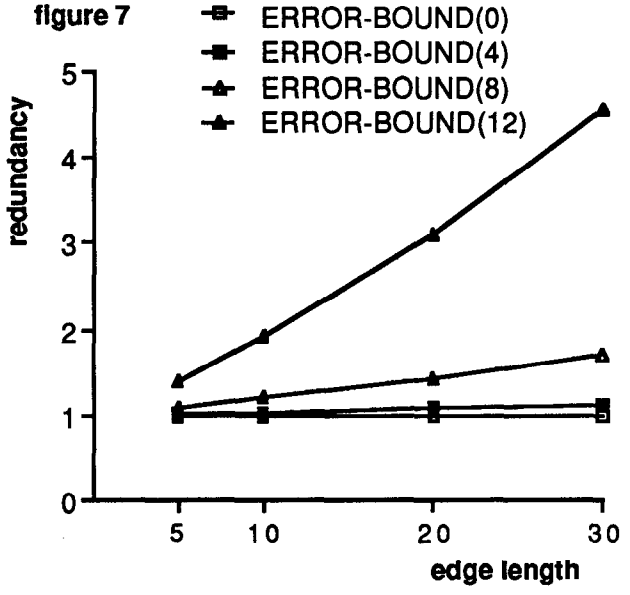
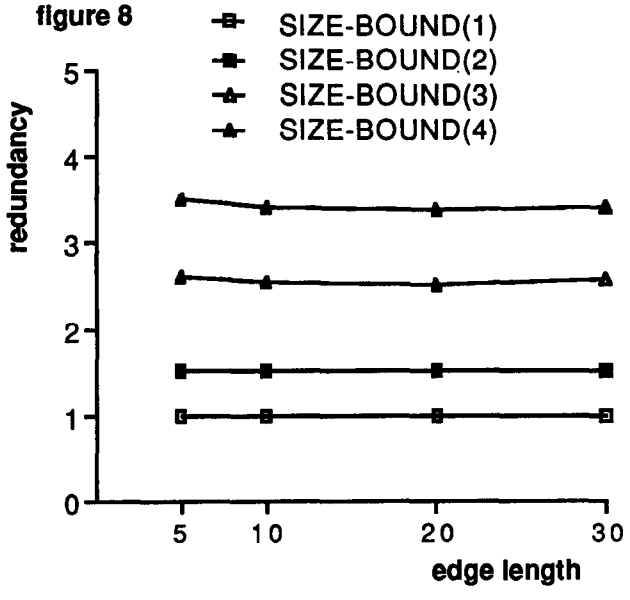


figure 8



With the ERROR-BOUND(g) strategy, redundancy increases with object size. This occurs because, as objects get larger, more elements are needed to accurately represent the object at its boundary, (the interior is usually covered by a few large elements). As g increases, a larger number of smaller elements are needed to represent the data object with the required accuracy, resulting in increased redundancy. (Recall that g is the number of splits forming each partition that represents the permissible error. Each additional split halves the size of the

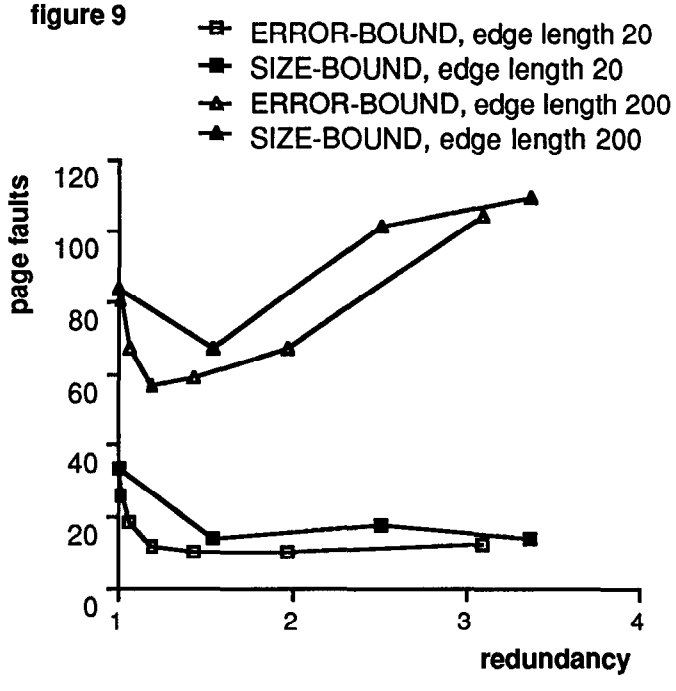
resulting partition.)

The redundancy of SIZE-BOUND(n) is independent of object size, because the factor controlling redundancy is the number of elements generated by the decomposition, regardless of precision. The redundancy observed is always less than n due to the merging of adjacent elements, formed by two way splits, (see section 4.2.2).

6.2 Page faults

Figure 9 shows how page faults vary with redundancy. The graph shows the results for data objects of size 20 x 20, queries of size 20 x 20 and 200 x 200, and for the SIZE-BOUND(n) and ERROR-BOUND(g) strategies. Here, and in all graphs, "edge length" refers to the size of the query. For each strategy, a range of parameter values (n and g) are examined to obtain the redundancy values. The n values used are {1, 2, 3, 4}, and the g values are {0, 2, 4, 6, 8, 10}. The full set of results obtained (for other data and query sizes), cannot be presented due to space limitations. The results presented are typical, and the trends discussed below were observed in all experiments. (Complete results can be obtained from the author.)

figure 9



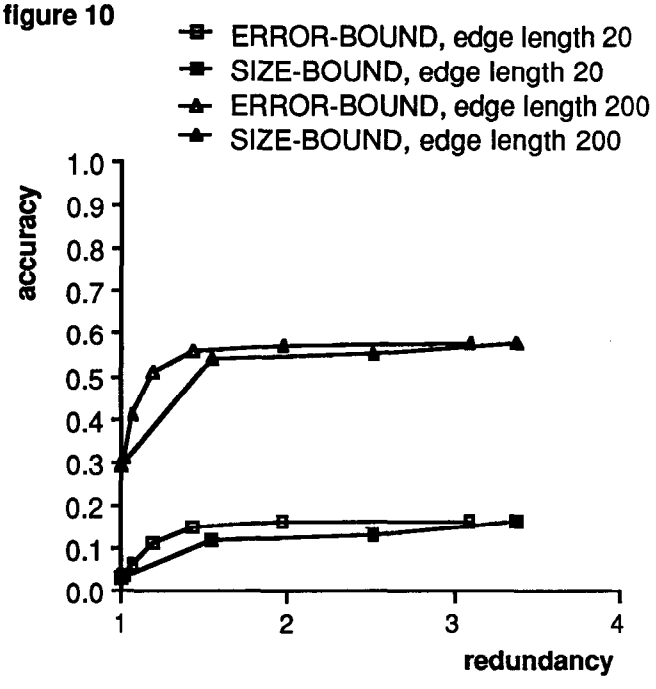
The shape of the curves suggest that the reasoning of sections 4 and 5 is valid. The existence of R_{opt} is apparent in the graph shown (and in all graphs obtained). Performance, as measured by page faults, improves rapidly as redundancy is increased from its minimal value of 1, reaching a minimum at R_{opt} which, for the experiments conducted, varied from about 1.3 to 1.7, for both the SIZE-BOUND and ERROR-BOUND strategies. After R_{opt}, the degeneration of performance is more gradual and almost non-existent for the smaller data sizes. It is interesting to note that, as far as page faults go, optimal performance is obtained with a modest amount of redundancy.

ERROR-BOUND appears to be the superior strategy. It almost always beats SIZE-BOUND for a given amount of redundancy. ERROR-BOUND is also more amenable to fine-tuning. This can be seen by noting that small values of g, (the ERROR-BOUND parameter), result in redundancy values that are close together. This is especially important since it provides fine control where it is needed most, near R_{opt}. SIZE-BOUND provides less control near R_{opt}. It might be possible to obtain finer control by permitting the SIZE-BOUND parameter to vary, e.g. based on the volume of the object, or even randomly. For example, an overall redundancy of 1.5 could be obtained by setting the SIZE-BOUND parameter to be 1 or 2 with equal probability.

6.3 Accuracy

Figure 10 shows how accuracy of the filter step varies with redundancy, using the same experiment parameters as in section 6.2. As with the results on page faults, the results presented here are typical of all experimental results.

figure 10



Accuracy increases rapidly with redundancy, levelling off rapidly at low redundancy. This is fortunate — at low values of redundancy, not only are page faults minimized (at R_{opt}), but accuracy is almost always very close to its maximum.

ERROR-BOUND shows a small but consistent advantage in accuracy over the SIZE-BOUND strategy. In some experiments, SIZE-BOUND was very slightly better.

6.4 Results for VLSI data

In the experiments described above, the data objects in each experiment were all of the same size, and the locations were uniformly distributed. These assumptions are highly unrealistic, so nothing can be claimed about the robustness of the results. Another set of experiments was carried out to see how well the results of sections 6.1 - 6.3 apply in one application. A set of 4741 boxes was obtained from a VLSI circuit design. The data was loaded into zkd b-trees as described

above. X and Y coordinates ranged from 0 to 361, and 0 to 368, respectively. All coordinates were doubled, to better fill the 1000 x 1000 space. (This is fair since it is reasonable to expect the bounds of the space to be known in a VLSI application.) Query positions were randomly generated as before, but the region of $x > 800$ and $y > 800$ was avoided, as queries in these empty areas would artificially lower the search times.

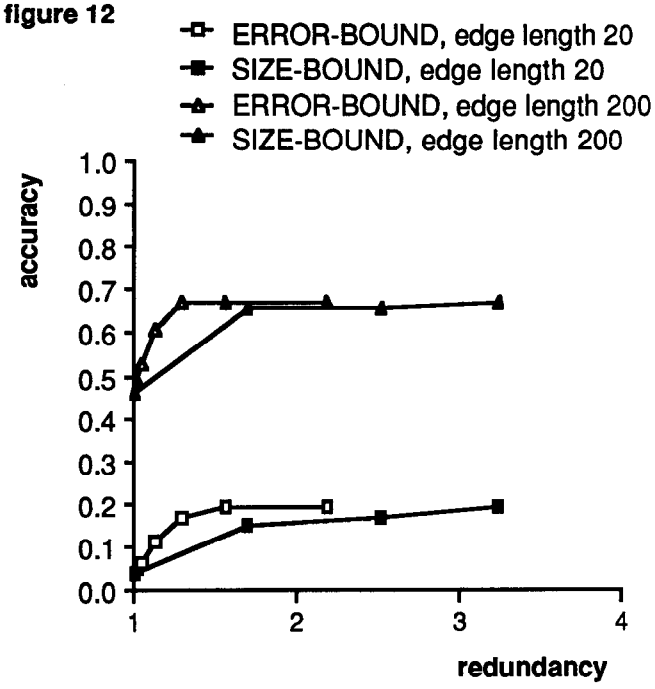
The data objects were all rectangles. About 70% of the boxes were approximately square (ratio of side lengths < 2). Most of the remaining boxes were 2-4 times as tall as they were wide. The average box size was 165 pixels (corresponding to an edge length of approximately 13), but this was severely skewed by a single very large box of 285,621 pixels (edge length 534). Without this datum, the average box size was about 100 pixels, corresponding to edge length 10. For this reason, the results from the VLSI data set are compared to results obtained for the uniformly generated 10 x 10 boxes.

The box sizes were distributed as follows: 14% of the boxes covered 90-99 pixels, 21% covered 30-39 pixels, and another 25% fell between these two ranges. Ignoring the one very large datum, the remaining 40% ranged up to 1180 pixels in size.

To examine the distribution of object locations, the space containing the data was divided into a 16 x 16 grid, and the number of object mid-points per cell was counted. The distribution was fairly uniform except for a large number of relatively sparse cells (14% had less than 7 object mid-points). These sparsely filled cells, most of which were empty, occurred near the edges of the space.

Time requirements for the filter step followed the trends discussed in section 6.2, (see figure 11). R_{opt} is in the range reported earlier. Results for accuracy are similar to what was observed in section 6.3, (see figure 12).

figure 12



The results for VLSI data and the generated boxes with edge size 10 compared as follows. The page faults for VLSI data are higher, possibly due to the fact that the objects are not square, but, for the most part, taller than wider. However, the accuracy obtained for the VLSI data is *higher*. (See figures 13 and 14. Due to space limitations, only ERROR-BOUND results are shown.) Space requirements for the two data sets were very similar.

figure 11

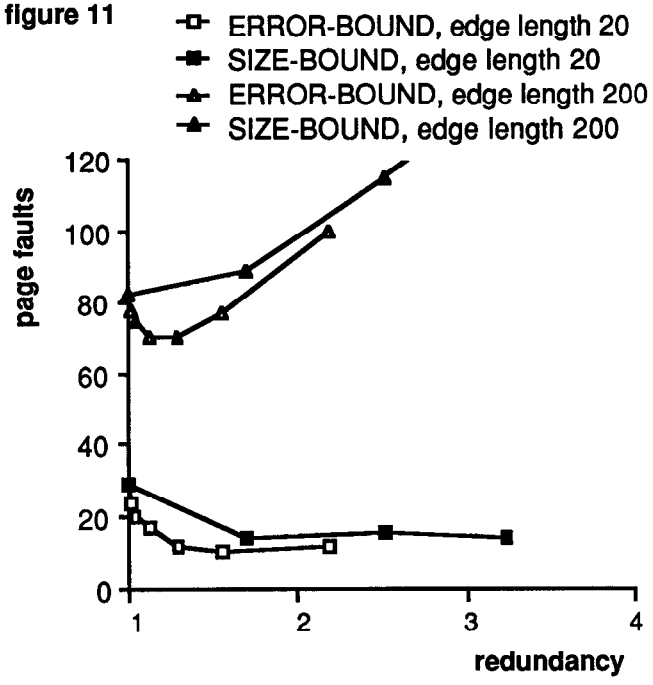


figure 13

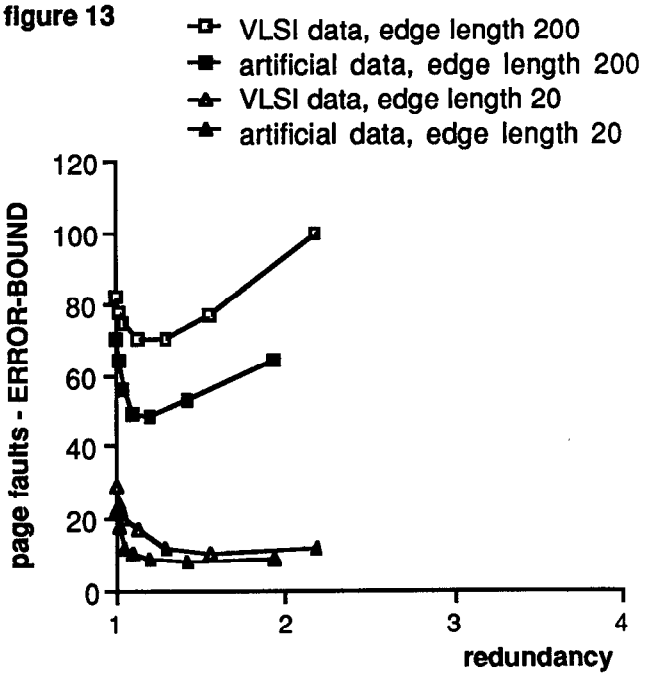
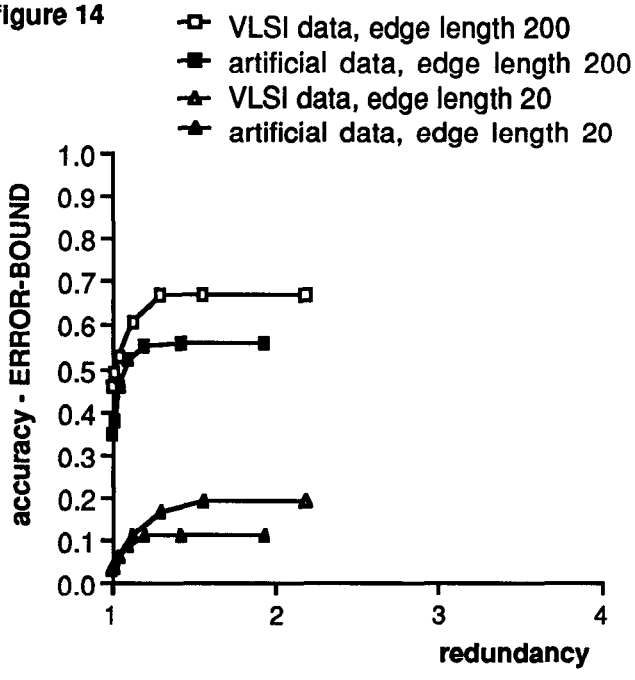


figure 14



7 Conclusion and future work

For the data sets studied, it appears that small amounts of redundancy (between 30% and 70% depending on the data set), provide the best overall results. The page faults required by the filter step drop rapidly as redundancy is increased from 1 to its optimal value. Page faults increase more slowly as redundancy is increased further. Accuracy increases rapidly to near-optimal values. At the point where page faults are minimized, accuracy is almost always very close to optimal. For both criteria, (page faults and accuracy) ERROR-BOUND is almost always superior to SIZE-BOUND, although the difference is sometimes very small. However, the differences are most pronounced at the critical low redundancy values.

These results can be tied to specific ERROR-BOUND and SIZE-BOUND parameter values — ERROR-BOUND(8) and SIZE-BOUND(1) or SIZE-BOUND(2) provide the best results overall.

The results obtained with VLSI data are highly consistent with the results obtained for generated data of comparable size. The main difference was a more costly but more accurate filter step.

Much more work is needed to see what happens in situations not explored by these experiments. In particular, with larger, or more irregularly-shaped data objects, how well will the results reported here hold up?

In all experiments, there was a single query object. The more general case of the overlap query, in which both inputs have many objects has to be examined. Finally, of the four query processing strategies outlined in section 3.3, only one has been examined, strategy 3 in which partitioning is only along element boundaries. Nothing is yet known about the benefits of decomposition based on object structure, or the performance trade-offs in working in (higher-dimension) parameter spaces.

Acknowledgements

I am grateful to Prof. Timos Sellis of the University of Maryland for providing the VLSI data used in section 6.4. Thanks also to Audrey Hart for assistance in debugging the software and proof-reading the paper.

References

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

DAYA87 U. Dayal, M. DeWitt, D. Goldhirsch, J. Orenstein.
PROBE final report.
Technical report CCA-87-02, Xerox Advanced Information Technology Division (formerly Computer Corporation of America).

FALO87 C. Faloutsos, W. Rego.
A grid file for spatial objects.
Technical report CS-TR-1829, Department of Computer Science, University of Maryland, College Park, (1987).

GUTT84 A. Guttman.
R-trees: a dynamic index structure for spatial searching.
Proc. ACM SIGMOD, (1984).

HINR85 K. H. Hinrichs.
The grid file system: implementation and case studies of applications.
Doctoral dissertation, ETH Nr. 7734, Swiss Federal Institute of Technology, Zurich, Switzerland, (1985).

HORN87 D. Horn, H.-J. Schek, W. Waterfeld, A. Wolf.
Spatial access paths and physical clustering in a low-level geo-database system.
Technical report, Technical University of Darmstadt, West Germany (1987).

LOME87 D. B. Lomet, B. Salzberg.
The hB-tree: a robust multi-attribute indexing method.
Technical report TR-87-05, Wang Institute of Graduate Studies, (1987). (The Wang Institute is defunct. Contact Lomet at DEC, Spitbrook or Salzberg at Northeastern University.)

MERR78 T. H. Merrett.
Multidimensional paging for efficient database querying.
Proc. Int'l Conference on 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).

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

OREN83 J. A. Orenstein.
Algorithms and data structures for the

implementation of a relational database system. Ph. D. thesis, McGill University, School of Computer Science (1983).

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.

OREN86a J. A. Orenstein.
Spatial query processing in an object-oriented database system.
Proc. ACM SIGMOD, (1986).

OREN86b J. A. Orenstein, F. A. Manola.
Spatial data modeling and query processing in PROBE.
Technical report CCA-86-05, Xerox Advanced Information Technology Division (formerly Computer Corporation of America).

OREN88 J. A. Orenstein, F. A. Manola.
PROBE spatial data modeling and query processing in an image database application.
IEEE Trans. on Software Eng. 14, 5 (May, 1988) 611-629.

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

SAME84 H. Samet.
The quadtree and related hierarchical data structures.
ACM Comp. Surv. 16, 2 (1984).

SELL87 T. Sellis, N. Roussopoulos, C. Faloutsos.
The R+-tree: a dynamic index for multi-dimensional objects.
Proc. VLDB, (1987).

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