TR--89--001

DYNAMIC BUCKETING :
A DATA STRUCTURE FOR FAST
REGION SEARCHES

中研院資訊所圖書室
3 0330 03 000111 4

中央研究院
資訊科學研究所
78.10.-5
圖書室

0111

DYNAMIC BUCKETING: A DATA STRUCTURE FOR FAST REGION SEARCHES

Y. S. Kuo[*], San–Yih Hwang[**] and H. F. Hu[*]

[*]Institute of Information Science
Academia Sinica

[**]Dept. of Computer Science and Information Engineering
National Taiwan University

Address for correspondence:

Dr. Y. S. Kuo
Institute of Information Science
Academia Sinica
Nankang, Taipei, Taiwan
R. O. C.

Abstract

A region query finds all objects intersecting a specified region, usually a rectangular window. Bucketing, also known as fixed cells, is a data structure especially suitable for small window queries. The bucketing structure is enhanced in two ways: It is made dynamic by maintaining a two–level directory structure. By using the technique of extendible hashing, the directories can grow gracefully as more and more objects are inserted into the structure. Dynamic bucketing is further enhanced by employing the quad list structure in order to improve the performance of large window queries.

1. Introduction

Many algorithms in computer–aided design (CAD) applications require region queries of the objects in a plane. A region query finds all objects intersecting a specified region, usually a rectangular window. With the advent of VLSI circuits and the fine–line PCB technology, such search problems may involve tens of thousands of objects. Thus a data structuring technique that organizes these objects appropriately in order to support fast region searches is very crucial.

The quad tree was originally proposed for the retrieval of points in a specified query window [10]. Kedem extended this structure to allow general objects such as rectangles and polygons [11]. Brown improved the quad tree structure by storing some objects in more than one quad [5]. Recently, Weyten & de Pauw refined the quad tree by introducing a quad list structure, making the quad tree especially efficient for large window queries [19]. The multidimensional binary tree (or k–d tree) was originally developed as a search mechanism for multi–key files [2]. Lauther adopted the k–d tree as a structure for fast region searches [12]. However, it has been reported that k–d trees use significantly more storage than quad trees [16].

Both quad trees and k–d trees use tree structures for searching. The query speed is thus affected by the number of objects in the 2–dimensional space (2–space). Bucketing [1][8], also known as fixed cells [3], is a data structure that supports the direct access of objects in 2–space. In this scheme, the 2–space is partitioned into equal–sized rectangular regions each of which has an associated bucket containing all objects intersecting the region. Given a query window, it is then straightforward to determine the regions intersecting the window. And the search time for a small window is constant, independent of the number of objects in the 2–space. Edahiro improved the bucketing structure by incorporating the window list structure from filtering search [6][7].

Different from all the schemes mentioned previously, corner stitching is a specialized data structure suitable for not only region searches but also the report of nearest neighbors [15]. However, it has been argued that corner stitching has great difficulties with overlapping rectangles which severely limits its usefulness [16].

Bucketing has the advantage of direct access. However, it has the drawback that it is not dynamic. Thus, we propose a dynamic bucketing scheme for on–line region searches. The 2–space is partitioned into different–sized regions, thus more adaptable to nonuniform distribution of certain objects. Two levels of directories are maintained, reflecting the partition of the 2–space into regions. The directories will grow gracefully as more and more objects are inserted into the buckets. This dynamic directory structure has its root from extendible hashing, a technique originally proposed for fast retrieval of dynamic data in 1–space [9]. Like the original bucketing structure, dynamic bucketing can also support the direct access to objects intersecting a query window.

The proposed dynamic bucketing scheme is further enhanced by employing the quad list structure [19]. Bucketing is especially appropriate for small window queries. The quad list structure is very effective in improving the performance of large window queries. The combination is thus a data-structure efficient for both small and large window queries.

We have implemented the dynamic bucketing and the well known quad tree algorithms. Many experiments have been performed in order to compare the performances of the two structures. Test results from these experiments have been in favor of dynamic bucketing.

2. Background

First of all, let us make clear certain assumptions with regard to the region query problem. These assumptions are valid in most CAD applications [4] and, possibly, in some others as well.

1. All objects and query windows are rectangles. In other words, if an object is not a rectangle, then it is represented by its smallest enclosing rectangle.
2. All objects and query windows fall within a large rectangular region, the 2–space.
3. Without loss of generality, we assume that the 2–space, objects and query windows all have nonnegative integers as their coordinates.
4. The objects are quite small compared with the entire 2–space.
5. The objects are almost uniformly distributed over the entire 2–space.
6. Region queries occur more frequently than insertions. Thus the query speed is more important than the insertion time.
7. Insertions occur more frequently than deletions. Thus we can ignore the influence of deletions on the performance of the technique.

For the rest of this section, we briefly review the directory structure of extendible hashing [9].

Assume that the objects are points in 1–space; each object has an integer as its coordinate. A directory is organized as a linear array having 2^d entries where d is called the depth of the directory. Each entry in the directory has a pointer to a bucket. A bucket can contain at most T objects, where T is a threshold number chosen for efficiency. To locate an object, we use the first d bits of its coordinate as an index to the directory. Then the pointer in the appropriate directory entry is followed to retrieve the bucket containing the object. Fig. 1 shows a directory structure when d=3. As shown in this figure, each bucket has a header that contains its local depth $d' \leq d$. For a bucket with local depth d', there are $2^{(d-d')}$ directory entries pointing to it.

When inserting an object into a full bucket (a bucket containing T objects) with local depth d', $d'<d$, we execute a bucket split: We split the bucket into two buckets with local depth d'+1, distribute the objects between the two buckets according to their coordinates, and then change the pointers in the appropriate directory entries. As an example, for the directory structure in Fig. 1, the bucket with local depth 1 can be split into two buckets as shown in Fig. 2.

When inserting an object into a full bucket with local depth $d'=d$, we first execute a directory doubling: We double the size of the directory; each directory entry becomes two adjacent directory entries with pointers to the same bucket; directory depth d is then increased by one. Now a bucket split to the overflow bucket can be executed similarly as before. For the example in Fig. 2, if the bucket pointed to by the 010 pointer overflows, a directory doubling will be executed followed by a bucket split as shown in Fig. 3.

An object can be deleted from the structure in a way opposite to the insertion of an object; buckets can be merged or the directory can be halved whenever it is appropriate.

3. Dynamic Bucketing

In dynamic bucketing, two levels of directories are used for structuring the objects in 2–space. The top level directory called the horizontal directory represents a partition of the 2–space into vertical strips. Each entry in the horizontal directory has a pointer to a vertical directory. There is a one–to–one correspondence between the vertical directories and the vertical strips in 2–space. Each entry in a vertical directory has a pointer to a bucket. Correspondingly, each vertical strip is partitioned into regions, and there is a one–to–one correspondence between the buckets and the regions. A typical structure of dynamic bucketing is shown in Fig. 4. Its associated partition of 2–space into regions is shown in Fig. 5. Note that there are 3 vertical directories; thus the 2–space is partitioned into 3 vertical strips. These vertical strips are further divided into different–sized regions associated with the buckets.

In this structure, all objects are stored in buckets. A bucket, associated with a region, contains all objects that intersect the region. Since an object may intersect more than one region, some objects will be stored in more than one bucket. To save the memory space used by these duplicate objects, we store pointers to the object rather than the object itself in the buckets. This multiple storage strategy has been used in the quad tree structure [5].

In dynamic bucketing, the horizontal directory and each of the vertical directories are organized like a directory for extendible hashing. Their data structures are demonstrated as follows:

The horizontal directory, h_dir, consists of fields h_dir.depth, h_dir.entry[0],..., h_dir.entry[h–1], where $h = 2^{h\_dir.depth}$. h_dir.depth is the directory depth of h_dir. Each h_dir.entry [i], i = 0,1,...,h–1, contains a pointer to a vertical directory.

A vertical directory, v_dir, consists of fields v_dir.local, v_dir.depth, v_dir.entry[0],..., v_dir.entry[v–1], where $v = 2^{v\_dir.depth}$. v_dir.local is v_dir's local depth with respect to h_dir. v_dir.depth is the directory depth of v_dir. Each v_dir.entry[j], j = 0,1,...,v–1, contains a pointer to a bucket.

A bucket, bkt, consists of fields bkt.local, bkt.count, and a list of object pointers. bkt.local is bkt's local depth with respect to a vertical directory. bkt.count indicates the number of object pointers stored in bkt.

Dynamic bucketing can be considered as a generalization of extendible hashing to the 2–space. All algorithms sketched in Section 2 thus can be extended for structuring the directories of dynamic bucketing.

To answer a region query, it is necessary to find all regions intersecting a given window. This can be done with two loops. The outer loop finds all vertical strips intersecting the window. For each intersecting vertical strip, the inner loop then finds the regions intersecting the window. We use the first h_dir.depth (resp. v_dir.depth) bits of the window's x (resp. y) coordinates as the lower and upper bounds to the outer (resp. inner) loop. The vertical directories (resp. buckets) are accessed by stepping through pointers h_dir.entry[i] (resp. v_dir.entry[j]). To avoid accessing a vertical directory (resp. bucket) more than once, the index i (resp. j) is each time incremented by t, where $t=2^{h\_dir.depth-v\_dir.local}$ (resp. $t=2^{v\_dir.depth-bkt.local}$).

The above nested loops can be executed very efficiently. For a small query window which intersects a constant number of regions, the search time is constant since only a constant number of directory entries are examined. For example, exactly two directory entries are examined for a point serach.

When inserting an object, we use the same outer and inner loops just described to locate the regions the object intersects. A pointer to the object is then stored in each of the buckets associated with these regions. If some of these buckets are full, we must execute certain operations such as bucket split and directory doubling in Section 2 to make room for the new object. Now there are a bucket split and two directory doubling operations: vertical directory doubling and horizontal directory doubling. In addition, there is a vertical directory split in analogy with a bucket split. When executing a vertical directory split, a vertical directory is split into two; each bucket linked to the vertical directory is split into two buckets; each directory entry becomes two directory entries each in one directory and having a pointer to a split bucket. For the structure in Fig. 4, a vertical directory split is illustrated in Fig. 6.

Let us consider the directory update strategy upon insertion. Assume that a bucket, bkt, has been located; bkt is pointed to by pointer v_dir.entry[j] in vertical directory v_dir; and v_dir is pointed to by pointer h_dir.entry[i] in the horizontal directory h_dir. If bkt is full, we use the following procedure to create buckets for new objects.

```
if v_dir.depth > bkt.local then
    execute a bucket split
else if h_dir.depth > v_dir.local then
    execute a vertical directory split
else let region_ratio be the aspect ratio (height/width) of the
```

```
        region associated with bkt, and
    let object_ratio be the average aspect ratio (total_height/total_width)
        of the objects in the bucket.
    if region_ratio ≥ object_ratio then
         execute a vertical directory doubling and then a bucket split
    else execute a horizontal directory doubling and then a vertical directory split
```

Using the above procedure, the regions in the 2–space tend to be so partitioned to have approximately equal aspect ratios as the majority of objects intersecting them. This heuristic has the effect of minimizing the number of objects that intersect more than one region.

Since deletions occur infrequently, the deletion of an object is simply done by removing all its pointers in the buckets without merging buckets or halving the directories.

4. Some Refinements

As described in the last section, some objects in the bucketing structure are duplicated in more than one bucket. This results in the inefficiency of inspecting an object more than once on one region research. The quad list structure is employed to avoid this inefficiency [19]. The directory structure of dynamic bucketing remains the same as before, but each bucket is refined to contain four lists rather than one.

A bucket, bkt, contains fields bkt.local and bkt.count as before, and four lists of object pointers,namely, lists 0, 1, 2, and 3. An object intersecting the region associated with a bucket is stored in one of the four lists depending on the following two tests: whether the object crosses the lower (resp. left) boundary of the region or not. The classification is made according to Table 1.

When answering a region query, the same procedure as in Section 3 can be used to locate all regions intersecting the given window. For each intersecting region, some of the four associated bucket lists have to be inspected depending on whether the window's lower (resp. left) boundary crosses the region or not. Table 2 shows the lists to be inspected in different cases. In this way, each object intersecting the query window is guaranteed to be reported exactly once [19].

A problem that may arise with dynamic bucketing is the nonlinear growth rate of the directory. The directory size of extendible hashing is quite sensitive to the existence of clusters of objects. Tamminen has proposed a refinement of extendible hashing which has linear average storage utilization [18]. Using this scheme, a lower bound is imposed to the size of a region. When a region reaches this lower bound, it can not be subdivided, and overflow buckets are created. This refinement also solves the problem that more than T objects may have the same coordinates (T = threshold).

The 2–space is the maximum space that can possibly be used for layout design. For a specific layout, it is quite possible that only a portion of the 2–space is actually used. When this happens, those directory entries of dynamic bucketing outside certain ranges will all point to empty buckets or empty vertical directories. The memory space for the directories can be saved by recording the ranges and omitting those directory entries outside the ranges.

5. Test Results

We have coded the dynamic bucketing and the quad tree algorithms [19] in C running on SUN 3/60. Many experiments have been performed in order to compare the performances of the two structures. For each experiment, the sizes and locations of objects are generated by a random number generator, and all objects are generated inside a fixed 2–space bounded by vertices (0,0) and (32767,32767). For example, for w=h=250±125, the width and the height of an object are random numbers generated between 125 and 375. We consider query windows of two different sizes: a large window has its width and height equal to 16000; and a small window has the same dimensions as the average dimensions of an object. For each experiment, 100 query windows of each size are generated randomly at 100 positions. The reported results are averages taken from the 100 windows.

We use the following notation.

N = total number of objects inserted

T = threshold = number of object pointers a bucket can hold

b = average number of object pointers a bucket actually holds

load factor = b/T

duplicate factor = (number of object pointers stored)/N

Fig. 7 illustrates the memory space used by dynamic bucketing and the quad tree. Dynamic bucketing uses slightly more (8%) memory space than the quad tree when the average number of objects in a bucket is relatively small. It should be pointed out that in these tests the buckets (leaf quads) were implemented as dynamically allocated linked lists. One may want to implement the buckets by allocating a fixed–sized (depending on T) consecutive piece of storage for some system's considerations (eg. database, page fault). When the buckets are allocated consecutive pieces of storage (4–byte pointers linking the quad lists are replaced by one–byte offsets.), dynamic bucketing uses less memory space than the quad tree since the former has a higher load factor than the latter. The average load factor of dynamic bucketing is 0.64 while that of the quad tree is 0.52.

When answering region queries, the dynamic bucketing and the quad tree algorithms are equally efficient in terms of the numbers of objects they examine. This is shown in Fig. 8 for large window queries. (Small window queries also demonstrate similar results.) More objects than those really intersecting the query window must be examined. This is due to the boundary effect (regions cross the window boundary may contain objects disjoint with the window) rather than the duplication of objects. Fig. 9 shows the numbers of directory entries and tree nodes examined by the dynamic bucketing and the quad tree algorithms for small query windows. It should be noted that since more operations must be executed at each tree node (containing four pointers) of the quad tree than at each directory entry (containing one pointer) of dynamic bucketing, the difference between the two methods is more than what Fig. 9 indicates. Since for small window queries, the query speed is, to some extent, determined by the search time, dynamic bucketing is particularly appropriate for small window queries.

All test results presented so far have N = 20000 even though many other values have also been tested. We have made the following observation: When N and b increase or decrease proportionally, the storage utilization of dynamic bucketing remains almost unchanged. For example, the test case N = 20000, b = 20 and the test case N = 40000, b = 40 showed approximately equal duplicate factors. The performance behavior regarding the query speed is not so regular. Test results for different values of N have shown similar curves as those in Fig. 8 and 9, but at a different scale.

VLSI layouts usually contain many long narrow wires. Thus, it is important for a data structure to be able to handle long narrow objects efficiently. Fig. 10 shows the memory space used by dynamic bucketing and the quad tree when N = 20000, w = 80$\pm$20, h = 780$\pm$300. The space efficiency of the quad tree degrades apparently while dynamic bucketing still maintains an equal storage utilization as the case of square–shaped objects (with reference to Fig. 7). Due to the directory update heuristic described in Section 3, dynamic bucketing can handle long narrow objects very efficiently. This suggests that horizontal and vertical wires in a layout should be stored in two separate bucketing structures.

6. Discussion

The original bucketing structure is not efficient in storage utilization when there are many long narrow objects such as wires. (The quad tree does not seem to be efficient either.) Edahiro adopted a special structure, the window list, to handle these long narrow objects [7]. We have presented a heuristic approach to this problem. The partitioning of the 2–space is made according to the shapes of the objects. The same effect as that gained by the window list seems to be achievable by using this simple scheme. In fact, the window list approach has a good theoretical bound in its storage utilization. However, in its implementation each long object must be stored for at least three copies which may severely limit its efficiency in practice.

The region query problem considered in this paper is related to the multikey file organization problem. The grid file [13] and multidimensional extendible hashing [14] [17] are structures for the latter problem which share the same spirit as dynamic bucketing. They use k–dim arrays as their directories while dynamic bucketing uses two–levels of 1–dim arrays. When an array is resized (eg. doubling), the entire array has to be inspected. Thus, using a single 2–dim array as directory, the worst–case insertion time is $O(D)$ where D is the number of directory entries. On the other hand, using the two–level directory structure of dynamic bucketing, only 1–dim arrays of size approximately $O(\sqrt{D})$ are to be resized. The time complexity of insertion is thus determined by that of vertical directory split which is $O(b\sqrt{D})$ where b is the average number of objects in each bucket. Since $\sqrt{D}$ is usually much larger than b, we consider the proposed directory structure more appropriate for its use in interactive programs.

References

1. T. Asano, M. Edahiro, H. Imai, M. Iri, and K. Murota, "Practical use of bucketing techniques in computational geometry", Computational Geometry (G. T. Toussaint, ed.) North–Holland, pp.153–195, 1985.
2. J. L. Bentley, "Multidimensional binary search tree used for retrieval on composite keys", Acta Informatica, Vol.4, No.11–9, 1974.
3. J. L. Bentley and J. H. Friedman, "Data structures for range searching", ACM Computing Surveys, vol. 11, no 4, 1979.
4. J. L. Bentley, D. Haken and R. W. Hon, "Statistics on VLSI Designs", Department of Computer Science, CMU, CS–80–111, April 1980.
5. R. L. Brown, "Multiple storage quad trees: a simpler faster alternative to bisector list quad trees", IEEE Trans. on CAD, Vol.CAD–5, No. 3, pp.413–419, July 1986.
6. B. Chazelle, "Filtering search : a new approach to query answering", SIAM J. Comput., vol 15, no 3, August 1986.
7. M. Edahiro, "A new bucketing technique for automatic/interactive layout design", Proc. VLSI, Vancouver, August 1987.
8. M. Edahiro, K. Tanaka, T. Hoshino, and T. Asano, "A bucketing algorithm for the orthogonal segment intersection search problem", Proc. 3rd ACM Annu. Symp. Computational Geometry, pp.258–267, June 1987.
9. R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible hashing – a fast access method for dynamic files", ACM trans. Database Syst., Vol. 4, No.3, pp.315–344, Sept. 1979.
10. R. A. Finkel and J. L. Bentley, "Quad trees – a data structure for retrievals on

composite keys", Acta Informatica, Vol.4, No.1–9, 1974.

11. G. Kedem, "The quad–CIF tree: a data structure for hierarchical on–line algorithms", Proc. 19'th Design Automation Conf., pp. 325–357, June 1982.

12. U. Lauther: "4–dimensional binary search trees as a means to speed up associative searches in design rule verification of integrated circuits", Journal of Design Automation and Fault Tolerant Comuting, 2, 3. July 1978, pp. 241–247.

13. J. Nievergelt, H. Hinterberger, K. C. Sevcik, "The grid file: an adaptable, symmetric multikey file structure", ACM Trans. on Database Systems 9, 1 (1984), 38–71

14. E. J. Otoo, "A mapping function for the directory of a multidimensional extendible hashing", Proc. 10th Int. Conf. VLDB. Singapore, 1984.

15. J. K. Ousterhout, "Corner stitching : a data–structuring technique for VLSI layout tools", IEEE Trans. on Computer–Aided Design, Vol. CAD–3, No. 1, Jan. 1984.

16. J. B. Rosenberg, "Geographical data structures compared study of data structures supporting region queries", IEEE Trans. on CAD, Vol. CAD–4, No.1, Jan. 1985.

17. M. Tamminen, "The extendible cell method for closest point problems", BIT, vol. 22, 1982, pp. 27–41

18. M. Tamminen, "Extendible hashing with overflow", Information Processing Letters, Vol. 15, No. 5, Dec. 1982.

19. L. Weyten and W. de Pauw, "Quad list quad trees: a geometrical data structure with improved performance for large region queries", IEEE Trans. on Computer–Aided Design, Vol. CAD–8, No. 3, March 1989.

Acknowledgement

The anthors are grateful to the referees for their valuable suggestions.

Table 1. List types

object crossing lower boundary	object crossing left boundary	list type
no	no	0
no	yes	1
yes	no	2
yes	yes	3

Table 2. Lists to be inspected

lower boundary crossing region	left boundary crossing region	lists to be inspected
no	no	0
no	yes	0, 1
yes	no	0, 2
yes	yes	0, 1, 2, 3

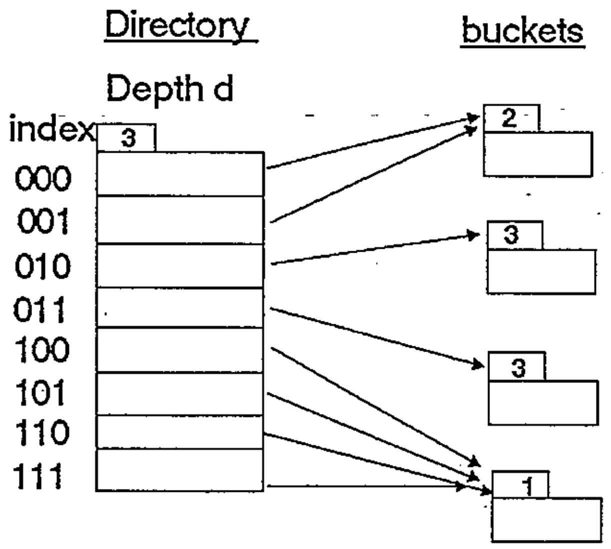

Fig. 1. Directory in 1 – space

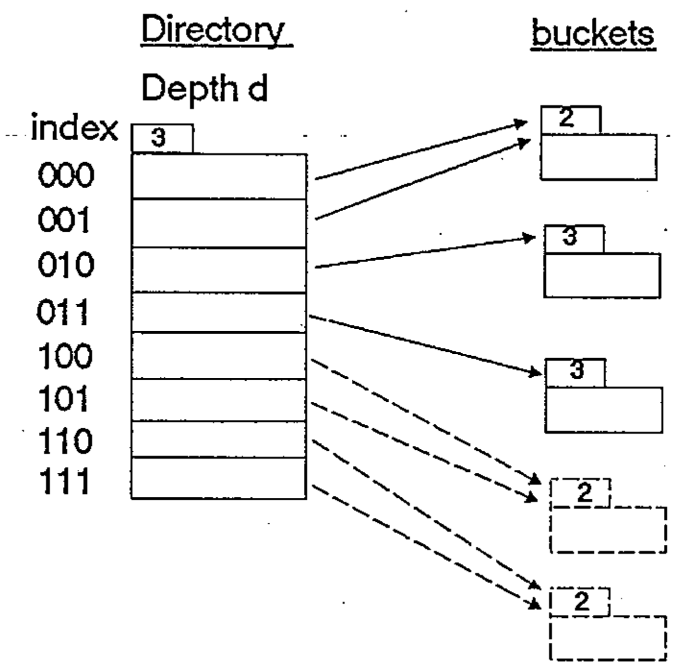

Fig. 2. Bucket split

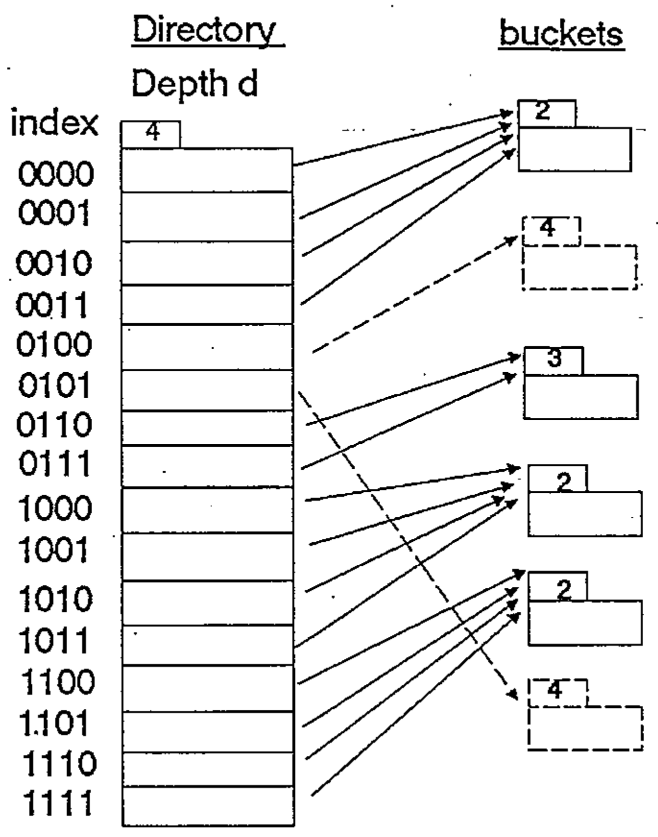

Fig. 3. Directory doubling followed by bucket split

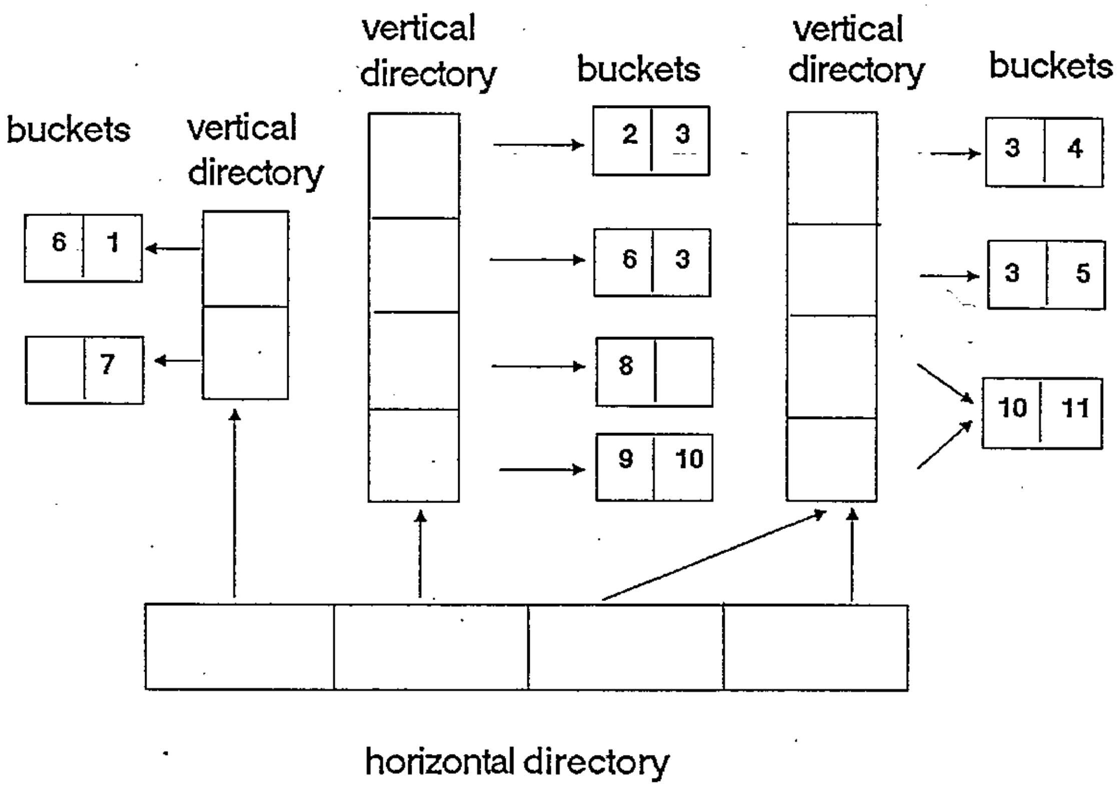

Fig. 4. Storage structure

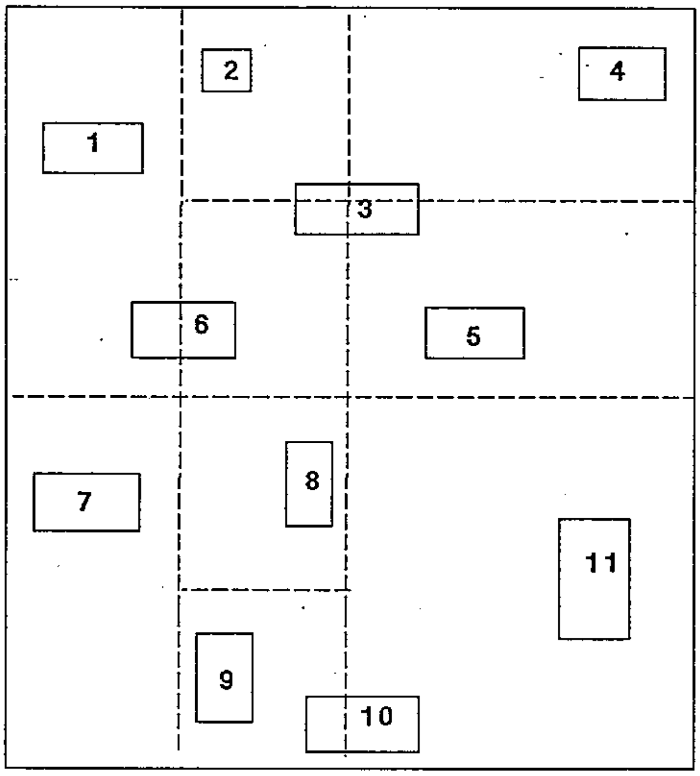

Fig. 5. Partitioned 2 – space

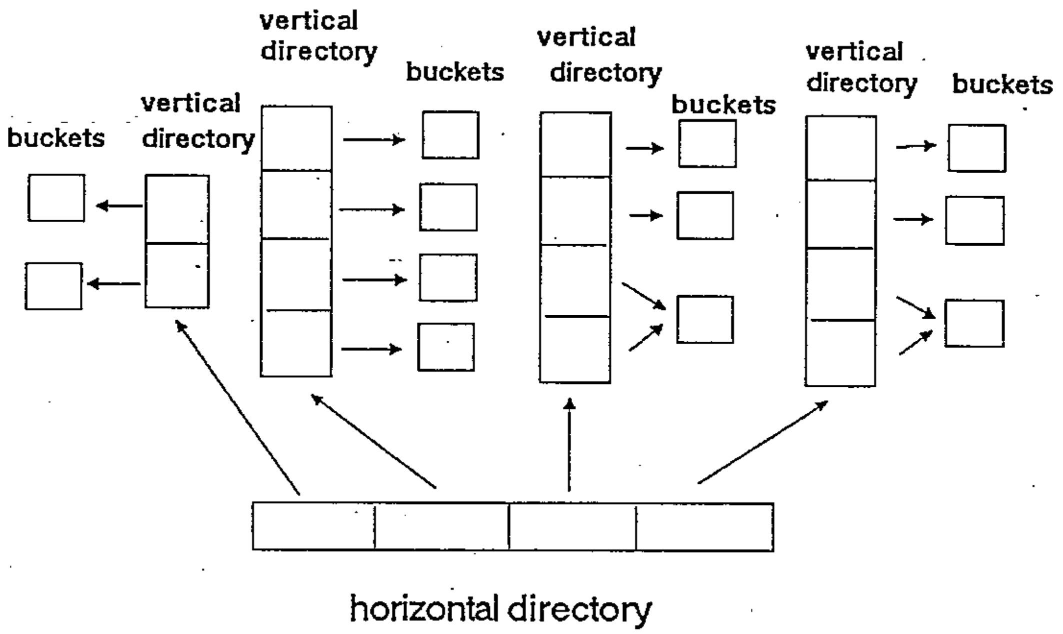

Fig. 6. Vertical directory split

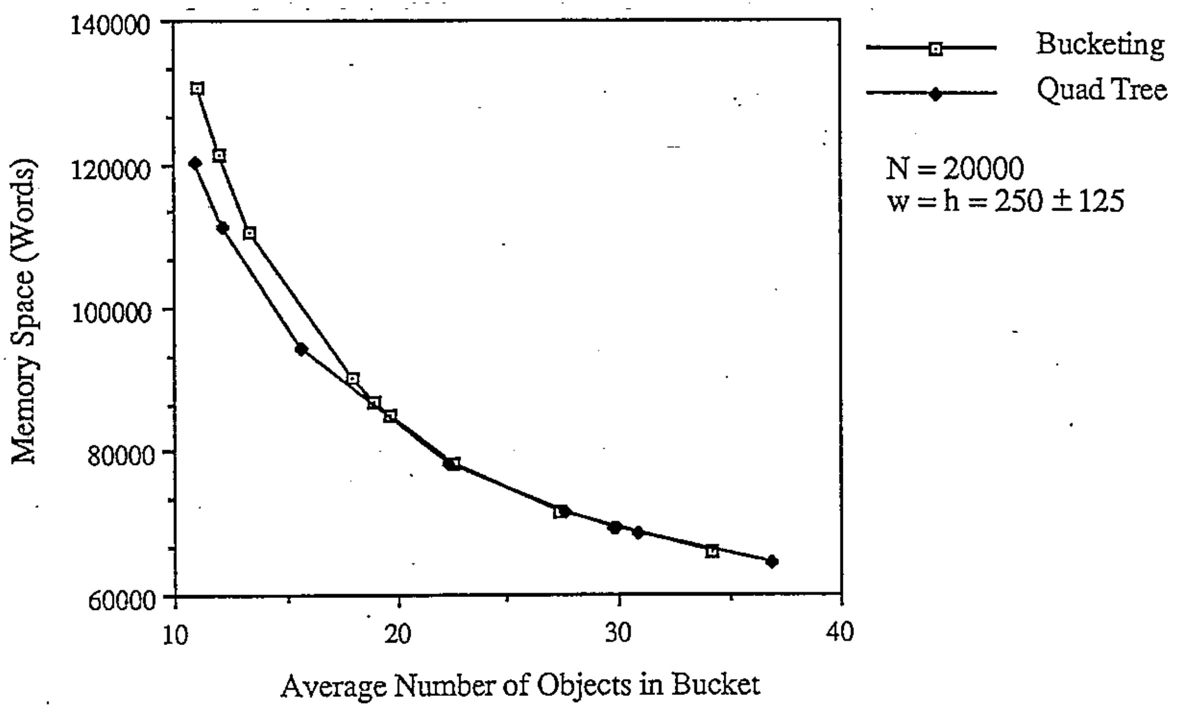

Fig. 7. Storage Utilization

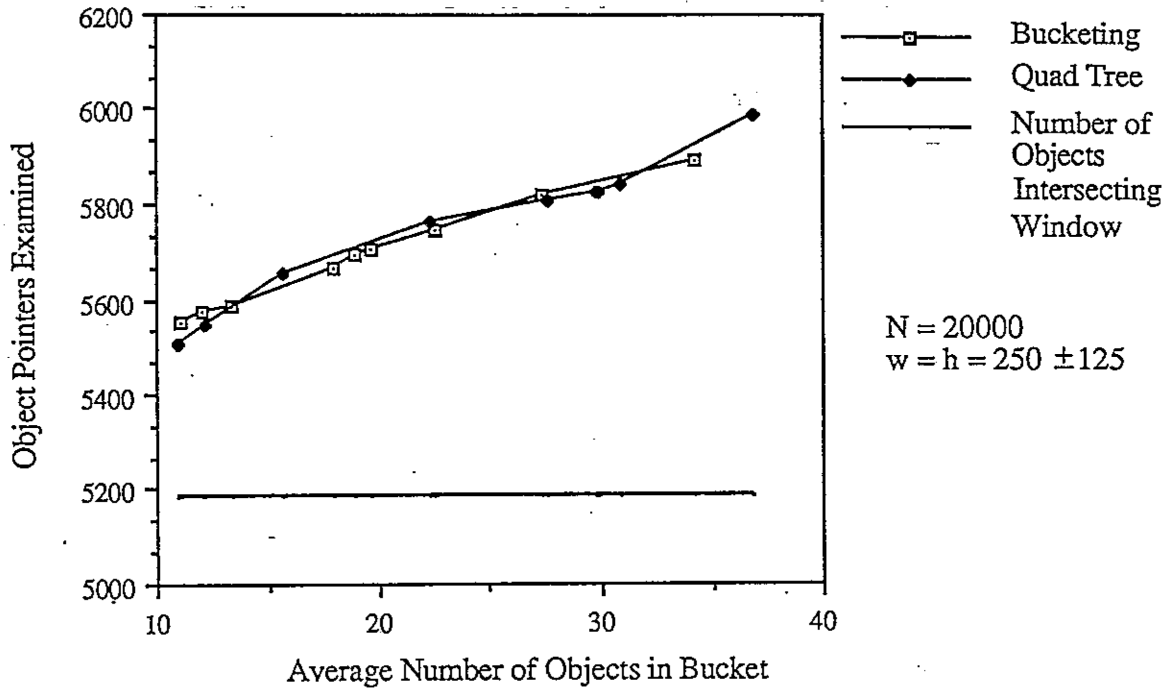

Fig. 8. Number of Object Pointers Examined

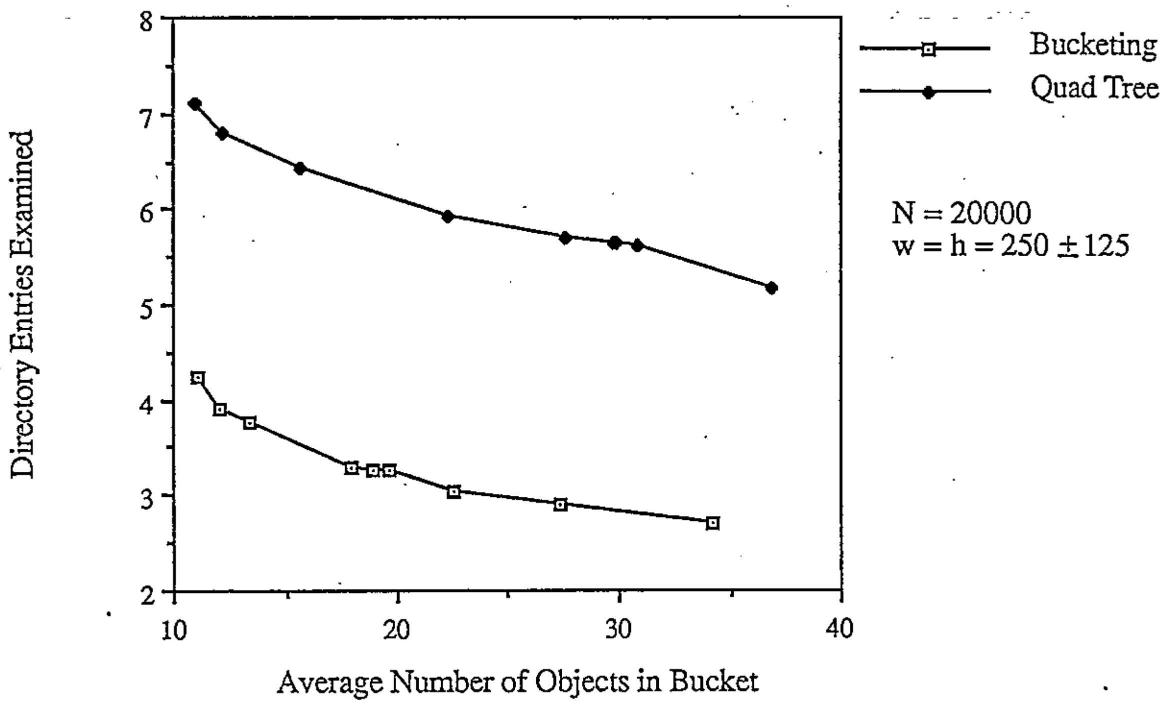

Fig. 9. Number of Directory Entries Examined

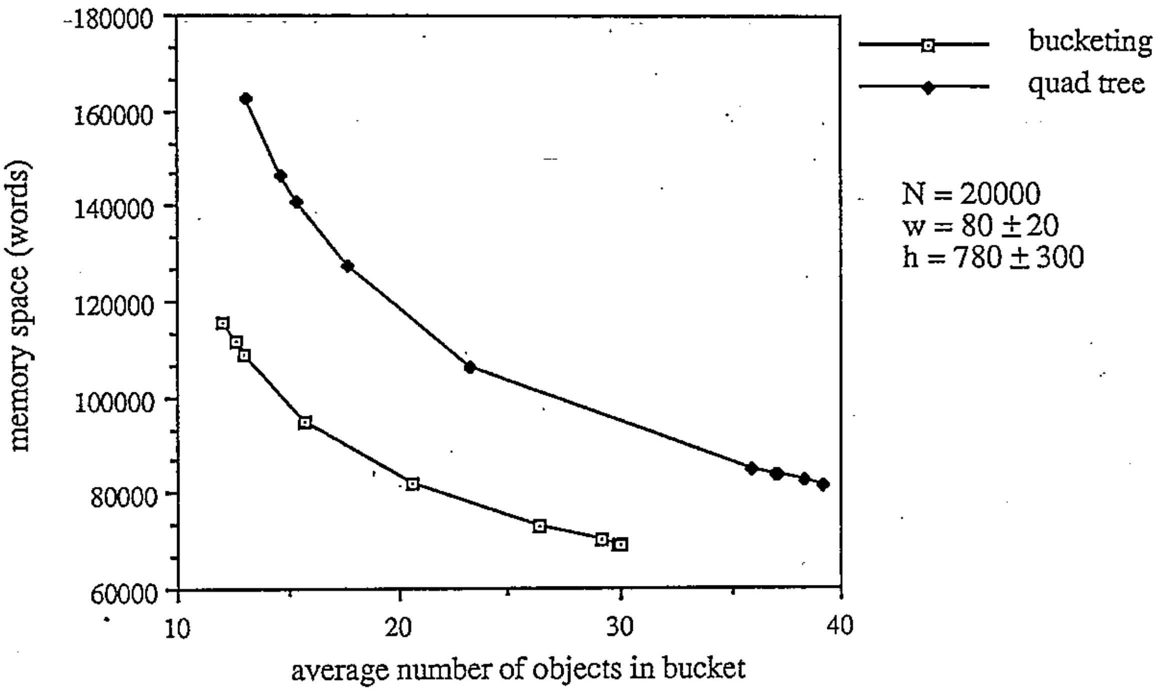

Fig. 10. Storage utilization for long narrow objects