Twin Grid Files: Space Optimizing Access Schemes *

Andreas Hutflesz
Universität Karlsruhe
Postfach 6980
7500 Karlsruhe
West Germany

Hans-Werner Six
FernUniversität Hagen
Postfach 940
5800 Hagen
West Germany

Peter Widmayer
Universität Karlsruhe
Postfach 6980
7500 Karlsruhe
West Germany

Abstract

Storage access schemes for points, supporting spatial searching, usually suffer from an undesirably low storage space utilization. We show how a given set of points can be distributed among two grid files in such a way that storage space utilization is optimal. The optimal twin grid file can be built practically as fast as a standard grid file, i.e., the storage space optimality is obtained at almost no extra cost. We compare the performances of the standard grid file, the optimal static twin grid file, and an efficient dynamic twin grid file, where insertions and deletions trigger the redistribution of points among the two grid files. Twin grid files utilize storage space at roughly 90%, as compared with the 69% of the standard grid file. Typical range queries – the most important spatial search operations – can be answered in twin grid files at least as fast as in the standard grid file.

1 Introduction

In non-standard databases, as used in geographic, engineering, CAD and VLSI applications, access schemes for the efficient manipulation of large sets of geometric objects are needed. In particular, proximity queries must be handled efficiently. Secondary storage access schemes for supporting insertions, deletions, and range queries in a set of multidimensional points have been proposed [4,7,9,10,13,14,16]. The grid file [13] is useful for maintaining geometric data [6,18], because it adapts dynamically to the distribution of points, even if there are clusters.

However, like many other schemes based on recursive halving [12], the grid file suffers from an undesirably low average storage space utilization of roughly 69% only. Especially if storage space is costly, one might wish to increase space utilization. For a static set of data points optimal space utilization is desirable. In some applications, like e.g. geographic databases, point sets are almost static, i.e., few insertions and deletions occur. Starting from a storage space optimal structure, the space efficiency should be maintained dynamically, on a level close to optimal. Reorganizations of the database may bring space utilization back to optimal at certain times. In an interactive setting, where a user queries and updates the database, only limited restructuring operations per insertion and deletion can be tolerated. In contrast, it is usually intolerable to increase the response time for range queries.

As a solution to these problems, we propose *twin grid files*, access schemes that use two grid files [13], both spanning the data space, and that appropriately partition the point set between these two grid files. In Section 2, we describe the general mechanism underlying the twin grid files in more detail. In a twin grid file, storage space utilization is high; experiments show it to be around 90% on the average for two-dimensional points. Range queries can be answered with at least the same efficiency as in the standard grid file, for average ranges. In Section 3, we describe the principle of operation of the dynamic twin grid file, useful to keep space utilization high in a dynamic setting. As compared to the standard grid file, the number of accesses to secondary storage per insertion or deletion increases by a factor of at most 2, depending on the amount of redistribution efforts. We characterize the storage space optimal twin grid file in Section 4. We show how it can be built practically as fast as a standard grid file, i.e., the storage space optimality is obtained at almost no extra cost. We have implemented the optimal static and the dynamic twin grid file and compared their performances with the standard grid file; the performance evaluation is presented in Section 5. Finally, we draw a conclusion in Section 6.

* This work was partially supported by grants Si 374/1 and Wi 810/2 from the Deutsche Forschungsgemeinschaft

2 The basic idea

For the purpose of explaining the twin grid file mechanism, let us restrict our attention in this paper to two-dimensional points. Consider a set of points in the plane to be stored in a grid file. The storage space for each point as well as the bucket size are fixed. The bucket

capacity b is the maximum number of points that can be stored in one bucket; in our example, let $b = 3$. Empty buckets are not stored explicitly on secondary storage; instead, an empty region is represented by a special directory entry, the so-called dummy.

Figure 2.1 (a) shows five points, stored in a standard grid file using three buckets. This is clearly more than the minimum of two buckets, required to store five points. To save storage space, we propose to distribute the points among two grid files, the *primary grid file* P and the *secondary grid file* S, both spanning the entire data space. The primary and the secondary grid file together constitute the twin grid file. By using only one bucket for each grid file, the five example points can be stored in two buckets (see Figure 2.2 (a)).

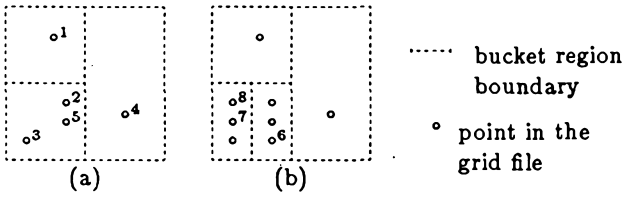


Figure 2.1: A standard grid file ($b = 3$)

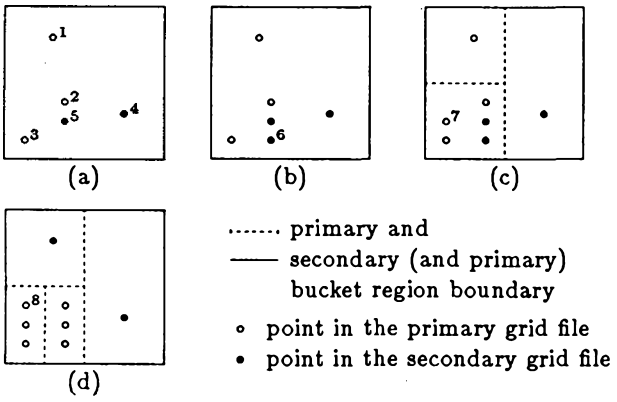


Figure 2.2: A twin grid file ($b = 3$)

In the example of Figure 2.1 (b), an extra bucket is needed to store point 6. In the twin grid file, no extra bucket is needed if point 6 is stored in S and not in P (see Figure 2.2 (b)). Since point 7 comes to lie in a bucket region of a full bucket in P and in S, the bucket in P is split. In our example, it needs to be split twice before it actually can be used to store point 7 (see Figure 2.2 (c)). Similarly, point 8 causes the split of a bucket in P. Unless further optimization takes place, one secondary and three primary buckets store our eight points. The twin grid file utilizes storage space better than that: points 5 and 6 are stored in a primary instead of a secondary bucket, and point 1 is stored in a secondary instead of a primary bucket, as shown in Figure 2.2 (d). Hence, only three instead of four buckets suffice to store our eight points. In addition, further insertions are more efficient in the twin grid file as against the standard grid file: any next point can be inserted into the twin grid file without the need for an extra bucket, whereas in the standard grid file an extra bucket may be necessary.

In case the set of points is known beforehand, a space efficient twin grid file can be constructed faster than by repeatedly inserting points. In our example in Figure 2.2, for instance, point 5 should be stored in the primary grid file in the first place, instead of being shifted back and forth. In Section 4, we precisely describe an efficient algorithm to compute the optimal twin grid file for a given set of points.

3 The dynamic twin grid file

A preliminary version of the dynamic twin grid file has been presented in [8]; here, we briefly recall its principle of operation, in order to compare it to the optimal twin grid file.

In the dynamic twin grid file D, insertions and deletions trigger restructuring operations; range queries do not initiate restructuring. In an insertion, the point to be inserted into D is first inserted into P; then, a restructuring operation takes place. Similarly, a point to be deleted from D is deleted from P or S, depending on where it is stored, and then a restructuring operation follows. Note that an insertion or deletion within P or S may already lead to some reorganization within P or S, as prescribed by the standard grid file mechanism. Range queries need to be carried out on both parts of the twin grid file, but remain unaffected from the restructuring operations.

The dynamic twin grid file D is restructured by transferring a point from P to S, a *shift down* operation, or from S to P, a *shift up* operation, and carrying out the necessary adjustments within P and S, as prescribed by the standard grid file mechanism. After a sequence of shift operations, a bucket in P or in S may become empty or two buckets can be merged, resulting in one bucket less. We call this a *saving* in P or in S, respectively. If a sequence of shifts necessitates a bucket split resulting in two non-empty buckets, or an empty bucket region becomes non-empty, an additional bucket in P or in S is needed. We call this a *loss* in P or in S, respectively. If the number of buckets in P or in S remains unchanged after a sequence of shift operations, we call this *neutral* in P or in S, respectively.

In order to achieve a high space utilization, our restructuring operations aim at minimizing the number of buckets used to store a given set of points. This objective is pursued locally, i.e., restructuring actions are limited to points in a (generally small) connected subregion R of the data space. The larger the restructuring region R, the more external accesses are needed, without necessarily increasing the space utilization.

To minimize the number of buckets, we merge two buddies, if a saving in P or in S is possible without any shift. If there exists a sequence of shift down operations leading to i savings in P and less than i losses in S, for

some integer $i > 0$, then we perform these shifts (see Figure 3.1 for $i = 2$). Note that this restructuring action for $i \geq 2$ is essential for achieving any savings at all when inserting points into the initially empty dynamic twin grid file.

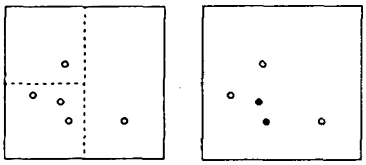


Figure 3.1: Before and after restructuring

To support future optimization efforts, we try to keep secondary regions large and secondary buckets rather empty. To this end, we perform a sequence of shift up operations if it leads to i savings in S and at most i losses in P, for $i > 0$ (see Figure 3.2 for $i = 1$). For the same reason, we perform any shift up operation neutral in P and in S. As a consequence, all points stored in S lie in empty or full bucket regions of P.

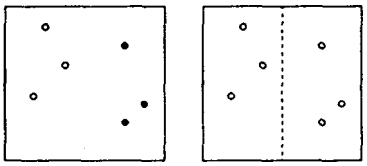


Figure 3.2: Before and after restructuring

Figures 2.2 (c) to (d) of the last section illustrate that even though all of the above actions are simple and their effect straightforward, their combined and repeated application in general may rearrange the association of points with buckets quite substantially.

However, the distribution of points among P and S according to our actions is not even locally optimal. Figure 3.3 (a) shows a situation after insertion of 6 points. Point 7 causes a split in P (Figure 3.3 (b)), and the dynamic twin grid file remains optimal. After insertion of point 8 (see Figure 3.3 (c)), none of our restructuring actions can be applied, even though this situation is not optimal, as Figure 3.3 (d) illustrates. The next section is devoted to the study of optimality in the twin grid file.

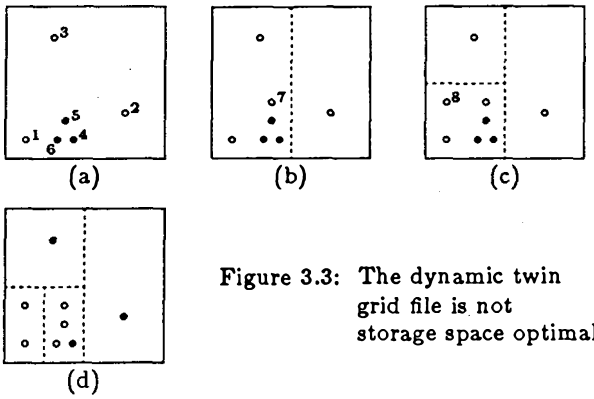


Figure 3.3: The dynamic twin grid file is not storage space optimal

4 The optimal twin grid file

In this section we characterize the storage space optimal twin grid file, and we describe an algorithm to compute it for a given set of points. First, however, we stipulate the split and merge strategy of the underlying standard grid file. A split line partitions a bucket region by cutting the longer of the edges into halves. If the bucket region is square, the split line is vertical (we do not use the freedom in choosing horizontal or vertical split lines). For simplicity, we want the merge operation to be symmetric to the split operation, i.e., two buckets can only be merged if their bucket regions are two halves created by a split. Hence, for each bucket there is at most one buddy with which it can be merged. Of course, two buddies are merged only if all points in both buckets fit into one bucket. As a consequence, by splitting and merging we only arrive at situations that can also be created by only splitting the initially unpartitioned data space.

4.1 The storage space optimization problem

In the twin grid file approach, two standard grid files are used to increase storage space utilization. Let us now consider the problem to achieve the highest possible — the optimal — space utilization for a set of points. Each point has to be stored in one of two grid files. We want to divide the points among the grid files such that the total number of buckets is minimal. Note that since the storage space consumed by the grid directories is fairly small as compared to the space for data buckets, we deliberately ignore the former in our definition of storage space optimality.

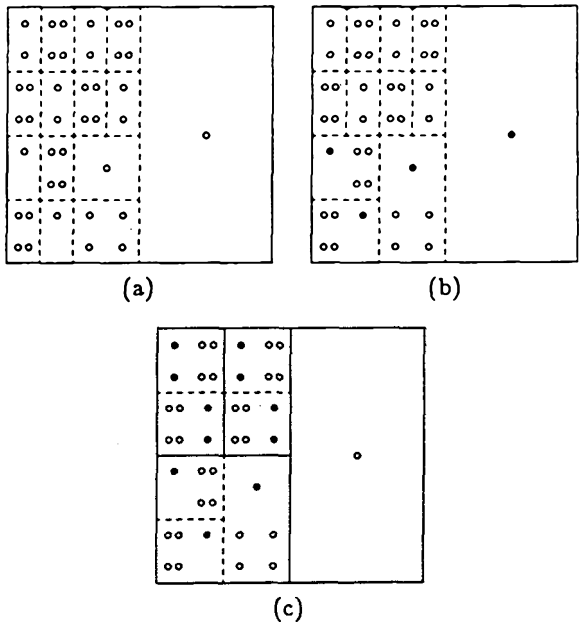


Figure 4.1: A twin grid file with 40 points and $b = 4$

To get some insight into the problem, let us now assume that all points are stored in the primary grid file P; the secondary grid file S is empty. Figure 4.1 (a) shows 40 points stored in the primary grid file with $b = 4$. Here, 15 buckets are needed to store these 40 points. Now we try to repeatedly use one more secondary bucket to increase the storage space utilization. At the beginning, we use one secondary bucket and shift down a set of points leading to the maximal number of savings in P. For Figure 4.1 (a), the result of this operation is shown in Figure 4.1 (b); 4 savings in P, i.e., 3 net savings, are possible. If we now vertically split the secondary bucket into two, we have to use one bucket more, and on both sides of the split line no more savings are possible. Thus, the number of overall savings decreases by one. Nevertheless, this split of the secondary bucket is necessary to approach the global optimum. In our example of Figure 4.1 (a), 3 secondary bucket splits are optimal (see Figure 4.1 (c)). Hence, the storage space optimum of 11 buckets cannot be achieved if only secondary bucket splits with non-negative savings are carried out.

4.2 Storage space optimality

Let us consider a set of x points stored in a twin grid file. Depending on the number x of points being stored, we distinguish 4 cases.

If no point is stored ($x = 0$), the twin grid file is empty, and therefore it is optimal.

If $0 < x \leq b$, one bucket in either of the two grid files is optimal. The other grid file is empty.

If $b < x \leq 2b$, two buckets are necessary and sufficient. Sometimes the two buckets may belong to one grid file, in which case the other grid file is empty. In general, one bucket for each grid file always realizes the optimum.

If more than $2b$ points are stored ($x > 2b$), the data space of at least one grid file has to be split, because at least three buckets are needed to store the x points. We distinguish two cases, depending on whether the data space of the other grid file has to be split to achieve the optimal storage space utilization.

In the first case, the data space of the primary grid file is split, whereas the data space of the secondary grid file is not. For this situation to be optimal, the points have to be distributed among the two grid files in such a way that the number of buckets in P is minimal.

In the second case, the data space of both grid files is split into two subspaces. Let n_s be the number of buckets to store the x points in this case. For this situation to be optimal, the points in each of the two subspaces have to be organized in an optimal way, as the following theorem shows.

Theorem: Let n_1^* and n_2^* be the optimal number of buckets for the two subspaces of the data space. Then $n_s = n_1^* + n_2^*$.

Proof: Since the two subspaces can be organized such that they use n_1^* and n_2^* data buckets, it is clear that $n_s \leq n_1^* + n_2^*$. To show that $n_s \geq n_1^* + n_2^*$, let us now assume that $n_s < n_1^* + n_2^*$. Since the subspaces created by our split strategy are the same for both grid files, either in the first subspace less than n_1^* buckets or in the second subspace less than n_2^* buckets may only be used. Therefore, either n_1^* or n_2^* cannot be the optimal number of buckets for the corresponding subspace, a contradiction.

4.3 The storage space optimization algorithm

Assume that we want to store x points in a twin grid file with the goal of achieving the highest possible storage space utilization. For the given set of points, our algorithm computes the subset of points to be stored in S. All other points then have to be stored in P. After this computation, the two subsets of points are stored in the two standard grid files, resulting in a space optimal twin grid file. If the optimal subdivision of the data space into bucket regions is not unique, we partition the points as unevenly as possible among the two grid files.

Let us now briefly outline the basic concepts of our algorithm. The algorithm recursively generates the optimal solution for a data space DS and the set X of points lying in DS. Initially, DS is the entire data space. For a sufficiently large point set X, the data space DS is partitioned into two halves DS_1 and DS_2 with point sets X_1 and X_2, according to the split strategy. The optimal solutions for DS_1 and DS_2 are used to compute the optimal solution for DS. If X contains no more than b points, no further recursion is necessary, since all points in X fit entirely into one bucket.

For data space DS and point set X, our algorithm returns several values, most of which are only necessary during the recursion. Among these values, we get a set s^* of points to be stored in S, representing an optimal solution for DS and X. To compute s^*, we consider the two cases that there is only one bucket region for DS in S, and that there is more than one bucket region for DS in S. We compute the optimal solutions for both cases. Then s^* can be determined by taking the better of the two optimal solutions. In more detail, our algorithm delivers the following values:

- the minimal number n^* of buckets in P and S needed to store all points in X in P and S;
- a set s^* of points to be stored in S to achieve n^*;
- the number n of buckets in P, provided that all points in X are stored in P;
- a set s of points that achieve a minimal number of buckets in P if they are stored in S, provided that exactly one bucket is used in S.

There are at most b points in s, since all these points have to fit into one bucket. Let $m \geq 1$ be the number of buckets saved in P by storing all points of s in S instead of in P. We partition s into m subsets, called groups, such that one bucket is saved in P if all points in a group of s are stored in S. For example, in Figure 2.1 (b),

$s = \{1,4\}$, and the groups are $\{1\}$ and $\{4\}$ (see also Figure 2.2 (d)).

The algorithm distinguishes three disjoint cases.

Case 1: $X = \emptyset$. Then we get:

- $n^* = 0$
- $s^* = \emptyset$
- $n = 0$
- $s = \emptyset$

Case 2: X contains at least one and at most b points. These points fit entirely into one bucket of P, and one bucket is also necessary:

- $n^* = 1$
- $s^* = \emptyset$
- $n = 1$
- $s = X$, in one group

Case 3: The number x of points in X is more than b. DS is split into two parts DS_1 and DS_2, according to the split strategy. Then our algorithm recursively computes the values n_1, n_1^*, s_1, s_1^* for DS_1, and the values n_2, n_2^*, s_2, s_2^* for DS_2. Since $x > b$, $n_1 + n_2 > 1$ must hold. We distinguish 2 cases.

Case 3.1: $n_1 = n_2 = 1$. DS_1 and DS_2 are regions for one bucket in P each. These two buckets can be merged after storing $x - b$ points in S, since they are buddies. Here, we decide to store all points in buckets in P. From the values for DS_1 and DS_2 we get the values for DS:

- $n^* = n_1^* + n_2^* = 2$
- $s^* = s_1^* \cup s_2^* = \emptyset$
- $n = n_1 + n_2 = 2$
- $s = x - b$ arbitrary points of $s_1 \cup s_2$, in one group

Case 3.2: $n_1 \geq 2$ or $n_2 \geq 2$. DS_1 and DS_2 cannot be merged, since at least one of these subspaces is further subdivided. Clearly, $n = n_1 + n_2$. First we create set s with at most b elements by transferring the largest possible number of groups of s_1 and s_2 into s, i.e., by transferring the smallest groups of s_1 and s_2 into s until the number of points in s reaches b. Let m be the number of these groups. If we assume that only one bucket is used in s, the optimal number n_0^* of buckets for DS and X is $n_0^* = n_1 + n_2 - m + 1$. We distinguish two cases.

Case 3.2.1: $n_0^* \leq n_1^* + n_2^*$. Here, using one bucket in S is at least as good as splitting DS in S. Therefore, we get:

- $n^* = n_0^* = n_1 + n_2 - m + 1$
- $s^* = s$
- $n = n_1 + n_2$
- $s =$ the set of points described above

The optimal sets s_1^* and s_2^* are of no further interest, since storing all points of these sets in S would not lead to better storage space utilization than storing all points of s in S.

Case 3.2.2: $n_0^* > n_1^* + n_2^*$. It is better to split DS in S, i.e., the optimal configuration in DS_1 and DS_2 is better than leaving DS unsplit in S. Now we get the values:

- $n^* = n_1^* + n_2^*$
- $s^* = s_1^* \cup s_2^*$
- $n = n_1 + n_2$
- $s =$ the set of points described above

The most important decision in this algorithm is whether or not to split data space DS in S. Set s^* causes the split of DS in S only if the optimal number of buckets decreases. Among all optimal twin grid files, our algorithm generates one with a minimal number of points in S.

4.4 The complexity of the storage space optimization algorithm

We will show that the time complexity of our algorithm to build an optimal twin grid file is linear in the number k of bucket regions in a standard grid file G for storing the given set X of points (note that k is not the number of directory entries).

Initially, all points are stored in G. Then, the partition of X into s^* and $X - s^*$ is computed. The recursion of the algorithm for computing s^* terminates whenever the data space is a bucket region in G. Therefore, we get $2k - 1$ recursive calls. A detailed analysis shows that the complexity of each recursive call is constant for constant bucket capacity b. Hence, the partition of X between P and S can be computed in time linear in k. Finally, all points in s^* are deleted from G and inserted into S; now P is identified with G. This step also consumes time linear in k. Therefore, the time to build an optimal twin grid file is linear in k.

Further analysis shows that in main memory, only constant storage space for each recursive call is needed, since as soon as s^* contains more than b^2 points, these can be transferred to S.

5 Implementation and performance

For two-dimensional points, we implemented the standard grid file, as well as the optimal static twin grid file, and the dynamic twin grid file on an IBM-AT in Modula-2. The twin grid files are based on two standard grid files with the same bucket capacity. Each standard grid file has a two-level directory.

Our implementation of the optimal twin grid file slightly differs from the description of the algorithm in Section 4.3. We first store all points in P. After computing the points to be stored in S, these points are inserted into S and deleted from P; all other points remain in P.

For the dynamic twin grid file, the directory contains the number of points stored in each bucket, in addition to the bucket addresses. This information greatly reduces

the number of bucket accesses in restructuring the dynamic twin grid file. The storage space requirement for directory pages grows by 50% (a directory entry consumes three instead of two bytes), but space utilization altogether, including data buckets, is not affected noticeably (in our case, the directory accounts for 1% of the storage space).

Now we sketch the algorithm used to restructure our dynamic twin grid file. If the point to be inserted fits in its corresponding bucket in P or S without requiring an additional bucket, the point is inserted. Otherwise, the point is inserted into P, using one additional bucket; then restructuring actions take place.

The region for restructuring is determined by the bucket region in S which covers the insertion point. Let b_S denote the corresponding bucket; if this region is empty, b_S is the empty bucket.

After one bucket in P has been split into two, all points of b_S that can be stored in the two new buckets are shifted up. Among the following four actions, we carry out those that lead to savings, in the given order:

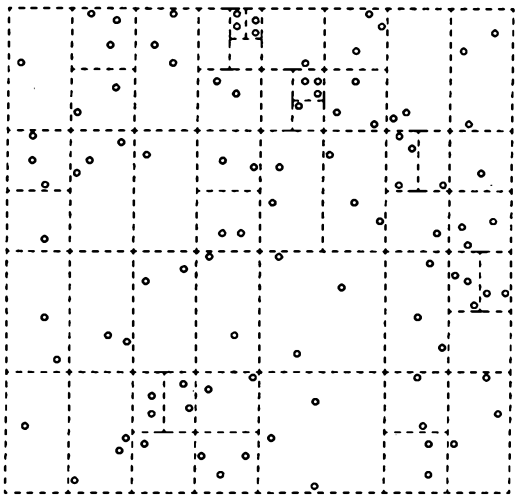
(A1) merge b_S with its buddy without any shift operation;

(A2) make b_S empty with shift up operations requiring only one additional bucket in P;

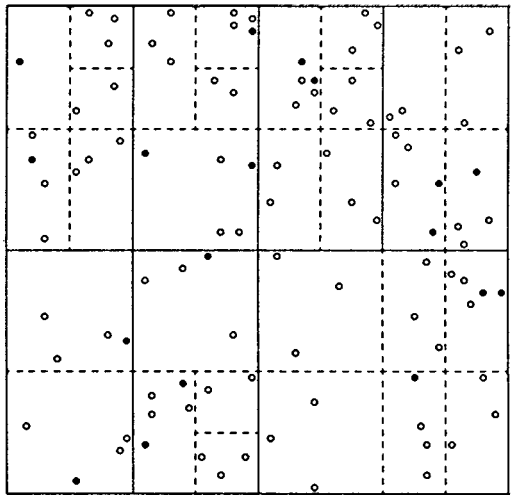
(A3) shift down points into b_S, to realize at least one saving in P, or two savings if b_S had been empty before;

(A4) split b_S, if b_S is not empty, and shift down points into b_S and its buddy to achieve at least two savings in P.

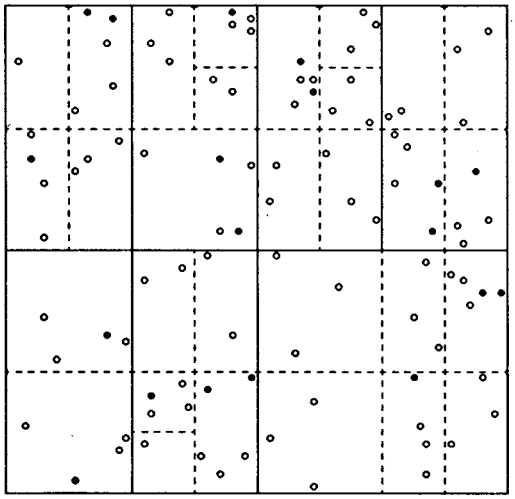
Whenever possible, two buckets in P are merged after shift down operations. This implementation just represents one specific way to determine the local restructuring region R, and the application of the restructuring actions.



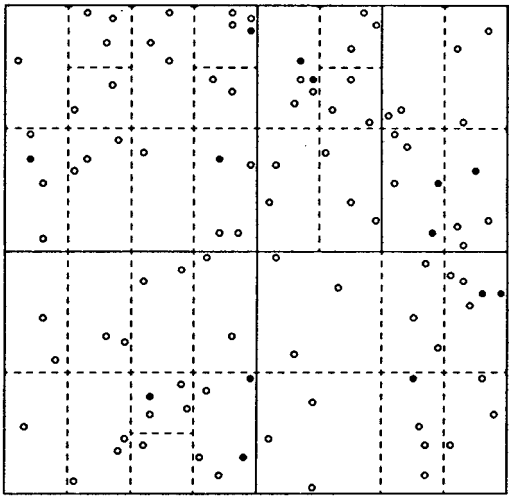
(a) The standard grid file



(b) Our optimal static twin grid file



(c) Our dynamic twin grid file



(d) A dynamic twin grid file variant

Figure 5.1: 100 uniformly distributed points in grid files, where each bucket can hold 3 points

Figures 5.1 (a) through (c) show the effect of our three implementations for a sequence of 100 insertion points. The standard grid file (Figure 5.1 (a)) needs 46 buckets to store the 100 points, whereas our optimal static twin grid file (Figure 5.1 (b)) requires 36 buckets. The dynamic twin grid file (Figure 5.1 (c)) needs only 37 buckets, which is almost optimal, at the expense of 36 restructuring operations. For comparison, Figure 5.1 (d) shows the result of our dynamic twin grid file after only 5 restructuring actions, performed during the last 20 insertions. Here, 39 buckets are needed, i.e., 7 savings have been achieved as against the standard grid file.

Table 5.1 shows the performance evaluation of our three implementations G (standard grid file), O (optimal static twin grid file), and D (dynamic twin grid file). Each bucket holds at most 20 points, a realistic number for many applications. We have inserted three sequences of 20000, 30000 and 40000 uniformly distributed two-dimensional points into the initially empty grid files. We measured the storage space utilization, counted the number of read and write operations, and carried out 300 range queries with square ranges of three different sizes at uniformly distributed positions. We calculated the average number of read operations to answer these range queries.

			number of stored points		
			20000	30000	40000
storage space utilization (in %)		O	90.50	88.76	90.87
		D	90.17	88.65	90.01
		G	69.78	69.83	69.42
number of bucket and directory accesses to store the points	R	O	38100	58986	79901
		D	54548	86591	118754
		G	35728	55413	75112
	W	O	23211	35001	46692
		D	47245	74019	101517
		G	22630	34046	45510
av. number of bucket and directory accesses to answer range queries (area in % of the area of the data space)	0.09	O	7.50	8.77	9.72
		D	7.55	8.87	9.94
		G	5.77	7.02	7.99
	1	O	24.10	32.92	39.40
		D	24.25	33.06	40.05
		G	25.18	34.18	42.72
	4	O	64.64	94.06	116.88
		D	65.67	94.31	118.52
		G	75.78	107.32	138.41

Table 5.1: Performance evaluation

The storage space utilizations in both twin grid files vary around 90%, with small differences. This exceeds the storage space utilization of the standard grid file G: G needs 29% more buckets than D or O. In our experiments it turns out that 11% (14%) of all points are stored in the secondary grid file in O (in D). To achieve the high space utilization in D, for the insertions less than twice the number of block accesses are needed as compared with G. In contrast, in O just 5% additional block accesses are required.

It is essential, however, that range queries do not lose their efficiency in the twin grid files versus G. For ranges covering about 20 buckets, i.e., roughly 350 points are affected, the twin grid files perform as good as G. As expected, for larger range queries, D and O outperform G, due to the better space utilization. For all range sizes, O works slightly better than D.

Operations not referring to the spatial neighborhood of points are of minor importance in the applications we have in mind. However, they may be carried out efficiently as well. Let us illustrate this briefly for exact match queries: they cost at most twice as many block accesses as in the standard grid file. Successful exact match queries cost 1.11 (1.14) times as many block accesses as in the standard grid file, on the average, due to the distribution of points among P and S in O (in D). For unsuccessful exact match queries, the average block access factor is only 1.54 (1.57), because for query points falling into non-full and non-empty primary bucket regions, no secondary bucket access is necessary.

The performance in space and time of the twin grid file does not depend on the type of distribution of the points' coordinates. As long as the distributions are fairly smooth, e.g. an exponential distribution for the x-coordinates and a Gaussian distribution for the y-coordinates, our experiments indicate that space utilization remains almost exactly as in Table 5.1. The number of accesses for insertions slightly increases for D against G and O; the results for range queries are even slightly better for O and D against G.

Our dynamic twin grid file D can be adapted to various situations and user requirements in numerous ways. For instance, for a sequence of insertions, restructuring actions at the beginning have a weaker effect on space utilization than at the end. If the total number of insertions is known beforehand, our experiments show that for achieving a space utilization of roughly 90%, it is sufficient to restructure only during the last fifth of all insertions, saving more than 20% of all block accesses as against D. Otherwise, block accesses can be saved by restructuring less frequently than in D, during the whole sequence of insertions, at the cost of decreasing space utilization. In a different attempt to save block accesses, we may restrict the restructuring actions applied in D. For example, 10% of all block accesses can be saved, by restructuring according to actions (A3) and (A4) only, with a resulting space utilization of 88%. Furthermore, applying action (A1) is slightly less costly than action (A2), but brings out better space utilization.

6 Conclusion

We have presented a storage space optimal access scheme for multidimensional points, the twin grid file, supporting spatial proximity queries efficiently. In the twin grid file, the points to be stored are distributed among two standard grid files, both covering the entire data space. For a static point set, we characterize the optimal storage space utilization and propose an algorithm for achieving it efficiently. A dynamic variant with space utilization varying with the amount of redis-

tribution efforts proves to be close to optimal. Typical range queries can be answered in both types of twin grid files at least as fast as in the standard grid file.

We have implemented the standard grid file, the dynamic twin grid file, and the optimal static twin grid file, and compared their performances. Our implementation of the dynamic twin grid file has redistributed points among the two grid files according to few criteria, and the corresponding actions have been applied in very limited combinations only. Nevertheless, the twin grid file outperforms the standard grid file significantly; surprisingly it turns out that its space utilization is nearly optimal (close to 90%). The optimal static twin grid file can be built at a fraction of the cost of building the dynamic twin grid file; range queries can be answered slightly faster for the optimal twin grid file.

The principle of building twin structures can be applied to other index structures as well. For instance, extendible hashing [2] can clearly be used as the basis for an optimal static and for a dynamic twin structure. We conjecture that the storage space utilization of other structures based on the principle of recursive halving, like e.g. the digital B-trees [11] and the BANG file [4], will also benefit from applying the twin principle. It is open by how much the application of the twin principle increases space utilization for structures that partition the data space according to the actual points, like e.g. B-trees [1] or K-D-B-trees [16], and for structures suitable for non-zero size spatial objects [3,5,17,18].

Note that in our characterization of the optimal twin grid file, the definition of a very restricted strategy for splitting and merging has been essential. With this strategy, the behaviour of the two-dimensional twin grid file is no more general than for a one-dimensional twin grid file. More general split and merge strategies, exploiting the freedom of choosing split and merge directions in higher dimensions, as well as more than two sibling structures [15], are currently under investigation.

Acknowledgement

We wish to thank anonymous referees for suggestions improving the presentation, and Brunhilde Beck and Gabriele Reich for typesetting the paper and styling the layout.

References

1. R. Bayer, E. M. McCreight:
Organization and Maintenance of Large Ordered Indices, Acta Informatica, Vol. 1, 3, 1972, 173–189.
2. R. Fagin, J. Nievergelt, N. Pippenger, H. R. Strong:
Extendible Hashing: A Fast Access Method for Dynamic Files, ACM Transactions on Database Systems, Vol. 4, 3, 1979, 315–344.
3. C. Faloutsos, T. Sellis, N. Roussopoulos:
Analysis of Object Oriented Spatial Access Methodes, Proc. ACM SIGMOD International Conference on Management of Data, 1987, 426–439.
4. M. Freeston:
The BANG file: a new kind of grid file, Proc. ACM SIGMOD International Conference on Management of Data, 1987, 260–269.
5. A. Guttman:
R-Trees: A Dynamic Index Structure for Spatial Searching, Proc. ACM SIGMOD International Conference on Management of Data, 1984, 47–57.
6. K.H. Hinrichs:
The grid file system: implementation and case studies of applications, Doctoral Thesis No. 7734, ETH Zürich, 1985.
7. A. Hutflesz, H.-W. Six, P. Widmayer:
Globally Order Preserving Multidimensional Linear Hashing, Proc. IEEE 4th International Conference on Data Engineering, 1988, 572–579.
8. A. Hutflesz, H.-W. Six, P. Widmayer:
The Twin Grid File: A Nearly Space Optimal Index Structure, Proc. International Conference Extending Database Technology, 1988.
9. H.-P. Kriegel, B. Seeger:
Multidimensional Order Perserving Linear Hashing with Partial Expansions, Proc. International Conference on Database Theory, 1986, 203–220.
10. R. Krishnamurthy, K.-Y. Whang:
Multilevel Grid Files, IBM Research Report, Yorktown Heights, 1985.
11. D.B. Lomet:
Digital B-Trees, Proc. 7th International Conference on Very Large Data Bases, 1981, 333–344.
12. D.B. Lomet:
Partial Expansions for File Organizations with an Index, ACM Transactions on Database Systems, Vol. 12, 1, 1987, 65–84.
13. J. Nievergelt, H. Hinterberger, K.C. Sevcik:
The Grid File: An Adaptable, Symmetric Multikey File Structure, ACM Transactions on Database Systems, Vol. 9, 1, 1984, 38–71.
14. E.J. Otoo:
Balanced Multidimensional Extendible Hash Tree, Proc. 5th ACM SIGACT/SIGMOD Symposium on Principles of Database Systems, 1986, 100–113.
15. K. Ramamohanarao, R. Sacks-Davis:
Recursive Linear Hashing, ACM Transactions on Database Systems, Vol. 9, 3, 1984, 369–391.
16. J.T. Robinson:
The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes, Proc. ACM SIGMOD International Conference on Management of Data, 1981, 10–18.
17. T. Sellis, N. Roussopoulos, C. Faloutsos:
The R+-Tree: A Dynamic Index for Multi-Dimensional Objects, Proc. 13th International Conference on Very Large Data Bases, 1987, 507–518.
18. H.-W. Six, P. Widmayer:
Spatial Searching in Geometric Databases, Proc. IEEE 4th International Conference on Data Engineering, 1988, 496–503.