Although many solutions to the problem of representing spatial data have been proposed, most are not practical. They cannot be guaranteed to perform well with large and arbitrarily distributed data collections. They may not be easily integrated with concurrency and recovery software already written for database systems. However, two proposed structures do have some guarantees, similar to those which have made the B+-tree so successful for one-dimensional data. These analytic guarantees include worst case space utilization in data and index pages, a minimal fan-out and exact match search time bounded by the height of the tree. The two methods are the holey brick tree and bit interleaving using a B+-tree. Both can also be integrated with concurrency and recovery systems in the same way that B+-trees are.

1. Introduction

Current Database Management Systems efficiently organize, access and manipulate enormous quantities of data for traditional applications in banks, airlines, hospitals and other large organizations. However, these DBMSs are not equipped to handle spatial data, such as the terabytes of meteorological, astronomical and geographic information now being gathered by satellites. New access methods must be developed and must be integrated with current software. Since the data collections are so large and so unpredictable in distribution, the new methods must have analytic guarantees for performance.

We anticipate queries such as:

"Find all the satellite photographs which contain the confluence of the Mississippi and the Missouri Rivers."

"Obtain all meteorological data within 100 miles of Kansas City and up to 50 miles above the surface of the earth during the month of April."

To answer these queries efficiently, we shall want to store nearby data in contiguous areas of a disk. This means that we are translating the concept of nearby in $k$-dimensional space to linear or one-dimensional space on a disk.

We also assume that our data collection is dynamic: new information is constantly being added to the data base. We wish to be able to maintain the index structure and the proximity on the disk of data which is nearby in several coordinates while the database is growing.

In addition, we assume that the data collection is so large that the index itself may not fit in main memory. So we are concerned that the index be paginated in such a way that retrieval does not require excessive disk accesses for obtaining the necessary index pages.

With these assumptions and goals in mind, we focus on the following desirable properties of spatial indexing structures.

1. Good space utilization in data pages.
2. Good space utilization in index pages.
3. High fan-out. (The index should be significantly smaller than the data collection.)
4. Fast exact match search. (If the coordinates are given, the data should be obtained very quickly.)
5. Fair clustering by all attributes.
6. Reasonably fast range searches.
7. A high degree of concurrency.

8. Integration with the query optimizer, locking system and recovery system of existing DBMSs.

9. Simple design for insertion algorithms—easy incremental growth.

These properties should hold for any data distribution, for any order of insertion, and for any query patterns.

In this paper, we indicate some of the progress which has been made in inventing structures with these desirable properties. First, in section 2, we say a little about the most successful one-dimensional access method, the B+-tree. In section 3, we describe bit-interleaving using a B+-tree[10]: the spatial access method which is the one to beat. To be considered as a practical solution to the problem of spatial access, other methods should at least match the guarantees of [10]. In section 4, we look at some other well-known spatial access methods, such as the grid file [9] and the R-tree [2]. In section 5, the hB-tree or holey brick tree is briefly described [4,5]. Section 6 contains some conclusions and indications of ongoing research.

2. The B+-tree

One access method dominates current Database Management systems. This is the B+-tree. There are very good reasons for this domination. The B+-tree has guarantees of space utilization in the data pages and in the index pages (except the root) of at least 50%, with average case of about 70% (ln2). All the data pages are at the leaves of the tree, and all leaves of the tree are at the same level. The upper levels of the tree contain only keys and pointers to the next level of the tree. This implies that the fan-out or ratio of the number of data pages to the number of index pages is very high. For example, if 4K pages are used with 8-byte keys and 4-byte addresses, (disregarding page overhead) the minimum number of children of a non-root node is 170. The average number is about 240. This implies that a tree with two levels above the leaves would have an average of 240 times 240 or 57,600 data (leaf) pages, while only needing 241 pages for the index.

This illustrates the fact that B+-trees are wide and short. The number of disk accesses for single-record retrieval is bounded by the height of the tree, and since the upper levels of the tree are often in memory, this number is usually less than the tree height. Since the trees are short, the number of disk accesses is small, typically one or two, even for very large files.

Unlike hashing methods, the B+-tree can be used for range queries ("Find all records with last-name after T"). Records with nearby key values are stored on the same data page, making these range queries efficient in terms of numbers of disk accesses. Because of this feature and because the guarantees on space utilization, single-record search time and fan-out hold for arbitrary data distributions, the B+-tree is often the only access method provided for many DBMSs. Hashing does not provide these guarantees, and while its average search time may be better for most data collections, its worst case is very bad. Worst case guarantees are usually more meaningful than average case performance when the data distribution is not known ahead of time.

Insertion in the B+-tree is incremental, usually involving only the data page where the record is inserted and very rarely involving changes in the upper levels of the tree. This means that the upper levels of the tree are rarely locked for writing. This is important since all transactions accessing the data through the index must descend the tree from the root. Locking the root, for example, would prevent all other transactions from accessing the file. (An insertion which does change an index node is illustrated in figure 1).

Figure 1. Two additions to a B+-tree. In the first case, only the leaf is updated. In the second case a new leaf page is allocated and the index is changed as well.

In summary, the B+-tree has guaranteed worst-case space utilization in data and index pages, guarantees on fan-out, automatic clustering for range queries, and a local incremental insertion algorithm, usually confined
to a small portion of the tree. B+-trees have been inte-
grated with concurrency and recovery system of most
DBMSs (see [8] for example) and their parameters
are used by the query optimizers to estimate access plan
costs. More information on the B+-tree can be found in
[15].

3. Bit Interleaving and the B+-tree

*Bit interleaving* or *z-ordering* [10] creates a linear
order on a k-dimensional space according to the inter-
leaved bit patterns from the several attributes. One pos-
sible bit interleaving pattern is shown in Figure 2. The
first bit of one attribute ("y") becomes the first bit of
the new key number N to be placed in the B+-tree. The
first bit of "x" becomes the second bit of N. The second
bit of "y" becomes the third bit of N and so forth. The
pattern is recursive, so that although only 16 points are
shown, the generalization of the pattern is fairly obvi-
ous.

Since after the bit interleaving pattern is chosen, the in-
terleaved key is placed in a B+-tree, this method inher-
its the B+-tree's insertion algorithm, concurrency and
recovery methods as well as its guaranteed space utili-
zation and fan-out. Accesses for exact match search are
bounded by the height of the tree.

The path in Figure 2 describes the order of the points.
Thus when the records are placed in a B+-tree, one
leaf of the tree will correspond to a connected segment
of the path. That is, the records stored in that leaf will
have keys whose interleaved values all lie on that con-
ected path segment.

In general, the boundary of a region spanned by a path
segment can be quite uneven. This is illustrated in
Figure 3. Note that this does not at all imply that
clustering is poor; only that it is not rectangular.

3.1 Scaling and its Effect on Clustering

The larger problem comes when the scales of the differ-
ent attributes do not match. For instance, say most of
the points have 23 leading zeros in the binary expres-
sion of one of the attributes (but a few do not have 23
leading zeros). Then that attribute does not figure in the
classification scheme, and one might as well use con-
catenation.

For example, look at the salaries of the employees at
Chrysler Corporation. If Lee Iacocca’s salary is includ

![Figure 2. Points on a two-dimensional grid are mapped to a
one-dimensional ordering using bit interleaving. The points
a and b are close in space but not in the one-dimensional or-
dering. The points a and c are close in the one-dimensional
ordering but not in space. But, at least with uniformly dis-
tributed data, two points close in space are likely to be close
in z-ordering.](image)

![Figure 3. The path segments stored in a B+-tree leaf may
describe areas whose boundaries are uneven. Here points d, c,
a, e, and f are enclosed in an area spanned by a connected
path segment](image)
ed, most of the other salaries start with a number of leading binary zeros. If the values of one attribute differ on a smaller scale than those of other attributes (except for some outliers, so that leading zeros cannot be erased), bit interleaving acts as if that attribute were not present.

This kind of pattern can show up anywhere in the binary expression of the data. Perhaps bits 5 to 15 are the same for all the salaries or all the social security numbers. This phenomenon corresponds to having most but not all of the points cluster around a horizontal (or vertical) line. Bit interleaving schemes will cluster data by one attribute under these circumstances.

3.2 Range Searches

The algorithm for range searching in [11] represents the search region by a collection of prefix rectangles whose union covers the query region. Here we take advantage of the fact that prefixes of interleaved numbers represent rectangular regions in the space. That is, all points with the same prefix lie in a rectangular region. If the prefix is shorter, the region is larger. This is illustrated in Figure 4.

If the rectangles are small and numerous, the number of separate probes to the B+-tree will be large. (Each prefix rectangle is a range search in the B+-tree: Find all points starting with this prefix.) The number of records retrieved which are not in the search region will be small. On the other hand, if a small number of large prefix rectangles are used, the accuracy of the search will be poor, but the number of probes to the B+-tree will be small. [11] gives heuristics for choosing a collection of prefix rectangles to use in a range search with this trade-off in mind.

This question of clustering and of efficient range searches is explored from another viewpoint in [3]. Here the data is distributed evenly over the space (one data point at each grid point), the range queries are all possible square ones of the same size and the question is how many disjoint segments of the path are in each square when different shapes of path are considered. The ordering path shown here, one based on Gray codes and one on the space-filling curve of Hilbert are considered. The idea is to see which design fits better with rectangular query areas, assuming that disjoint path segments correspond to separate disk accesses. Thus z-ordering could be replaced by some other pattern interleaving and transforming the bits of the key attributes.

It is unclear, however, from any of these studies, how to evaluate spatial clustering and efficiency of range queries for general (possibly non-bit-interleaved) spatial access methods, arbitrary patterns of access and arbitrary distributions of data.

4. Difficulties in Spatial Access Method Design

Many proposed spatial search structures are not able to make minimal guarantees of space utilization, fan-out or single-record retrieval time. These methods do not match the guarantees of bit-interleaving with the B+-tree. There are usually several reasons why they will be likely to perform worse on very large unpredictably skewed data, and not much evidence that they will perform better in any special circumstances.

4.1 Methods Based on Grids

One way to organize data is to place a grid over the area, such as the latitude and longitude grid on the
earth, and assign an equal number of data pages to each cell. This approach is illustrated in Figure 5. Here, many of the data pages are empty. Pursuing this idea even further, suppose we have a gigabyte of data stored in pages of 4096 bytes each. Suppose all the data lies on the diagonal of the grid, and suppose each cell represents one page. The number of cells on each side of the grid is $2^{18}$. We need over 260,000 gigabytes of space to store one gigabyte of data. In general, methods involving grids use $O(n^k)$ space (where $n$ is the number of points and $k$ is the dimension).

4.2 Hierarchical Methods

Many proposed spatial search structures have a hierarchical page organization like a B+-tree, where the leaves of the tree are the data pages and the interior levels are successive refinements of the space, with the root being the least refined. These structures have the problem illustrated in figure 6. When it comes time to split an index page into two index pages, there is no convenient way to make the split. Moving the boldface split line to the left would result in better index space utilization, but poses the problem of what to do about lower level nodes which lie on both sides of the split line.

In addition, for the Grid File, the insertion algorithm is not local. Insertion can cause the creation of a new slice in the grid; this affects concurrency as at least the mirror slice—which may span many index pages—must be locked [14]. Some implementations even rewrite the entire index for this case, although [7] shows that this is not necessary.
5. The Holey Brick Tree

In this section we present the holey brick tree or \textit{hB-tree}[4,5]. The hB-tree has guarantees of worst case space utilization in the index pages and in the data pages. It has guarantees of worst case fan-out and it provides upper bounds for exact match search time. It can be integrated with the concurrency and recovery systems in existing DBMSs[1,5]. The hB-tree uses three basic concepts.

1. Index and data page space utilization are guaranteed by allowing several attributes to be used in the splitting process---instead of using a hyperplane, a brick is removed from from another brick, creating a brick with a hole, or a \textit{holey} brick.

2. Search and insertion algorithms are can be presented in detail because a \textit{k-d} tree is used to represent structures. Some other proposed structures ([13] for example) give no explicit internal representation.

3. In the large, that is, without looking at what happens inside a node, the search algorithms and insertion algorithms are exactly those of a \textit{B+}-tree. This implies:
   a) All leaves are at the same level.
   b) There is one path from the root to one leaf for an exact match search.
   c) To insert new records, first make an exact match search. Then put the new record in its leaf if it fits. Otherwise split the leaf and post parent information. This process is recursive if the parent information makes an index page overflow.
   d) Since posted information is bounded, worst-case guarantees on fan-out can be made.

5.1 Space Utilization in Data Pages in the Holey Brick Tree

In [5], it is shown that an upper or lower corner can always be found which will split a data page with at least a 1:2 ratio. (This implies the worst case data page utilization is 0.33.) This is illustrated in Figure 7 with some data which cannot be split this evenly by any hyperplane.

![Figure 7. A corner can always be found (no matter the dimension of the space) so that any data page is split in at worst a 1:2 ratio](image)

A corner split is represented using a \textit{k-d} tree as shown in Figure 8. In two dimensions, only two \textit{k-d} tree nodes are needed to describe the most complicated data page split. In general, the most information posted to a parent when a data page is split is \(k\) \textit{k-d} tree nodes.

![Figure 8. A holey brick is represented by a \textit{k-d} tree.](image)

5.2 Index Page Space Utilization in the hB-tree

The index pages of the holey brick tree contain no data. They contain only \textit{k-d} tree nodes used to describe the lower levels of the holey brick tree, in the same way that levels above the leaf in a \textit{B+}-tree contain only separators and pointers to lower levels.

A \textit{k-d} tree can be skewed, as in Figure 9. Attempting to split the \textit{k-d} tree at the root corresponds to an attempt to use the first historical splitting plane to split an index node. This could result in poor space utilization for index pages as we have already noted. Instead, a subtree is extracted as illustrated in Figure 9. This corresponds to removing a brick from the brick represented by the full index node.
Figure 9. The k-d trees in index pages are split by extracting a subtree with between 1/3 and 2/3 of the k-d tree nodes. This yields worst case guarantees for space utilization in index pages.

Let $N$ stand for the total number of k-d tree nodes in the index page. We show in [5] that a subtree can always be found with between ceiling $(2N/3)$ and floor $(N/3)$ k-d tree nodes. Thus the same guarantee of at worst a 0.33 space utilization also holds for index pages.

5.3 Fan-Out in the HB-tree

Since at most $k$ k-d tree nodes are posted when a data page is split and since data and index pages have lower bounds on space utilization, worst case guarantees on fan-out at the parent-of-leaf level can be made. To show high fan-out at higher levels of the holey brick tree, so that each level is significantly smaller than the one below it, the amount of information that is posted when index pages split must be limited.

In [5] it is shown that at most $2k + 1$ k-d tree nodes, all of which are contained in the path from the root of the
extracted k-d tree to the root of the full tree in the index
page being split, are posted. The required information
to be posted consists of those nodes on the path which
have not previously been posted and which satisfy:

1. The values are the least upper bound or greatest
lower bound on the path for one of the attributes. (Note
that there are at most 2k such attributes, and they de-
scribe the boundaries of the brick which is removed).
or
2. The k-d tree node is needed to integrate the newly
posted nodes with the nodes already in the ancestor
index pages. (This will be at most one additional k-d
node).

In Figure 10, an example is shown, using a binary
search tree (that is, restricted to one dimension).

5.4 Exact Match Search in the HB-tree

The height of the tree depends on the fan-out and the
number of data pages. Exactly one path from the root
page of the holey brick tree to the data page containing
the record is followed. Thus the number of disk acces-
ses for exact match search is strictly bounded by the
height of the tree.

6. Conclusions and Future Work

In summary, both bit interleaving with the B+-tree and
the holey brick tree have worst case guarantees for disk
space utilization, fan-out and exact match search. Other
proposed spatial indexing methods can not make these
guarantees. The hB-tree is more likely to provide good
spatial clustering (which influences range search effi-
ciency) because it has no problem with scaling. Bit in-
terleaving with the B+-tree provides a simpler insertion
algorithm. Both trees can be integrated with concurren-
cy and recovery methods used in DBMSs. (We have
been working on a simple concurrency method covering
many tree indexes[6]. The details of its application
to hB-trees are in [1].)

Our plans for the future include further investigation
into ways to evaluate a search structure for its perfor-
mance on range queries and for its ability to cluster data
spatially. In particular, we are interested in the question
of clustering when non-point objects (bounded by
bricks or rectangles) are represented by the corner coor-
dinates of their boundaries. This gives objects in k-di-
mensional space a "parametric" representation in 2k-
dimensional space. The question is how this parametric
organization relates to "nearness" in the original space.

This was investigated in [12], but the conclusions hold
only for one search structure and one class of range
search methods.

7. References

[1] G. Evangelidis, D. Lomet, B. Salzberg,
"Modifications of the hB-tree for Node Consolidation


[3] H.V. Jagadish, "Linear Clustering of Objects with
Multiple Attributes," Proc. SIGMOD 1990, pp. 332-
342.

search structure," Proc. 5th Int. Conf. on Data

multi-attribute access method with good guaranteed
performance,” Trans. on Database Systems vol. 15,

for Index Trees,” manuscript under consideration for

56-61.

High Concurrency Index Management Method using
Write Ahead Logging,” IBM Research Report RJ6846 ,
IBM Almaden Research Center, August, 1989.

grid file: An adaptable, symmetric, multikey file struc-
ture,” ACM Trans. on Database Systems, vol. 9, no.1,

tures for associative searching,” Proc. SIGMOD-
SIGACT PODS 1984, pp. 181-190.


