Hybrid Use of Hashing Techniques for Spatial Data

Wayne A. Davis

Department of Computing Science
The University of Alberta
Edmonton, Alberta, Canada T6G 2H1

Abstract

Hashing techniques are reviewed that are applicable to the organization and indexing of spatial data. An adaptive file organization scheme for polygon data is presented that is based on *extendible hashing* and *interpolation-based index maintenance*. This scheme aims to locate a record with a given key using an average of one disk access, and at most two. It also aims to process search by location and set operations efficiently. The proposed scheme has been shown to perform better than the B^+-tree or the EXCELL methods in access efficiency and storage utilization.

1. Introduction

This paper is concerned with organizing and indexing of spatial data. Spatial data has extensive applications such as geographic information systems and computer aided design and drafting. A characteristic of spatial data is that the file size is generally very large and users wish to locate data not only by attribute but also by location. The primary interest of this paper is in representing a polygon network which is a set of polygons that totally partitions a study area. A common method for storing polygons is a series of vectors which define the polygon boundaries. This method has serious shortcomings when operations such as search by location and polygon intersection or overlays are encountered. Alternatively polygon data can be represented in raster format.

A method of representing a polygon network that has seen increasing attention is the quadtree data structure [11]. In this method a region is successively subdivided into equal sized quadrants until each quadrant has a constant value. Large regions can then be represented by a single node in the resulting tree structure. Furthermore the location of a node in the tree is related to the location of the corresponding object in the region concerned. Additionally, set operations are easily performed using a tree traversal technique.

One of the drawbacks of quadtrees is that data stored in a tree format may take many memory accesses to determine the location of a specific object. The actual number of memory accesses, of course, will depend on the depth of the tree where the specific object is located. For small amounts of data where the tree can be stored in main memory this is adequate; however, for large files the data must reside in a bulk storage device like a magnetic or optical disk making the search time prohibitively long.

One technique to overcome this problem is to use a linear quadtree [11] where the tree is stored as a sequence of terminal nodes and the order of the nodes is related to the location of the object. The problem then occurs of how to organize the terminal nodes so that each record can be located with a minimum

number of disk accesses. Furthermore, the technique must be dynamic so that as data is added and deleted the files are expanded and contracted while the spatial relationships are maintained. Furthermore, it should be impossible to overload the system, in the sense that there should be no upper bound on the amount of data that can be accommodated.

In order to better understand the scheme to be presented a review of hashing techniques will be given in the next section. It focuses mainly on dynamic hashing, particularly extendible hashing and linear hashing, which turns out to be very useful for implementing cell methods.

2. Dynamic Hashing and Cell Methods

To be effective, a file organization scheme should allow both random and sequential access to records in the file and be able to support dynamic file maintenance. With the advent of direct access storage devices, several efficient file organization schemes have emerged [7]. Among them, balanced tree organizations and hashing schemes have been prominent.

A tree structure is inherently dynamic in the sense that the number of nodes in a tree varies with insertion and deletion of records. In particular, the B-tree [1] provides an efficient mechanism for structuring dynamic files. It gracefully adapts its shape in response to insertions and deletions of records and supports retrieval of a record with $O(\log n)$ disk accesses, where n is the number of records in the file. Although this logarithmic cost of tree structured files is attractive when compared with the $O(n)$ of sequential files, it is still far greater than the $O(1)$ of direct access methods. Hashing is a popular technique for organizing direct access files.

Nevertheless, traditional hashing schemes have a major disadvantage in that storage allocation for the hash table is static. That is, the size of a file must be estimated in advance and storage space must be allocated for the whole file at once. Thus, a high estimate of the data volume results in wasted space, while a low estimate results in either a costly reorganization of the whole file through rehashing or the attachment of overflow buckets, which slowly degrades the $O(1)$ access time toward $O(n)$ in the worst case. Therefore, when the nature of the data is unknown or the file is dynamic, hashing has not been a good choice.

Recently, however, several *dynamic hashing* [11] (sometimes called *virtual hashing*) schemes have been developed that overcome the difficulty in maintaining large dynamic files. Their main characteristic is that the storage space allocated to the file can be increased or decreased without reorganizing the whole file. Dynamic hashing schemes can be categorized into two types: those that maintain some kind of directory, the size of which varies with the file size, and those that do not use a directory. Those with a directory resolve collisions by *splitting* the overflow address. Those without a directory resolve the overflow condition by *chaining* the overflow address. A chain is *split*, however, when the global storage utilization factor exceeds a predefined upper threshold.

The most noticeable file organization schemes of the first type include *virtual hashing* [9], *dynamic hashing* [8] and *extendible hashing* [6]. *Linear hashing* [10] and *interpolation-based index maintenance* [2] are schemes of the second type. Among these new schemes, of particular interest are extendible

hashing, linear hashing and interpolation-based index maintenance that form the basis of the file organization scheme proposed in this paper.

Extendible Hashing

Extendible hashing has been designed to retrieve the data associated with a given key with two disk accesses, i.e., one access to retrieve the desired part of the directory and another to retrieve the data bucket that contains the key. Extendible hashing achieves its goal by merging the concepts of a radix search tree and hashing. An intuitive description of how these two file schemes have been merged into extendible hashing is now presented.

Radix search trees, which are naturally extendible, can be modified to yield optimum speed merely by having a pointer for each leaf node. Having a pointer for each leaf node has the effect of flattening the tree forcing it to be degenerate. Next, as for hashing schemes, in order to prevent their performance from deteriorating in a dynamic environment, a mechanism which ensures that the hash table remains balanced under insertions and deletions of records is desired. In extendible hashing, this is achieved by introducing the concept of a degenerate radix search tree to the hash table. In order to make the hash table extendible while the hash function is kept unchanged, the partition of the hash address space must have a variable number of variable sized blocks. It starts with a small number of blocks, and if a data bucket overflows, the partition is changed by introducing new boundaries so that in regions where keys cluster the partition is finer than in regions where keys are sparse.

Among various partition techniques, the *buddy system partition* is recommended for its simplicity. Then, when a data bucket overflows, the corresponding block in the address space is halved, a new data bucket is added, and only those keys in the halved block are reorganized. If a bucket underflows as a result of deletions and its buddy is underfilled as well, then the two buckets can be coalesced into one, decreasing the number of blocks in the partition by one.

The buddy system partition of the hash address space results in the same effect as a radix search tree with radix $r = 2$ over the alphabet $\{0,1\}$. Thus, the buddy system partition can be implemented in a hash table using a directory with 2^d entries, where d is the depth of the buddy system partition, or equivalently, the depth of the radix search tree. Then, the directory allows the d leftmost bits of a key to be taken as an index to the directory. If the depth of the partition increases, the directory doubles in size. Although the directory may become large and have to be kept in external storage, it is normally accessible with one disk I/O operation without using a special algorithm since the bit pattern of a key is used as an index to the directory. Therefore, each record in a file can be retrieved in two disk accesses with extendible hashing.

Linear Hashing

Linear hashing is particularly remarkable because it maintains good performance without using an explicit directory. The major advantages of linear hashing are: (1) the file size grows and shrinks gracefully, (2) there is no directory to be kept, and (3) the utilization of the allocated storage is controlled. For an insight into the design principles of linear hashing, consider the following situation. Assume a set of records is being inserted into a number of indexed

buckets, using some hash function, and that a collision occurs at the insertion of a record with a given key. Furthermore, assume a bucket split is performed to avoid the accumulation of overflow records. A natural approach would be to split the bucket in which the collision occurs. However, this approach calls for some sort of directory since buckets will be chosen randomly for a split.

Next, consider the following approach. Suppose that splits are performed on buckets in the order of the index, regardless of which bucket a collision occurs in. That is, first, bucket 1 is split into buckets 1 and N+1, next, bucket 2 is split into buckets 2 and N+2, and so on, until bucket N is split into buckets N and $2N$. After each of the N buckets is split, bucket 1 is chosen again for a split, this time, into buckets 1 and $2N$+1, and so on. Since the index of the bucket to be split is predefined while the index of the bucket with a collision is random, a bucket with a collision is split not necessarily when the collision occurs but with some *delay*. In the meantime, overflow buckets are attached to resolve the collision. The result is that the file becomes progressively larger and no directory is required. Linear hashing is based on this approach.

The insertion of a record with a given key is also simple. First, the access algorithm determines which chain should contain the specified key. If the key is not present, the record with the given key is inserted and the load factor is updated accordingly. Next, if the load factor exceeds the upper limit, a chain split is performed.

The deletion operation is similar to insertion. First, the access algorithm is utilized to determine which chain contains the key to be deleted. If the key is found in the chain, it is removed from the file and the load factor is updated. Then, if the load factor falls below the lower limit, merging of two chains is triggered.

With linear hashing the allocated storage increases linearly, i.e., one chain at a time, maintaining a good overall storage utilization. Although overflow keys are moved to primary buckets with some delay, the rate of overflow keys usually remains small if the bucket capacity is much larger than 1 [10]. Thus, it is reasonable to expect that a key is normally found in one disk access.

Both extendible hashing and linear hashing lend themselves to an adaptable cell method of structuring k-dimensional data. The cell method organizes data by cells that are produced by dividing the study area into a regular grid of intervals corresponding to units of external storage such as disk blocks. Cell methods are attractive because cells capture locality. In fact, most of the successful cell methods that have been implemented are multi-dimensional variations of either extendible hashing or linear hashing. The *EXCELL* method [12, 13, 14] is an adaptation of extendible hashing to k-dimensional data, and the *interpolation-based index maintenance* [2] is a k-dimensional generalization of linear hashing.

Interpolation-based Index Maintenance

Interpolation-based index maintenance (hereafter, interpolation hashing) is a multi-dimensional variation of linear hashing. It has been designed to handle range queries efficiently while preserving the performance of linear hashing in terms of the number of disk accesses required for insertion, deletion and retrieval of a record. Similar to the EXCELL method, interpolation hashing handles range

search efficiently by associating each chain, i.e., each data bucket, with a cell in the search space.

The association between chains and cells of the search space has been made available to interpolation hashing through modification of the split conditions of linear hashing. When a chain is split into two, the split conditions of interpolation hashing distributes the keys into two chains such that the smaller keys remain in the same chain while the larger ones move to the new chain. In other words, the split conditions of interpolation hashing examine the leftmost bits of the key to determine which chain should contain the key, in the same way as the EXCELL method does.

Another similarity of interpolation hashing to the EXCELL method is found in the way it implements hashing. Both methods produce keys for k-dimensional data by shuffling their k coordinates, i.e., by taking the binary representation of the coordinates for each of the k axes and interleaving them. The shuffling of coordinates, together with the modified split conditions, produces for interpolation hashing a grid partition of the search space at intervals determined by a radix. Consequently, as in the EXCELL method, cell division lines with interpolation hashing are always equidistant, and the axis for a split is chosen in a cyclic fashion. Hence, the cells of the search space have the same shape as those of the EXCELL method; namely, the cells are either square or rectangular with their length twice as long as their width.

The major difference between interpolation hashing and the EXCELL method is that the correspondence between cells and chains is given by formulas rather than by a directory. Furthermore, with interpolation hashing the relationship between chains and cells of the study area is a one-to-one mapping in contrast to the one-to-many mapping between buckets and cells with the EXCELL method. It should be also noted that with interpolation hashing there exist at most two different sizes of cells at any moment. Besides, with interpolation hashing the number of cells of the search space grows by one while with the EXCELL method the number of cells grows by doubling.

Comparison

Each of the cell methods discussed so far has its advantages and disadvantages. Methods based on linear hashing provide for linear growth of the file without using a directory. In contrast, methods based on extendible hashing have overhead in terms of a directory. Particularly, with the EXCELL method, the directory may become large and unwieldy if the data is not evenly distributed over the search space.

Linear hashing based methods, however, usually have overflow buckets attached to their chains and may result in a file with unbalanced bucket occupancy. Especially, a non-uniform distribution of data may force linear hashing based methods to keep many overflow buckets as well as many underfilled primary buckets at the same time. On the other hand, with methods based on extendible hashing, the occupancy of data buckets is significantly more uniform because extendible hashing leads to a partition of the search space into blocks of a variable number of cells.

Each method behaves well for data that is uniformly distributed over the study area. The worst case occurs when data is non-uniformly distributed. In order to improve the performance in the worst case, it is desirable to develop a method that provides a compromise in the tradeoff between extendible hashing based methods and linear hashing based methods, namely, the tradeoff between the overhead of maintaining a large directory and the advantages of balancing bucket occupancy.

If it can be assumed that the data density of a particular region is more or less the same as that of its neighbours, a file structure that combines the principles of extendible hashing and linear hashing may be useful. That is, a new cell method may be developed by modifying linear hashing so that it has a directory and resolves collisions by splitting the address space instead of chaining until the directory reaches a predefined maximum depth. Typically, the maximum depth of a directory will be defined such that the directory may reside in main memory. Equivalently, a new method may be developed by modifying extendible hashing so that when the depth of the directory exceeds the predefined value, chaining of the overflow address is used instead of indefinitely expanding the directory. This new cell method will reduce the effect of a single local variation of data density on the file structure, achieve more uniform bucket occupancy than when linear hashing alone is applied, and retrieve each record in a file with one disk access in general. From this point of view, a new cell method which merges the concepts of extendible hashing and linear hashing is proposed. The resulting file scheme incorporates the features of both the EXCELL method and interpolation hashing.

3. Definitions and Notation

Let the study area be an image of 2^n x 2^n unit square pixels that intersects a polygon network, and let each of the pixels have a *color* or name associated with it. Furthermore, let the polygon network be represented by a linear quadtree. To yield an arbitrary but consistent total ordering among the blocks of a quadtree, an *order preserving hash function*, *s*, is used. The key produced by s is essentially the same as the locational code of the block in a linear quadtree [5]. Now, the file that represents a quadtree encoded polygon network is given by a set of 3-tuples containing the key, size and color for each quadtree node.

In order to structure a file using an adaptive cell method, the study area is partitioned into a set of *blocks* and/or *subblocks* which are defined in the following way:

A *block* of *depth d*, $0 \leq d \leq maxd \leq 2n$, where *maxd* is the predefined maximum depth of a block partition, is a rectangular region in the study area with a standard shape and a standard location that are the same as those of a region produced by recursively halving the study area *d* times with lines alternately perpendicular to the x and y axes. A block with its depth equal to *maxd* is called a *minimal block*.

A *subblock* of *depth d*, where $maxd < d \leq 2n$, is a rectangular region in the study area with a shape and location that are the same as those of a region produced by recursively halving the study area *d* times with lines alternately perpendicular to the x and y axes. Within each *minimal block* there exist at most two different depths of subblocks, i.e., d' and d'' such that $d'' = d' + 1$.

Throughout this paper, *maxd* will be used to denote the predefined maximum depth of a block partition.

A *fixed data bucket* is a bucket which contains no more than a predefined number of records, *b*, and an *expandable data bucket* is a bucket which may contain more than *b* records by attaching one or more overflow fields to it. An *adaptive cell method* of organizing a file is an abstract data type that:

1. guarantees:

a) for every cell and every record contained therein that the order of the keys of the records is the same as the order of the indices of the cells.

b) for every subblock of a minimum block and every record contained therein that the order of the keys of the records is the same as the order of the indices of the subblocks.

2. asserts that every block of depth *d* has exclusively one fixed data bucket associated with it if $d < maxd$; otherwise (the case of a minimal block), it has associated with it either a single fixed bucket exclusively or two or more expandable buckets that are contiguously located in physical memory such that:

a) each expandable bucket is exclusively associated with exactly one subblock of the minimal block, and

b) the overall *load factor*, i.e., the ratio of the number of existing records to the number of slots available, of these expandable buckets is within some predefined range.

4. Mapping between Regions and Data Buckets

Ordinarily the set of records in a file is distributed over a number of data buckets, and each data bucket has associated with it a block or a subblock in the study area. The mapping between blocks and data buckets is achieved by a directory. A directory is a set of elements, each of which corresponds to a cell of size 2^{n-d}, where *d* is the maximum of the depths of the existing blocks. Thus, a directory has associated with it a depth whose value is the same as *d*.

Each element of a directory has a pointer to a data bucket, or a set of buckets, that contains records describing the quadtree leaf nodes which intersect the corresponding cell in the study area. At depth *d* of a directory, there are altogether 2^d pointers, indexed from 0 to 2^d-1, which are not necessarily unique. The pointers of a directory are indexed in such a manner that a data bucket or a set of data buckets pointed to by a pointer with an index *i* contains all the records whose keys are prefixed with bits that are identical to the binary representation of *i*. That is, a data bucket or a set of data buckets pointed to by pointer 0 contains all the keys that start with *d* consecutive "0" bits, a data bucket pointed to by pointer 1 contains all the keys that start with *d* - 1 consecutive "0" bits followed by a "1" bit, and so on. Thus, the pointer *i* is guaranteed to find all the keys whose first *d* bits agree with the binary representation of *i*. This indexing scheme is in fact equivalent to a Morton sequence [11] and naturally satisfies the first of the requirements in Section 3.

The mapping between directory elements and data buckets, or sets of data buckets, is many-to-one. Note that all the records contained in a data bucket of depth *d* have the same bit pattern in their leftmost *d* bits. The correspondence between subblocks and expandable data buckets, however, is not shown in the directory. That is, the corresponding pointer in the directory points to the starting

address of a set of buckets that are physically located together, but it does not specify the correspondence between each of the subblocks and data buckets.

How the mapping between subblocks and data buckets are achieved will now be described. The subblocks of a minimal block are indexed in a similar manner as the cells corresponding to directory elements are indexed. In fact, when all the subblocks are the same size, they are indexed in exactly the same manner. However, when there exist two different sizes of subblocks, the larger subblocks have two candidates for their index. In that case, the smaller of the two is selected for the index of the subblock. As a result, when subblocks are of different sizes, the indexes of subblocks are not continuous.

Every subblock has an expandable data bucket associated with it. The keys of the records contained in a subblock of depth d have the same bit pattern in their leftmost d bit places. More explicitly, their leftmost *maxd* bits agree with the index of the minimal block the subblock belongs to, and the next (d - *maxd*) bits agree with the index of the subblock itself. Now, the mapping between subblocks and buckets is achieved by numbering each of the buckets that belongs to the same minimal block in a specific way as follows: Let a data bucket D be associated with a subblock whose index is i. Furthermore, let the maximum depth of existing subblocks be d. Then, i can be represented by a bit string S which is (d - *maxd*) bits long. Next, let k be a number represented by a bit string S' which is the reversed bit string of S. Then, k is the bucket number of D, i.e., D is the $(k + 1)st$ of the set of buckets that are contiguously located. The proof of this "reversed bit pattern" relation between i and k can be easily shown by induction, see [2] for a formal proof.

The proposed file organization scheme allows the file structure to adapt its shape automatically to the nature of the data to be stored, i.e., the amount and the distribution pattern. The adaptability of the scheme is obtained mainly by a dynamic partition of the data space, which is implemented by *splitting* and *merging* mechanisms, the details of which are given in [4]. As for the performance of the proposed scheme in terms of the variety of spatial queries supported and the access efficiency for single record retrieval, see [3, 4].

5. Conclusion

In conclusion, hashing techniques have been reviewed with regard to their use in organizing spatial data. It has been shown how extendible hashing and interpolation hashing can be combined to obtain better performance in terms of the number of disk accesses for single record retrieval. The resulting scheme is an adaptive cell method that normally retrieves a record with a given key with one disk access or at most two, maintaining a high storage utilization ratio. Additionally, compared with the B^+-tree structure which is often used in organizing quadtree nodes, the proposed scheme has the advantage of being a cell method in that set operations and range search can be efficiently performed.

6. Acknowledgement

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada under Grant A7634. The assistance, help and council of Chung Hee Hwang is also gratefully acknowledged.

7. References

1. Bayer, R. & E. McCreight, "Organization and Maintenance of Large Ordered Indexes", *ACTA Informatica*, vol. 1, pp. 173-189, 1972.
2. Burkhard, W. A., "Interpolation-Based Index Maintenance", *BIT*, Vol. 23, pp. 274-294, 1983.
3. Davis, W. A., & C. H. Hwang, "Organizing and Indexing for Spatial Data", to be presented at the 2nd International Conference on Spatial Data Handling, Seattle, July 1986.
4. Davis, W. A., & C. H. Hwang, *File Organization Schemes for Geometric Data*, TR 85-14, Dept. of Computing Science, Univ. of Alberta, 1985.
5. Davis, W.A., & X. Wang, "A New Approach to Linear Quadtrees", *Proceedings of Graphics Interface '85*, Montreal, pp. 195-202, May 1985.
6. Fagin, R. J., et al., "Extendible Hashing - A Fast Access Method for Dynamic Files", *ACM TODS*, Vol. 4, pp. 315-344, Sep. 1979.
7. Knuth, D.E., *Sorting and Searching, The Art of Computer Programming*, V.3, Addison-Wesley, Reading, Mass, 1973.
8. Larson, P.-A., "Dynamic Hashing", *Bit*, Vol. 18, pp. 184-201, 1978.
9. Litwin, W. "Virtual Hashing: A Dynamically Changing Hashing", *Proc. 4th Int. Conf. VLDB*, pp. 517-523, 1978.
10. Litwin,W., "Linear Hashing: A New Tool for File and Table Addressing" *Proc. 6th Int. Conf. VLDB*, pp. 212-223, 1980.
11. Samet, H., "The Quadtree and Related Hierarchical Data Structures", *Comput. Surveys*, Vol. 16, pp. 187-260, Jun. 1984.
12. Tamminen, M., "Efficient Spatial Access to a Data Base", *Proc ACM-SIGMOD*, pp. 200-206, 1982.
13. Tamminen, M., "The Extendible Cell Method for Closest Point Problems", *BIT*, Vol. 22, pp. 27-41, 1982.
14. Tamminen, M., "Performance Analysis of Cell Based Geometric File Organizations", *Computer Vision, Graphics, and Image Processing*, Vol. 24, pp. 160-181, 1983.